METHOD FOR RESOLVING ADDRESS SPACE CONFLICTS BETWEEN A VIRTUAL MACHINE MONITOR AND A GUEST OPERATING SYSTEM.

Field of the Invention

The present invention relates generally to virtual machines, and more specifically to resolving address space conflicts between a virtual machine monitor and a guest operating system.

Background of the Invention

A conventional virtual machine monitor (VMM) typically runs on a computer and presents to other software the abstraction of one or more virtual machines. Each virtual machine may function as a self-contained platform, running its own "guest operating system" (i.e., an operating system hosted by the VMM). The guest operating system expects to operate as if it were running on a dedicated computer rather than a virtual machine. That is, the guest operating system expects to control various computer operations and have an unlimited access to the computer's physical memory and memory-mapped I/O devices during these operations. However, in a virtual machine environment, the VMM should be able to have ultimate control over the computer's resources to provide protection from and between virtual machines. To achieve this, the VMM typically intercepts and arbitrates all accesses made by the guest operating system to the computer resources.

With existing processors (e.g., IA-32 microprocessors), the VMM may not be able to intercept accesses of the guest operating system to hardware resources

unless a portion of the VMM code and/or data structures is located in the same virtual address space as the guest operating system. However, the guest operating system does not expect the VMM code and/or data structures to reside in the address space of the guest operating system and can attempt to access a region occupied by the VMM in this address space, causing an address space conflict between the guest operating system and the VMM. This conflict may result in abnormal termination of operations performed by the VMM or the guest operating system.

Thus, a mechanism is needed that will detect and resolve address space conflicts between a VMM and a guest operating system.

Brief Description of the Drawings

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

Figure 1 illustrates one embodiment of a virtual machine environment;

Figure 2 is a block diagram of a system for resolving address space conflicts between a virtual machine monitor and a guest operating system, according to one embodiment of the present invention;

Figure 3 is a flow diagram of a method for resolving address space conflicts between a virtual machine monitor and a guest operating system, according to one embodiment of the present invention;

Figure 4 is a flow diagram of a method for relocating a virtual machine kernel within a virtual machine address space, according to one embodiment of the present invention;

Figure 5 illustrates operation of a virtual machine kernel that supports guest deprivileging, according to one embodiment of the present invention;

Figure 6 is a flow diagram of a method for handling a virtualization trap generated by a guest operating system, according to one embodiment of the present invention; and

Figure 7 is a block diagram of one embodiment of a processing system.

Description of Embodiments

A method and apparatus for resolving address space conflicts are described. In the following description, numerous details are set forth, such as distances between components, types of molding, etc. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present invention.

In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention can be practiced without these specific details.

Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times,

principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer

system bus. Instructions are executable using one or more processing devices (e.g., processors, central processing units, etc.).

The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose machines may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope

of the present invention is defined only by the appended claims, along with the full scope of equivalents to which such claims are entitled.

The method and apparatus of the present invention provide a mechanism for resolving address space conflicts between a guest operating system and a virtual machine monitor (VMM). **Figure 1** illustrates one embodiment of a virtual machine environment 100, in which the present invention may operate. In this embodiment, bare platform hardware 116 comprises a computing platform, which may be capable, for example, of executing a standard operating system (OS) or a virtual machine monitor (VMM), such as a VMM 112. A VMM, though typically implemented in software, may export a bare machine interface, such as an emulation, to higher level software. Such higher level software may comprise a standard or real-time OS, although the invention is not limited in scope in this respect and, alternatively, for example, a VMM may be run within, or on top of, another VMM. VMMs and their typical features and functionality are well-known by those skilled in the art and may be implemented, for example, in software, firmware or by a combination of various techniques.

As described above, a VMM presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs). **Figure 1** shows two VMs, 102 and 114. Each VM includes a guest OS such as a guest OS 104 or 106 and various guest software applications 108-110. Each of the guest OSs 104 and 106 expects to control access to physical resources (e.g., memory and memory-mapped I/O devices) within the hardware platform on which the guest OS 104 or 106 is running and to perform other functions. However, in a virtual

machine environment, the VMM 112 should be able to have ultimate control over the physical resources to provide protection from and between VMs 102 and 114. The VMM 112 achieves this goal by intercepting all accesses of the guest OSs 104 and 106 to the computer's physical resources. For instance, a guest deprivileging technique may be used to enable the VMM 112 to intercept the above accesses. Guest deprivileging forces all guest software to run at a hardware privilege level that does not allow that software access to certain hardware resources. As a result, whenever the guest OS 104 or 106 attempts to access any of these hardware resources, it "traps" to the VMM 112, i.e., the VMM 112 receives control over an operation initiated by the guest operating system if this operation involves accessing such hardware resources. It should be noted that any other technique known in the art may be used to transfer control over a similar operation from the guest OS 104 or 106 to the VMM 112.

When using guest deprivileging or other techniques enabling the VMM 112 to intercept accesses of the guest OSs 104 and 106 to the computer's physical resources, a portion of VMM code and/or data structures may be architecturally required to reside in the same virtual address space as each of the guest OSs 104 and 106. However, since the guest OSs 104 and 106 are unaware of the VMM's presence, they may attempt to access a region occupied by the VMM code and/or data structures in the virtual address space associated with the guest OS 104 or 106. Such an attempt may result in collision between the code and data structures of the guest OS and the VMM code and data structures in the virtual address space, causing an abnormal termination of an operation performed by the guest

OS 104 or 106, or the VMM 112. The present invention provides a mechanism for resolving such address space conflicts.

Figure 2 is a block diagram of a system 200 for resolving address space conflicts between a VMM and a guest OS, according to one embodiment of the present invention. System 200 includes bare platform hardware 214 that includes a computing platform capable of executing a guest OS (e.g., guest OS 104 or 106), a VMM (e.g., VMM 112), etc. Two separate address spaces 204 and 202 are allocated for guest software and the VMM. That is, VM address space 204 is allocated to hold code and data structures of the guest OS and other guest software, and VMM address space 202 is allocated for VMM code and data structures.

As described above, certain components of the VMM code and/or data structures may be architecturally required to reside in the same address space as the guest OS to enable the VMM to intercept the guest OS's accesses to hardware resources. For instance, for the IA-32 instruction set architecture (ISA), when guest deprivileging is used to ensure control of the VMM over the guest OS's accesses to hardware resources, an interrupt-descriptor table (IDT), which includes pointers to trap handling routines, is architecturally required to reside in the same address space as the guest OS. One embodiment of the present invention that supports guest deprivileging will be described in greater detail below in conjunction with **Figures 5 and 6**. For other ISAs, various other portions of VMM code and/or data structures may be architecturally required to reside in

the same space address as the guest OS to enable the VMM's control over accesses made by the guest OS to hardware resources.

In one embodiment, the VMM code and structures are divided into two portions. The first portion of the VMM includes a set of code and/or data structures that are required to reside in the address space of the guest OS, i.e., in the VM address space 204. The second portion of the VMM includes the remainder of the VMM code and data structures. In one embodiment, a software program (referred as a virtual machine kernel 210) collects a minimal set of the VMM code and/or data structures that are required to be located in the same address space as the guest OS. The remainder of the VMM code and data structures is compiled as a separate program and located in the VMM address space 202. The virtual machine kernel (VMK) 210 then maps itself into both the VM address space 204 and the VMM address space 202.

Subsequently, when the guest OS attempts to access a region occupied by the VMM code and/or data structures in the VM address space 204, the VMK 210 detects this attempt of the guest OS. In one embodiment, the VMK 210 receives control over an event initiated by the guest OS if this event may potentially cause an address space conflict between the guest OS and the VMM. Guest deprivileging or any other hardware or software mechanisms known in the art may be used to transfer control over such an event from the guest OS to the VMM code and/or data structures residing in the VM address space 204.

The VMK 210 then evaluates this event to determine its cause. Upon detecting that the event was caused by the attempt of the guest OS to access the

region occupied by the VMM code and/or data structures, the VMK 210 re-maps itself into a different region within the VM address space 204 to allow the guest OS to access the region previously used by the VMK 210. One embodiment of a method for relocating the VMK 210 within the VM address space 204 is described in greater detail below in conjunction with **Figure 4**.

Figure 3 is a flow diagram of one embodiment of a method 300 for resolving address space conflicts between a VMM and a guest OS, according to one embodiment of the present invention. Method 300 begins with dividing the VMM into a first portion and a second portion (processing block 304). As described above, the first portion includes a set of VMM code and/or data structures that are architecturally required to reside in the same address space as the guest OS. The second portion of the VMM includes the remainder of the VMM code and data structures. In one embodiment (described in greater detail below), the first portion of the VMM includes a set of trap handlers and an interrupt-descriptor table (IDT). In alternative embodiments, the first portion includes various other data structures and code of the VMM that must reside in the same address space as the guest OS.

Next, a first address space (i.e., VM address space 204) is created to hold code and data structures of the guest OS and other guest software (processing block 306), and a second address space (i.e., VMM address space 202) is created for the VMM code and data structures (processing block 308). In one embodiment, these address spaces are created during the boot process.

Further, the first portion of the VMM is mapped into both the VM address space and the VMM address space (processing block 310), and the second portion of the VMM is loaded into the VMM address space (processing block 312).

At processing block 314, an attempt of the guest OS to access a region occupied by the first portion of the VMM is detected. In one embodiment, such an attempt is detected by transferring control over an event initiated by the guest OS to the first portion of the VMM if the event may potentially cause an address-space conflict between the guest operating system and the VMM. One embodiment of detecting a potential address space conflict is described in greater detail below in conjunction with **Figures 5 and 6**.

Afterwards, at processing block 316, the first portion of the VMM is relocated to another region within the VM address space to allow access of the guest OS to the region previously occupied by the first portion of the VMM. Any subsequent attempt to access the new region occupied by the first portion of the VMM will again result in its relocation within the VM address space. One embodiment of a method for relocating a VMK, which contains the first portion of the VMM, is shown in **Figure 4**.

Referring to **Figure 4**, after an address space conflict between the guest OS and the VMM is detected (processing block 404), the VM address space is searched for an unused region (processing block 406). At decision box 408, a determination is made as to whether an unused region exists in the VM address space. If the determination is positive, the VMK containing the first portion of the VMM code and data structures is remapped into this unused region, and control is

transferred back to the guest OS, which may now access the region previously used by the VMK.

Alternatively, if no unused region exists in the VM address space, i.e., the guest OS has used the entire VM address space, then a random region is selected within the VM address space (processing block 412), the content of the memory located at the selected region is copied to a buffer in the VMM address space (processing block 414), and the VMK is remapped into the selected region in the VM address space (processing block 416). Subsequent memory accesses to this selected region (i.e., new VMK region) are serviced through emulated memory accesses from the buffer in the VMM address space that contains the original content of the new VMK region. In one embodiment, the frequency of such emulated memory references may be reduced by periodically relocating the VMK to random regions within the VM address space until a region is found that is infrequently used.

Figure 5 illustrates operation of a VMK that supports guest deprivileging, according to one embodiment of the present invention. As described above, guest deprivileging causes the guest OS to run at a lesser privileged level so that the guest OS "traps" to the VMM whenever it attempts to issue privileged instructions that operate on the processor system state. In one embodiment, the VMM supporting guest deprivileging installs pointers to trap handling routines (i.e., trap handlers 552) in the interrupt-descriptor table (IDT) 514. Some ISAs (e.g., IA-32 ISA) require that the IDT 514 be resident in the currently active virtual address space (i.e., VM address space 504). In one embodiment, the entries in the

IDT 514 are task gates, which provide an address space switch. That is, when a trap is generated, the IDT 514 is searched for a pointer to a trap handling routine. If this pointer is a task gate, it will enable a direct switch to the VMM address space, which contains a trap handling routine for the generated trap. Accordingly, a trap handler corresponding to a task gate does not need to reside in the VM address space, although the task gate itself must reside in the VM address space. In another embodiment, entries in the IDT 514 are trap gates or interrupt gates, which do not provide address space switches. Consequently, trap handlers associated with such IDT entries must reside in the VM address space. In addition, the VMM may place shadow versions of other data structures (e.g., global descriptor table) in the VM address space.

In one embodiment, the VMK 510 collects together a minimal set of trap handlers and/or data structures (e.g., the IDT 514) that must be located in the VM address space, maps them into both the VM address space 504 and the VMM address space 502, and sets access rights of the pages holding the VMK 510 to the most privileged level (e.g., the "supervisor" privilege level with ring =0 for IA-32 microprocessors). As described above, the guest OS runs in the deprivileged mode (e.g., the "user" mode with ring=3 for IA-32 microprocessors). As a result, in one embodiment, the guest OS generates virtualization traps whenever it attempts to access privileged machine resources, including the pages holding the VMK 510 that are protected with the most privileged access rights.

In one embodiment, when a virtualization trap is generated, the IDT 514 is searched for a corresponding pointer to a trap handler. In one embodiment, a

trap may need to be handled by the VMM-resident trap handler. In this embodiment, the VMK performs two address space switches - one switch to deliver the trap to the trap handler in the VMM address space 502, and a second switch to transition back to the VM address space 504 after the trap has been serviced by VMM-resident trap-handler.

Alternatively, a trap can be handled in a VMK-resident handler. For instance, a trap may be caused by an instruction of the guest OS to reset a flag in the processor's register. Such a trap can be handled entirely in the trap handler 552, without transferring control to the VMM in the VMM address space 502, and such an implementation would result in better performance.

One type of virtualization traps is a conflict fault which is generated when the guest OS attempts to access a region of the VM address space 504 that is currently in use by the VMK 510. The VMK 510 handles these conflict faults by re-mapping itself into a new region within the VM address space 504 as described in greater detail above in conjunction with **Figure 4**.

Figure 6 is a flow diagram of a method 600 for handling virtualization traps generated by a guest OS, according to one embodiment of the present invention. Method 600 begins with setting access rights of the region occupied by the VMK to a more privileged level than a privilege level associated with the guest OS (processing block 604). For instance, all VMK pages may be mapped with supervisor-only privilege (ring=0) and the guest OS may be set to run in the deprivileged user mode (ring=3).

At processing block 606, a trap generated by the guest OS is received. The trap is caused by an attempt of the guest OS to access privileged hardware resources. At decision box 608, a determination is made as to whether the trap can be handled internally by the VMK (e.g., in a VMK-resident trap handler). If the trap is too complex to be handled by the VMK, it is delivered to the VMM address space (e.g., to a VMM-resident trap handler) (processing block 610) and then resumed back to the VM address space after the trap has been serviced by the VMM (processing block 612). Afterwards, control over the event that caused the trap is returned to the guest OS (processing block 620).

Alternatively, if the trap can be handled internally by the VMK, a determination is made as to whether the trap was caused by an address space conflict between the VMK code and data structures and the code and data structures of the guest OS (decision box 614). If the trap was indeed caused by an address space conflict, the VMK code and data structures are relocated to a new region within the VM address space (processing block 618). Alternatively, the trap is handled in a corresponding trap handler (processing block 616). Afterwards, control over the event that caused the trap is returned to the guest OS (processing block 620).

Figure 7 is a block diagram of one embodiment of a processing system. Processing system 700 includes processor 720 and memory 730. Processor 720 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. Processing system 700 can be

a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other system that includes software.

Memory 730 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, or any other type of machine medium readable by processor 720. Memory 730 can store instructions for performing the execution of the various method embodiments of the present invention such as methods 300, 400 and 600 (**Figures 3, 4 and 6**).

It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

CLAIMS

What is claimed is:

1. A method comprising:

detecting that a guest operating system attempts to access a region occupied by a first portion of a virtual machine monitor (VMM) within a first address space; and

relocating the first portion of the VMM within the first address space to allow the guest operating system to access the region previously occupied by the first portion of the VMM.

2. The method of claim 1 wherein the first portion of the VMM includes a set of VMM code and data structures that are architecturally required to reside in the first address space.

3. The method of claim 1 wherein the first portion of the VMM includes a set of trap handlers and an interrupt-descriptor table (IDT).

4. The method of claim 1 further comprising:

dividing the VMM into the first portion and a second portion;

creating the first address space associated with the guest operating system;

creating a second address space associated with the VMM;

locating the second portion of the VMM in the second address space associated with the VMM; and

mapping the first portion of the VMM into the first address space and the second address space.

5. The method of claim 1 further comprising:

receiving control over an event initiated by the guest operating system when the event may potentially cause an address space conflict between the guest operating system and the VMM.

6. The method of claim 5 wherein receiving control further comprises:

setting access rights of the section occupied by the first portion of the VMM to a more privileged level than a privilege level associated with the guest operating system; and

receiving a trap caused by an attempt of the guest operating system to access a hardware resource having a higher privilege level than the privilege level associated with the guest operating system.

7. The method of claim 6 further comprising:

determining that the trap can be handled by the first portion of the VMM;

executing code associated with the trap; and

returning control over the event to the guest operating system.

8. The method of claim 6 further comprising:

determining that the trap should be handled by the second portion of the VMM;

delivering the trap to the second portion of the VMM;

passing control over the event to the guest operating system after code associated with the trap was executed by the second portion of the VMM.

9. The method of claim 1 wherein relocating the first portion of the VMM further comprises:

finding an unused region within the first address space; and

re-mapping the first portion of the VMM into the unused region.

10. The method of claim 1 wherein relocating the first portion of the VMM further comprises:

determining that no unused region exists within the first address space;

selecting a random region within the first address space;

copying content of a memory located at the random region to the second address space; and

re-mapping the first portion of the VMM into the random region.

11. The method of claim 10 further comprising:

receiving control over an event initiated by the guest operating system, the event corresponding to an attempt of the guest operating system to access the content of the memory previously located at the random region; and

accessing the copied content of the memory in the second address space.

12. The method of claim 11 further comprising periodically relocating the first portion of the VMM to random regions within the first address space until finding a region that is infrequently accessed.

13. An apparatus comprising:

a first address space associated with a guest operating system;

a second address space associated with a virtual machine monitor (VMM); and

a virtual machine kernel to detect that the guest operating system attempts to access a region occupied by a first portion of the VMM within the first address space and to relocate the first portion of the VMM within the first address space to allow the guest operating system to access the region previously occupied by the first portion of the VMM.

14. The apparatus of claim 13 wherein the first portion of the VMM includes a set of VMM code and data structures that are architecturally required to reside in the first address space.

15. The apparatus of claim 13 wherein the first portion of the VMM includes a set of trap handlers and an interrupt-descriptor table (IDT).

16. The apparatus of claim 13 wherein the virtual machine kernel is to divide the VMM into the first portion and the second portion, to locate the second portion of the VMM in the second address space associated with the VMM, and to map the first portion of the VMM into the first address space and the second address space.

17. The apparatus of claim 13 wherein the virtual machine kernel is to receive control over an event initiated by the guest operating system when the event may potentially cause an address space conflict between the guest operating system and the VMM.

18. The apparatus of claim 13 wherein the virtual machine kernel is to receive control by setting access rights of the section occupied by the first portion of the VMM to a more privileged level than a privilege level associated with the guest operating system, and by receiving a trap caused by an attempt of the guest operating system to access a hardware resource having a higher privilege level than the privilege level associated with the guest operating system.

19. The apparatus of claim 18 wherein the virtual machine kernel is to further determine that the trap can be handled by the first portion of the VMM, to execute

code associated with the trap, and to return control over the event to the guest operating system.

20. The apparatus of claim 18 wherein the virtual machine kernel is to further determine that the trap should to handled by the second portion of the VMM, to deliver the trap to the second portion of the VMM, and to pass control over the event to the guest operating system after code associated with the trap was executed by the second portion of the VMM.

21. The apparatus of claim 13 wherein the virtual machine kernel is to relocate the first portion of the VMM by finding an unused region within the first address space and re-mapping the first portion of the VMM into the unused region.

22. The apparatus of claim 13 wherein the virtual machine kernel is to relocate the first portion of the VMM by determining that no unused region exists within the first address space, selecting a random region within the first address space, copying content of a memory located at the random region to the second address space, and re-mapping the first portion of the VMM into the random region.

23. The apparatus of claim 13 wherein the virtual machine kernel is to receive control over an event initiated by the guest operating system, the event corresponding to an attempt of the guest operating system to access the content of

the memory previously located at the random region, and to access the copied content of the memory in the second address space.

24. The apparatus of claim 13 wherein the virtual machine kernel is to periodically relocate the first portion of the VMM to random regions within the first address space until finding a region that is infrequently accessed.

25. A system comprising:

a memory to include a first address space associated with a guest operating system and a second address space associated with a virtual machine monitor (VMM); and

a processor, coupled to the memory, to detect that the guest operating system attempts to access a region occupied by a first portion of the VMM within the first address space and to relocate the first portion of the VMM within the first address space to allow the guest operating system to access the region previously occupied by the first portion of the VMM.

26. The system of claim 25 wherein the first portion of the VMM includes a set of VMM code and data structures that are architecturally required to reside in the first address space.

27. The system of claim 25 wherein the first portion of the VMM includes a set of trap handlers and an interrupt-descriptor table (IDT).

28. A computer readable medium that provides instructions, which when executed on a processor, cause said processor to perform operations comprising:

detecting that a guest operating system attempts to access a region occupied by a first portion of a virtual machine monitor (VMM) within a first address space; and

relocating the first portion of the VMM within the first address space to allow the guest operating system to access the region previously occupied by the first portion of the VMM.

29. The computer readable medium of claim 28 comprising further instructions causing the processor to perform operations comprising:

finding an unused region within the first address space; and

re-mapping the first portion of the VMM into the unused region.

30. The computer readable medium of claim 28 comprising further instructions causing the processor to perform operations comprising: determining that no unused region exists within the first address space;

selecting a random region within the first address space;

copying content of a memory located at the random region to the second address space; and

re-mapping the first portion of the VMM into the random region.