Study of Virtualization & Management of Memory in Virtualized Systems

A Seminar Report

submitted in partial fulfillment of the

requirements for the degree of

Master of Technology

by

Aby Sam Ross
143050093

Computer Science & Engineering
Indian Institute of Technology Bombay
Mumbai 460076 (India)

April 2015

Abstract

Memory like other computing resources can be considered a scarce enough commodity. Multiplexing and management of this memory among various stakeholders in native computing environments has been well taken care of. But when it comes to virtualized environments it adds another layer of abstraction. This calls for new techniques, for partitioning and arbitrating memory among different VMs, that doesn't change the OS's perspective of the scheme of things and requires minimal changes. While the techniques for partitioning memory among different VMs in a virtualized setting has almost got standardized, there are still differing views and different approaches to managing memory. The new layer of abstraction introduced by virtualization adds considerable complexity in estimating the needs and utilization of memory. One cannot retrofit one strategy suitable for a particular setting to another and expect to get a clear picture of memory related parameters. This seminar is to understand different techniques of memory virtualization and to gain some insight into some of the existing management strategies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview of Virtualization

Hardware resource abstraction and management have always been a major aim and challenge in electronic systems. With the advent of computers these tasks fell upon a huge piece of software called the Operating System. It became the responsibility of the OS to multiplex the resources among the different processes running in a system. An obvious illustration is that of CPU multiplexing, where the CPU is scheduled in time slices among various processes according to priority. The physical memory in a system as well is a resource. Memory also needs to be abstracted and suitable interfaces are to be provided for easy and efficient use. Traditionally this has been achieved through the concept of virtual memory coupled with paging or segmentation.

With advancements in technology the capabilities of physical resources increased and cost decreased. Processors and memory became faster, smaller chips with more capacity became available. But these advancements didn't percolate naturally into better utilization of these resources. Often these resources were underutilized. In order to get better utility from these resources various options of sharing these resources at a larger granularity was explored. Thus came into existence the concept of virtualization.

Virtualization increases the granularity of abstraction from individual resources to abstracting the entire set of hardware as a single unit or in other words virtualization abstracts the entire computer system. With this we could have more than one machine running on top of the existing computing hardware. These machines came to be called *Virtual Machines* or VM. In other words with the increase in granularity of abstraction

the unit of allocation to the abstraction increased from a process to an entire OS.
The ringmaster who runs the show in a Virtual Machine is still the humongous piece of code called the OS. But traditionally OSes were designed to have ownership and control of the underlying hardware. Virtualization now created a separation between the hardware and the OS.They were now relegated to the status of a *guest OS*. A single OS was no longer the sole owner and controller of the resources. The entire hardware had to be abstracted, interfaced and multiplexed among different OSes manning different VMs analogous to the way in which an OS enabled multiplexing of resources among different processes in a native system. This role was now taken up by a new, but no less hideous, software layer called the *Virtual Machine Monitor* (VMM) or the *Hypervisor*. That is now we have the hypervisor sitting on top of the hardware directly and above it we have different VMs.
The introduction of a new orchestrator, the Hypervisor, and relegation of a guest OS to a lesser privilege brought about new challenges. The guest OS in a VM had no longer complete control of the underlying resources to arbitrate efficiently among the processes running in it. But to the process local to a VM the guest OS was still the ringmaster who ran the show. Hence it became of paramount importance that the introduction of a software layer, the hypervisor, between the guest OS and the resources didn't break the equivalence view i.e. a process running in a VM should see no or little difference in running on a native vs. virtualized system. At the same time we also had to ensure that every VM stayed within its allocated bounds and guest OS operations had enough efficiency. These requirements of hypervisor design are formally stated in Popek and Goldberg (1974).The hypervisor can choose from among the various strategies like instruction interpretation, trap & emulate, binary translation, para-virtualization, hardware assisted virtualization to provide the aforementioned requirements.
Once such a hypervisor is available efficient resource utilization and management comes next in order to meet the original design principles of virtualization. Memory like other resources will also be apportioned to the different VMs. But it is not necessary that all of the memory allocated to a VM will be in 100% use all the time. This gives us an opportunity to reallocate this unused memory to other VMs in need. But its sharing and management is not as simple as that of a flexible resource like CPU and nor are the effects of differences in the amounts of memory available that easily visible (Hwang *et al.*, 2013). The objective of this seminar is to get an understanding of the intricacies involved in this.

1.2 Scope of the Seminar

The objective of this seminar is not to delve deep into the implementation of virtualization as a whole but rather to concentrate on understanding how memory virtualization is achieved, the pros and cons of different techniques of memory virtualization, optimizations to it, challenges of memory management in virtualized systems, memory reallocation mechanisms, and some of the existing memory management strategies.

Chapter 2

Background

2.1 Memory Management in Native Systems

Memory management in non-virtualized native multiprogramming system is achieved using Virtual Memory. In multiprogrammed systems several programs are resident in memory at the same time. The memory management policy in such a system deals with protecting the memory of one program from another, loading a program into available space in main memory, (de)allocating memory dynamically from/to programs.

While programs are complied and linked with addresses starting at 0 and CPU uses these addresses to access the binary, it isn't necessary (and is the not the case that) that a program will get physical memory with the same addresses or it is not even guaranteed that there will be enough space in the physical memory to load the entire binary of the program. Most of the times only the immediately needed part of the program is loaded onto the available physical memory, (the rest will be held in a backing store - often the hard disk) and they will share the physical memory space along with parts of other programs. And the OS may choose to evict parts of the program to the backing store when in need of memory as a part of its memory management policy. Therefore the addresses generated by CPU needs to be translated into the corresponding physical addresses. The address generated by CPU is called *Virtual Address* or VA . Thus we need to have a mechanism of *Virtual Address (VA)* $\rightarrow$ *Physical Address (PA)* translation.

Hence the illusion that is given to a program that there is enough physical memory available to store its entire binary combined with relocation of code having contiguous virtual addresses into - not necessarily contiguous - available physical memory chunks are the central ideas of virtual memory.

Virtual Memory can be implemented in more than one ways. Paging is one of them. The idea behind paging is to divide the virtual address space and the physical memory into same sized units of allocation called *pages*. Hence the virtual address space is divided into *virtual pages* or simply *pages* and physical memory into *physical pages* or *frames* or *physical/machine frames*. It is thus clear that not all pages of a program will be present in the physical memory when the program is executing, the contiguous virtual pages belonging to the same program needn't be allocated contiguous frames and it is necessary to provide a way to map from the virtual pages to the associated machine frames.

This mapping should be there for each program and this mapping is called the *page table*. The organisation of page table is closely tied to the size of a page or a frame. Page size is taken as *power of 2* as this ensures that all binary representable addresses can be utilized and address manipulation can be done without arithmetic operations. There should an be an entry for all virtual page addresses of a program in its page table. This will result in a very long (large) page table and all the page tables of all programs that are currently resident in memory will consume considerable amount of physical memory. So instead of storing such a single large page table per program in memory we break the page tables into different levels to have a tree like structure. Thus following the design principles of virtual memory it is not necessary that even all the levels of the page table will be resident in memory all the time.

Often the entries in each level are grouped into different sets. And each entry will contain a physical address and some flag bits. The flag bits store permissions and other info about the this entry and the physical address in the entry points to a set of entries in the next level. These sets of entries in each level are often limited to single physical frame. Thus the physical address in a page table entry points to a physical frame in the next level. There is an exception for the last level of the page table. An entry in the last level contains the physical address of the actual virtual page that we were looking for and the flag bits of the entry include information like whether the physical page it points to is present or not. The physical address of the root of the page table (i.e. the physical address of the base of the outermost level of page table) is stored in a hardware register.

The translation of virtual address to physical address happens in the following manner:

The base address of the program's page table is obtained from the hardware register storing it. To this base address we add that part of the virtual address that corresponds to the outermost level. Thus we get the corresponding entry in the outer most level which points to the base address of the corresponding set of entries of the next level (i.e. a physical page of the next level) . Now to get to the corresponding entry within that physical page we add the part of virtual address for the next level to the physical page address we obtained from the outermost level. This process is continued till we get to a last level entry from which we get the physical frame address corresponding to the virtual address. And to go to the exact physical address location corresponding to the virtual address location add the last part of the virtual that doesn't correspond to any page table level i.e. the offset part to the physical frame address.

Hence it is clear from the above discussion that the parts of the virtual address that correspond to each level of page table give us an offset within that level and last part of the virtual address that doesn't correspond to any page table level gives us the offset within the actual frame that holds this virtual address. Illustration in figure 2.1.

For a virtual address generated by the CPU the translation to physical address i.e. traversing the levels of page table happens in hardware. If the corresponding physical page is present in the translated location it is a *hit* else it is a *fault*. Page faults are to be resolved by moving in the corresponding missing page from the backing store. Certain architectures like x86 choose to cache recently accessed virtual addresses and their mappings in a small cache often called the *translational look ahead buffer* or TLB, this is for faster translations the next the same address is accessed. This also brings about the need for coherence between the entries in the TLB and the page table. The hardware that does all this is called the *memory management unit* or MMU.

Linux OS on x86 (i.e. 32 bit) architecture has 3 levels of the page table. While for x86_64 (i.e. 64bit) it has 4. They are called *page global directory* or PGD, *page upper directory* or PUD, *page middle directory* or PMD and *page table entry* or PTE. x86 won't have the PMD. The *CR3* register holds the base address of the PGD.

2.2 Challenges to Virtualizing Memory

When we introduce virtualization the physical memory of the system is to be shared among different VMs. In addition to that there should be ways to dynamically (re)allocate memory among VMs analogous to the way in which memory is managed among programs in a native system as discussed in the previous sections. Hence the basic idea of

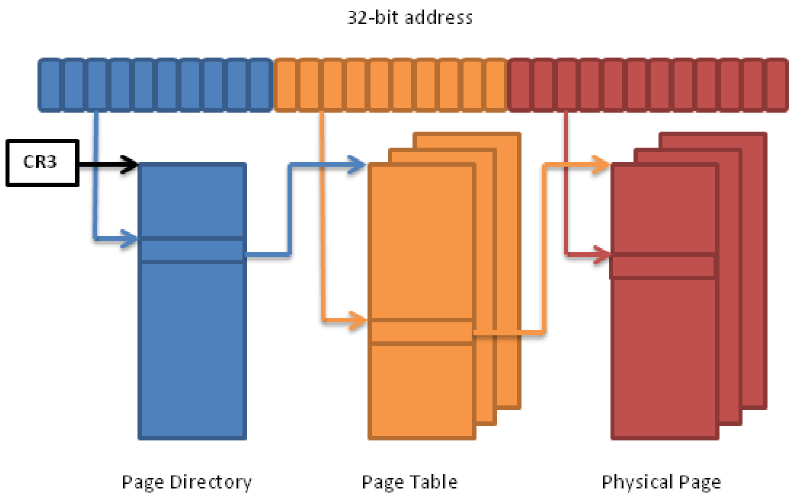

Figure 2.1: Illustration of *VA* → *PA* address translation.
`corensic.files.wordpress.com/2011/11/virtualmemory.png`

memory virtualization is similar to virtual memory in native systems. Here on top of the virtual memory abstraction that happens inside a VM we are introducing another layer of virtual memory sort of abstraction. If we want minimal changes to a stock guest OS or if the guest OS is unaware of the underlying hypervisor then the guest OS's perceived view of physical memory should be kept intact by allowing it to maintain the mapping of virtual addresses to what the guest OS perceives as its physical memory. While in reality what the guest OS perceives as physical memory is not the real machine memory and neither is it given unrestricted direct access to the entire real machine memory. This was done to ensure that guest OSs stay within their bounds and VM isolation was guaranteed. Moreover guest OSs no longer managed the resources on their own.

This was achieved by virtualizing the MMU or in other words by introducing MMU functionalities in the hypervisor level that backed the the guest OS memory management operations, trying as far as possible not to break the guest OS's view of the affairs, and that did effective multiplexing of actual machine memory among the different VMs.

Implementing this is not as easy as said. The first challenge lies in deciding how to divide or partition machine memory among different VMs. Dictating the techniques of access poses another. Finally the dynamic management aspect comes in.

Every VM can be allocated simple, static, contiguous, disjoint fixed partitions or more of a dynamic allocation scheme similar to demand paging discussed in previous section can be adopted. The former has obvious drawbacks of memory wastage if the allocated memory is underutilized, incapability to support a VM requiring more memory than the static allocation etc. All this will affect the number of VMs that can be hosted on a machine. In addition it cannot support other virtualization techniques like migration. The latter ensures better utilization of memory, enables memory over-commitment among VMs, similar to the one provided to different processes by normal virtual memory, but introduces considerable complexity in implementation. One reason for this complexity is the varying levels of dynamism in memory demands among different VMs which requires the VMM to detect and react quickly to these demands by adapting its policies for each VM. Another cause of this complexity is multiple memory resource control entities taking decisions independently; one the guest OS and other the VMM. This can often result in conflicting decisions being made that increases the memory management overhead.

Since the guest OS employs virtual memory, access to the MMU is often needed. But we also know the actual MMU functionalities that manages the machine memory lies in the hyervisor. Thus hypervisor needs to define ways for the guest OS to access these functionalities without breaking VM isolation.

The challenges of implementing memory management at the hypervisor comes next. First, a VMM needs to accurately determine memory resource usage statistics of individual VM to take any management policy related decision. Second, policy decisions taken at the VMM level can clash with the decisions taken within a guest OS. A good example of this the is *double paging* explained in Waldspurger (2002). Third, it is difficult to define a one-copy-fits-all management strategy for the various types of guest OSs and applications running in different VMs. For e.g. an application like DB which manages its own memory will be affected badly by a VMM memory management strategy that transparently reallocates memory assigned to it because for efficient working these applications need a current, consistent view of the memory allocated to it (Salomie *et al.*, 2013).

We now try to look into the existing solutions and implementations of various aspects of memory virtualization that address many of these issues.

Chapter 3

Memory Virtualization

3.1 Physical Memory Allocation Model in Virtualized Systems

Since within a VM the guest OS provides its application the virtual memory abstraction and processes running in the guest OS share what the guest OS perceives as the physical memory, the guest OS is allowed to maintain the mapping of virtual addresses to what it perceives as its physical memory. But as explained earlier the guest OS is tricked into believing that it is managing a real contiguous physical memory just as the way in normal virtual memory scenario a process is tricked by the OS into believing that it is loaded on and executing from physically contiguous machine memory.
Hence the guest OS's perception of physical memory is actually a pseudo-contiguous-physical memory abstraction provided by the VMM. As per common virtualization terminology it is called *guest physical frames* and a corresponding page number is called *Guest Physical Frame Number* or GPFN. And the corresponding original machine frame number backing a GPFN is termed machine frame number or MFN. Thus the guest OS page page tables contains the *virtual address* $\rightarrow$ *guest physical address* mapping also abbreviated as V $\rightarrow$ P mapping and the VMM should contain a *guest physical address* $\rightarrow$ *machine address* mapping often abbreviated as P $\rightarrow$ M. The P $\rightarrow$ M mapping should be maintained from the time a VM is created and should be updated every time a change occurs to the allocation of machine frames to a VM. This is illustrated in figure 3.1.

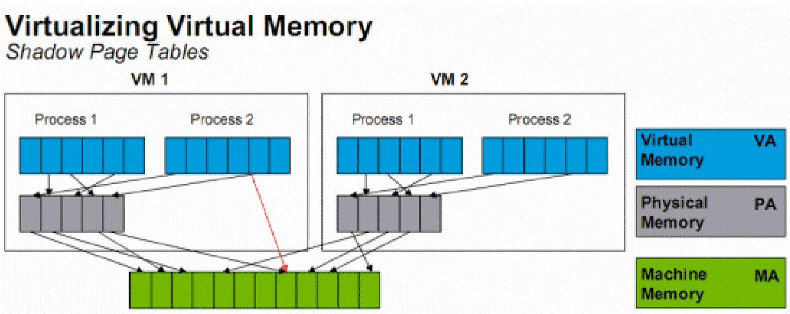

Figure 3.1: Physical memory allocation model in virtualized systems.

`images.anandtech.com/reviews/it/2008/virtualization-nuts-bolts/Shadowpt.gif`

3.2 Techniques of Virtualizing the MMU

This section discusses how the hypervisor defines ways for the guest OS to access the actual machine frames through the MMU functionalities provided by it.
We know that the guest OS maintains page table related information for every process running in a VM and also that the guest OS operates on pseudo physical pages or GPFNs. Memory references by processes running in a VM requires translation of virtual addresses into machine addresses via hardware MMU operations such as page table walk. The page table walk itself needs actual machine frames addresses to get to the different levels of page table as discussed in 2.1. But as discussed in 2.2 the guest OS cannot be given unrestricted direct access to the machine frames holding the page table pages to ensure VM isolation. This means that the guest OS access to the actual MMU (or its functionalities) in the VMM, like CR3 access, changes to page table entries need to be supervised by the VMM.
Table 3.1 lists the existing methodologies of virtualizing MMU.

3.2.1 Shadow Paging

Shadow paging mechanism is a software MMU virtualization mechanism. The access to CR3 is by default protected by design paradigms of virtualization which pushed the guest OS to lower privilege levels than the hypervisor. The machine frames that contain the different page table levels are protected by making them write protected. And for each process in a VM, just like a guest OS maintains a page table, the hypervisor also maintains

Table 3.1: MMU virtualization techniques.

Technique	Description
Shadow Paging	Software approach
Direct Paging	Para-virtualization approach
Nested Page Tables	Hardware assisted approach

a page table called the *Shadow Page Table*. The shadow page table is part of a plan to easily get the direct the V $\rightarrow$ M mapping instead of going through $V \rightarrow P$ and then $P \rightarrow M$ as mentioned in 3.1. Ideally every virtual address that is translated to corresponding guest physical page address in the guest OS should be translated to the corresponding machine frame address via P $\rightarrow$ M in the VMM. This applies to guest physical page addresses that correspond to the pages of the guest OS page table as well. But there is a subtle drawback to this approach. To translate a virtual address we know that the guest OS needs to access the page table whose base address is stored in the CR3. Assuming we have access to the CR3, for the time being, can we go ahead and directly access the guest page table? No, because what the guest OS have is only the guest physical page address of the the page table base. What we need is the address of the machine frame that holds this guest physical page. And in order to access it we need to do a translation using the P $\rightarrow$ M mappings. This needs to be repeated for the base address of every level of the page table. In order to alleviate this overhead we make use of the same P $\rightarrow$ M mappings maintained by the VMM for a VM to create shadow page tables for all process that run in that VM. And in the shadow page tables we maintain the direct V $\rightarrow$ M mappings. Thus every virtual address access by a process in a VM can now go through the corresponding shadow page table in the VMM saving on the number of memory accesses and the corresponding latency. Figure 3.2 illustrates this.

To pull off this trick the guest OS's CR3 is virtualized and guest OS access to the CR3 is to be prevented by a general protection fault (GPF) (Wikipedia, 2015). The VMM should load the CR3 with the base address of the shadow page table rather than with the base of guest OS page table. Virtual memory access that reads or writes already mapped frames can go ahead via the shadow page table unhindered. When a page fault occurs the VMM

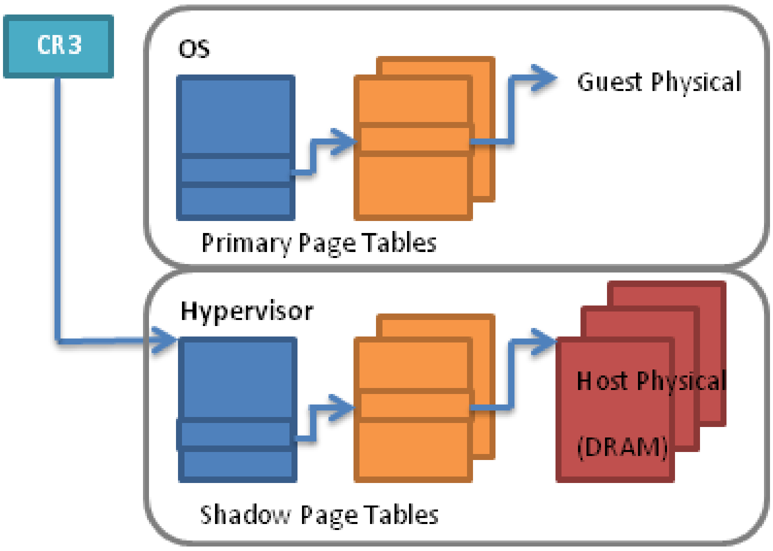

Figure 3.2: Shadow Page Table Illustration.
`corensic.files.wordpress.com/2011/12/shadowpagetables.png`

checks if the guest page table has valid entries for the faluting address, if that is the case then fault occurred because the VMM had moved the machine frame corresponding to the faulting address to its swap space. This type of fault should be hidden from the guest OS as such a fault would not have occurred if the guest OS was running on a native system. The VMM now swaps in the contents of the faulting address into machine memory, updates the P $\rightarrow$ M mapping to reflect this and the updates the corresponding shadow page table. On the other hand if the fault occurred because the guest page table did not have valid mappings for this virtual address, the VMM transfers control to the trap handler of the guest indicating a page fault. The guest OS will then issues I/O requests to effect a page-in operation, which may or may not cause a dirty page swap out. These requests are in fact serviced by the VMM as they are privileged instructions. Once the guest physical page (and the machine frame backing it is) is available the guest OS issues instructions to the modify the guest page tables (Smith and Nair, 2005). This causes an exception called **VMExit** as the frames containing guest page tables are write protected from the guest OS. The VMM now updates the guest page table and the mappings in the shadow page table before returning control back to the guest. This is done to ensure that the shadow page table is in sync with the guest page table.

3.2.2 Direct Paging

Direct Paging uses an approach called *Para-Virtualization* in which the guest OS is made aware of being virtualized. Guest OS is modified as a part of the MMU virtualization. This approach makes use of the **VMCALL** or *hypercall* API provided by para-virtualization. This is a solution to the x86 architecture virtualization issue. More can be read from Force (2000).
The major change that this brings about is in the way in which guest page tables are updated. There is no shadow page table in this approach. Here also the machine frames containing the guest page tables are write protected by making them read only. A change to the guest page table are carried out by a VMCALL from the guets OS to the VMM. Subsequently the VMM VMCALL handler carries out the changes to the corresponding machine frames after validating them to ensure isolation. Hence the guest page tables contains direct *virtual address* → *machine address* mappings that are VMM validated. As it is the case with shadow paging mechanism normal page table reads happen without VMM intervention and page faults are handled by the guest OS by updating the page table entries through the VMM. Figure 3.3 illustrates this approach.

3.2.3 Nested Page Tables

Extensions to the hardware in the form *extended page tables or nested page tables* (EPT or NPT) enabled a new mechanism to achieve memory virtualization called *hardware assisted MMU virtualization*. Contrary to shadow paging, nested paging removes the need for the VMM to supervise the guest OS page table operations once the nested pages are populated. Under nested paging both guest and the hypervisor have their own copy of the CR3 register. The guest OS maps virtual addresses to guest physical addresses and the Nested page tables (NPT) map guest physical addresses to machine frame addresses. The guest now is allowed to set up the guest page tables without any hindrance (i.e VMExits) by the VMM, except for getting a physical frame allocated (for the guest page table or any other guest physical page) which is still under the preview of the VMM. The VMM sets up the nested page table. A guest OS virtual memory reference, when nested page tables are set up, results in a 2-D page table walk - first in the guest page table and next in the nested page table - to get to the corresponding machine frame. Parts of virtual address is used to index into the guest page tables, as explain in 2.1, to obtain the guest physical address and using this guest physical address we do a similar indexing

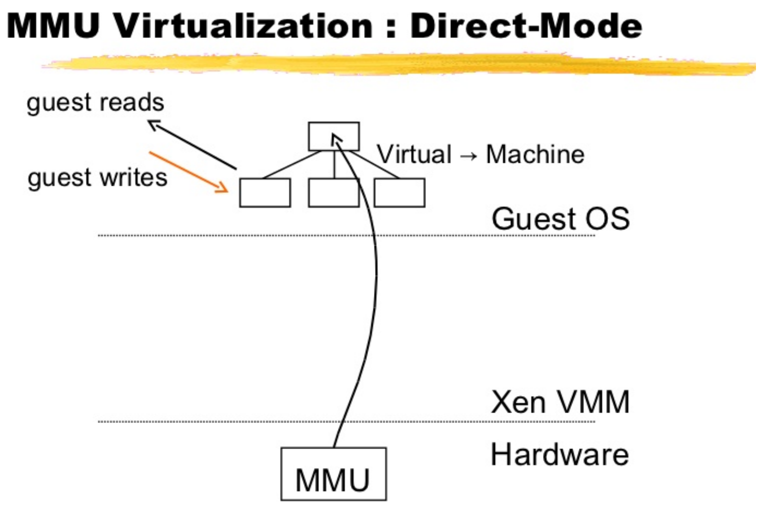

Figure 3.3: Direct Mapping Illustration.

image.slidesharecdn.com/overview-of-xen-30627/95/overview-of-xen-30-12-728.jpg

into the nested page table to obtain the machine frame address. There is a subtle issue here that may miss our eyes easily. While translating a virtual address we need to access machine frames holding the guest page table pages, so for each level of guest page table accessed we need to do a nested page table walk to reach the machine frame holding this page table page.

If the guest page tables contain *n levels* and the nested page table contains *m levels*, then to resolve a virtual address to corresponding machine frame we require

$$((n + 1) \ * \ m) + n$$

memory accessesGandhi *et al.* (2014). For getting the machine frame address of a particular level of guest page table physical page we need to traverse **n** levels of nested page table. Once the machine frame is identified we need 1 memory access for reading it. These are the **n + 1** memory accesses in the previous expression. This is repeated for all levels of guest page table. Hence the **(n+1) * m**. Once you reach the machine frame which holds the last level of guest page table i.e. the PTE machine frame you can read the guest physical address of the original virtual address address and finally to obtain its

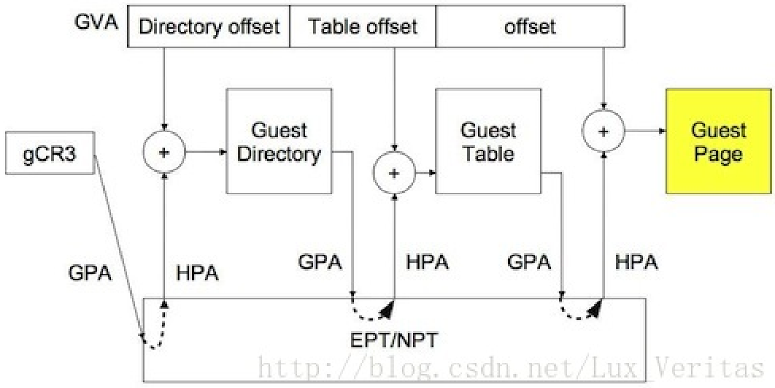

Figure 3.4: Nested Page Table Illustration.
`cnblogs.com/snake-hand/p/3181631.html`

machine address we need to traverse the nested page tables one last time. That is why the **n** term is added to the equation. This is illustrated in figure 3.4.

3.3 Comparison of Memory Virtualization Techniques

Following table compares the different memory virtualization techniques.

Table 3.2: Comparing Memory Virtualization Techniques.

Technique	Advantage	Disadvantage
Shadow Paging	Software approach, no changes to guest OS	Frequent costly VMExits for each guest page table updation.
Direct Paging	No VMExits	Requires changes to the guest OS.
Nested Page Tables	Hardware assisted approach, No VMExits, No changes to guest OS.	More number of memory for a virtual memory translation. About 6 times more.

3.4 Optimizations to Memory Virtualization Techniques

Shadow Paging and Nested Page Tables are two approaches that doesn't require changes to the guest OS. The solution approach they take, though seems similar, are independent of each other. If a process running in the guest OS is experiencing large number of page faults then shadow paging is going to degrade the performance because of frequent VMExits required to synchronize the guest and shadow page tables (refer 3.2.1). Hence shadow paging will prove to be a bad memory virtualization technique for such a process. Now consider another scenario where a process is experiencing large number of TLB misses but its pages are still in memory i.e. it has few page faults, then resolving their machine address by doing a 2-D page table walk in nested paging mechanism is going to degrade its performance due to increased number of memory accesses (refer 3.2.3). For this process nested paging proves to be a bad choice of memory virtualization technique.

3.4.1 Combining Shadow Paging & Nested Page Table

Neither of the two strategies is a clear winner all the time for different mix of processes possible in a VM. Hence in order to get the best of these two techniques Wang *et al.* (2011) proposes a dynamic switching mechanism that switches between the shadow paging and nested paging based on memory access behaviour of the VMs and they call it the *Dynamic Switching Paging* or DSP. Their solution approach is as follows:

- Determine page fault frequency and TLB miss frequency measured as the number of page faults and the number of TLB misses per thousand retired (finished) instructions.

- Determine system dependent upper and lower thresholds for page faults frequency & TLB miss frequency.

- If one metric is beyond its upper threshold & the other is well within within its threshold limits, then switch to the technique/mode that is the most suitable for the metric that is within its limits.

  - "If the TLB miss frequency is higher than the TLB miss upper-bound threshold and the page fault frequency is lower than 80 percent of the page fault upper-bound threshold, switch from" nested page table mode to shadow page table mode or stay in shadow page table mode if already in it.

- "If the page fault frequency is higher than the page fault upper-bound threshold and the TLB miss frequency is lower than 80 percent of the TLB miss upper-bound threshold, switch from" shadow paging mode to nested page table mode or stay in nested page table mode if already in it.

- If both metrics are below their lower threshold limits then remain in the current mode.

- If both metrics are above their upper threshold values or within their threshold limits find out the relative penalty between them.

- Calculate the relative penalty as the P-to-T ratio.

$$P - to - Tratio \ = \ \frac{Page\ Fault\ Frequency}{TLB\ Miss\ Frequency}$$

- P-to-T ratio is used along with a running average of recent P-to-T ratios called historic P- o-T ratios.

- If either of historic TLB miss frequency or current TLB miss frequency is zero then switch from shadow paging to nested page tables without estimating either historic or current P-to-T ratio. This avoids divide by zero errors.

- If both historic and latest P-to-T ratio are above P-to-T ratio threshold upper bound switch from shadow paging to nested page tables or stay in nested page tables if already in it.

- If both historic and latest P-to-T ratio are lower than P-to-T ratio threshold lower bound switch from nested page tables to shadow paging or stay in shadow paging mode if already in it.

- If both historic and latest P-to-T ratio are within the P-to-T ratio threshold's lower & upper bounds stay in current mode.

- If historic and latest P-to-T ratio falls into different threshold intervals, a clear trend cannot be estimated and hence it is better to stay back in current mode.

Through experimentations they claim that DSP is capable of matching the performance of the better among shadow paging and nested page tables, though they do not explain clearly how they attributed the performance degradation of particular processes to the memory virtualization technique used.

3.4.2 Reducing the number of Memory Accesses while using Nested Page Table

The advantages of nested Page table technique make it a good memory virtualization technique. But its drawback of large number of memory access for a single virtual address translation is too much of a cost for processes having poor memory access locality. Gandhi *et al.* (2014) proposes new hardware that makes use of *Direct Segments* both in the guest OS and the hypervisor to reduce the overhead of virtualized address translation. The concept of direct segments is explained in Basu *et al.* (2013). They incorporate the idea presented there to introduce virtualized direct segments that logically reside on the TLB miss path of the guest OS and on the address translation mechanism used in the VMM. Direct segments maps a processes virtual address space with segments and with paging. This enables large parts of contiguous virtual address space to be mapped to contiguous physical addresses with minimal hardware. And for portions that cannot be contiguously mapped direct segments fall back on normal paging. As with normal memory management with *egements* rather than pages, in native systems, which we did not discuss in section 1, direct segments are also managed with 3 registers: BASE, LIMIT and OFFSET. As we know a virtual address V within a segment ($BASE \leq V \leq LIMIT$) gets translated to physical address $V + OFFSET$. The contiguous address space required for the segment is obtained by mapping a large chunk of contiguous virtual addresses with the same permissions, which is called the *primary region* abstraction in Basu *et al.* (2013). The different modes of operation that they propose are the following:

- *Dual Direct* mode that uses direct segments both in the guest OS and VMM. It uses the direct segment registers both in the guest OS and VMM to translate directly from *guest virtual address* → *machine address*, thus bypassing both dimensions of virtualized address translation.

- *VMM Direct Mode* uses direct segments for the 2nd dimension of virtualized address translation i.e. *guest physical address* → *machine address*. Hence the guest

OS can be left unmodified. Only the VMM needs modifications to support this mode.

- *Guest Direct* mode limits the support of direct segments to the guest OS. *guest virtual address* → *guest physical address* translations now has zero overhead. VMM uses normal paging techniques to manage the actual machine memory.

In effect they flatten one or two dimensions of the page walk based on the mode of operation. They also address the possibility of lack of contiguous guest physical memory needed to create a direct segment due to fragmentation using *self ballooning*. Self ballooning uses *ballooning* to remove the fragmented guest physical memory form use and *hotplug* it back as contiguous guest physical memory that can be used to create guest direct segments. They also talk about plugging gaps caused by faulty pages in the contiguous direct segments by using *escape filters*. Escape filters are nothing but making use of conventional paging to map holes in the physical address that break the addresses' physical contiguity. Hence while using direct segments and translating addresses, along with the segment register we need too check the hardware that implements the filter to *escape* the faulty page from segment based translation to paging.

Chapter 4

Managing Virtualized Memory

As briefly mentioned in sections 1 and 2.2 the major task at hand once we are done with virtualizing memory is managing memory among the VMs to meet their memory demands. Improper memory allocation/management can adversely affect applications running inside the VM and can even lead to their termination due to memory shortage. It is clear beyond doubts that a *dynamic* management scheme gives better utilization of memory. Dynamic management schemes enable memory over-commitment - leading to increased consolidation of VMs within a single physical machine. There are different aspects that are to be carefully considered when implementing a dynamic memory management scheme in native as well as virtualized environments and these can well be classified as the challenges of memory management. We limit ourselves to the aspects (and/or challenges) of memory management strategies involved in the virtualized environs within the scope of this seminar and they can broadly be listed as:

- Accurately determining memory usage/requirement statistics of individual VMs.
- Describing policies/strategies that go along with guest OS memory management policies.
- Evolving policies that can embrace the varied guest OS policies and memory usage patterns of different applications.
- Finally the actual mechanisms, to carry out the policy decisions, that result in effective memory (re)allocation among VMs.

4.1 Techniques of Memory (Re)allocation

The main techniques to adjust VM memory allocation are the following:

- Demand Paging
- Ballooning
- Content based Page Sharing

4.1.1 Demand Paging

An obvious choice for managing machine memory among VM would be to employ demand paging at the VMM. The VMM needs to swap pages of VMs to and from the disk. The pages of a VM can be reclaimed in this way and assigned to another who is in need of memory i.e. the contents of the victim VM's pages will be swapped to disk and these pages will now be allocated to the target VM. This approach has the following problems.

- The VMM can select pages that are frequently being used in the VM. This can lead to frequent swapping in and out of pages.
- The VMM taking its own independent decisions while being in the dark about guest OS management decisions can give rise to problems like *double paging* (Waldspurger, 2002). Here the VMM swaps out a guest OS page and then the guest OS decides to swap out the same page ignorant of the fact that the page is already swapped out. This will force the VMM to swap in the page just to be swapped out immediately by the guest.
- The VMM, of course, needs to be provisioned with a demand paging system which will make the VMM design more complex.

4.1.2 Ballooning

Ballooning bridges the gap that exists in VMM demand paging by letting the guest OS take an informed choice about which pages to be given up for reclamation by the VMM. This is achieved with the help of a *balloon driver* module loaded inside the VM. On VMM request the driver "inflates" the balloon i.e. it acquires guest physical memory from the guest OS kernel allocator, similar to the way in which a device driver would allocate

pinned DMA buffers (Salomie *et al.*, 2013). The associated GPFNs are now considered by the guest OS to be private to the balloon driver. The GPFNs are now passed on to the VMM by the balloon driver and the VMM can use the machine frames backing these guest physical pages safely for other purposes. Here the guest OS is completely in control of what pages to be given up for the VMM. If it perceives that it has enough free guest physical pages it can allocate guest physical pages from this free pool. The best part of this strategy is evident when the guest OS doesn't have enough free guest physical memory; in this case it is the guest OS that selects which guest physical pages are to be swapped out to disk. This is much better than letting the VMM transparently swap out random guest physical pages. To return pages back to guest OS the balloon driver "deflates" the balloon. This is done by first ensuring that the pinned guest physical pages of the balloon driver are backed by machine frames and then by unpinning and freeing them to the guest OS kernel.
(Waldspurger, 2002)

4.1.3 Content based Page Sharing

The techniques that we saw so far were purely reactive ones that was driven by demand. There can be other approaches, ones that can be used more pro-actively, to increase the available free memory and thereby use this increased memory to alleviate memory pressure of VMs. *Content based page sharing* is such an approach which looks for opportunities for sharing pages, used by different VMs, that have similar content.
Multiple VMs using memory pages with the same content is equivalent to having multiple copies of the same page. Memory that is wasted by having these duplicates can be reclaimed if a single copy this physical page was shared by all the VMs. But there a few things that need to addressed while enabling page sharing.

- Changes to guest physical → machine address translation are needed.
- Any change to a shared page by a VM should be resolved by allocating a separate private copy of that page to the VM. To achieve this all shared pages are made read-only and marked as *copy-on-write or CoW*.
- Deciding how to search for duplicate pages.
  Waldspurger (2002) suggests an approach where a scanning process in the VMM periodically scans through the machine frames for content similarity. They also

discuss hashing the contents of a frame and making a *Hint Frame* for the page to assist in faster lookup and to avoid marking every page as CoW. Scanning for similar pages is an overhead that needs to be controlled. In addtion Mishra and Kulkarni (2015) explains other scanning mechanisms and optimizations to scanning mechanisms. Few optimizations to scanning mechanism explained there are like giving priority to memory pages used for disk access after the initial whole system scan, using the nested page tables to track and explore sharing opportunities among recently modified pages, finding duplicates in the disk access paths.

4.2 Determining Memory Usage Statistics

The effectiveness of a dynamic memory management technique depends on how accurately it can determine the memory usage/requirement statistics of a VM. There is an option of obtaining the memory usage statistics from what is maintained in the guest OS. This may not be an accurate determination as various guest OSs may be using different metrics for their individual estimations. *Working set size* or WSS is often taken as a measure of memory usage. WSS is the amount of physical memory that is being actively used by an application out of its total allocation in a recently concluded epoch of time. Hence WSS estimation at, well thought out, regular intervals gives us an idea about how much memory a VM needs or can give up. If the WSS is smaller than the actual memory assignment to the VM, it can can give up some of the unused memory without affecting it performance. There are different methods of estimating this WSS. An easy and obvious way of determining the WSS is to remove all *guest physical* → *machine frame* mappings for all the VM physical pages by manipulating the page present/protection bits. Subsequent access to these pages will result in a minor fault and this can be used to increment an access counter for each of these pages. This is rather CPU intensive approach. Discussed below are offshoots of this approach.

4.2.1 Sampling based WSS estimation

Waldspurger (2002) discusses a statistical sampling approach to obtain the WSS of each VM as a whole. According to this approach each VM is sampled independently and during each of the sampling intervals a small constant number of physical pages are sampled by invalidating all mappings associated with its GPFN. This leads to a trap on accessing these guest physical pages during the sampling period, and as a result the access count of those physical page for that interval is incremented. At the end of the sampling period, the fraction of memory that is actively accesed is given by

$$Access\ Count\ /\ Number\ of\ pages\ being\ sampled$$

. A sort of exponential weighted moving average is used to smooth out estimates across sampling periods. They then extrapolate the usage statistics to obtain the statistics for the entire system.

4.2.2 Page Miss Ratio Curve based approach

The sampling based WSS estimation can estimate the memory requirement of a VM only when there is free memory and cannot estimate the memory requirements of the VM if it is using up its entire allocation. To over come this Zhao *et al.* (2009) uses a *Page Miss Ratio Curve* or MRC based WSS estmation.

$$Miss\ Ratio\ =\ Page\ Miss\ Rate\ /\ Memory\ Size$$

gives us the page miss rate for a given allocated memory size. This is a more accurate measure of the memory utilization for a given allocation rather than the sampling based approach. MRC is obtained from LRU histogram and the basic idea behind it is the Mattson's Stack Algorithm explained in Zhou *et al.* (2004). This algorithm gives us the page miss rates possible, for different amounts of memory allocation, for the recent memory access patterns. Thus by looking at this information the VMM can deduce the memory allocation to be made to a VM, to achieve a tolerable page miss rate. Moreover Zhao *et al.* (2009) also looks at the swap space usage to find out memory requirements beyond the current allocation which many estimation techniques fail to capture. To avoid the overhead of trapping all memory accesses, pages are classified as Hot and Cold pages. Initially all pages start out as cold i.e. their user access privilege is revoked thereby causing an access to them to be trapped into the VMM as minor page faults. Once accessed they become hot and are moved to a hot page queue. As long as they are in this queue accesses to them are not trapped into the VMM i.e. their user access privileges are restored. Thus the WSS can never be smaller than the set of hot pages in the hot page queue. Once the queue is full the page at the head is moved back to the list of cold pages. Hence the page miss ratio is tracked with respect to cold pages alone.

4.3 Virtualized Memory Management for Applications that manage their own Memory

Applications that manage their own memory depend on their knowledge of available memory to work efficiently. In virtualized systems using conventional ballooning, it is the guest OS who makes the page replacement choices to satisfy the balloon's demand. But this can adversely affect applications that manage their own memory as these applications are configured with fixed quantity of memory and they optimize their use of this memory to maximize performance. But these application can always benefit from additional memory, if available. Salomie *et al.* (2013) looks at a memory management strategy that makes use of a ballooning mechanism, which preserves the applications ability to optimize performance based on having an accurate idea of available memory, to vary the applications memory allocation and to enhance collocation of VMs by efficiently consolidating available memory. They implemented *Application Level Ballooning* or ALB for MySQL and OpenJDK. These applications are configured with a maximum memory pool size. It is from this limit that ALB tries to balloon out non-utilized memory with the application's knowledge. And ALB gives back to this limit when the application faces a memory crunch.

The ALB module for MySQL is built into the InnoDB storage back-end. In JVM the most of the application's memory is managed in a heap controlled by the Garbage Collector (GC). Hence they do ballooning in and out of this heap space. The state of the pages, to be given to the balloon module, obtained from the application and from the kernel is different. Hence the application pages identified to be given up for the balloon driver should be converted to a form equivalent to a page obtained from the kernel and thus made usable by the balloon driver. Following describes the basic working of:

- A management and monitoring system determines policy and conveys a target balloon size to the ALB module of the ALB aware application via an RPC call to a local socket.
- The application allocates pages.
- Notifies the OS via a system call.
- The modified guest OS balloon driver processes the requests

- Makes the memory changes visible to Xen with a hypercall.

On similar note Hwang *et al.* (2013) proposes a modified *memcached* and a prototype *disk prefetching system* that makes of a VMM pooled memory, obtained by pooling together spare memory of different VMs within the PM and other PMs, as a volatile data cache. Memcached is also an application that works by allocating fixed size cache on each server in a data center cluster. But memcached is not exactly adept at utilizing the spare memory available. Hence mortar exposes unallocated system memory via the VMM & it is the VMM that controls access to this volatile storage area.

Chapter 5

Conclusion

Various aspects of virtualization of memory as a resource and multiplexing of virtualized memory among VMs were touched upon through this work. The techniques of virtualizing memory like Direct Paging, Shadow Paging & Nested Page Tables have been around for quite some time now and are like de facto standards of doing the same. It is the virtualized memory management angle that is still evolving to churn out new proposed strategies that suit niche needs and applications. There are a lot of strategies proposed in this regard and we touched upon a few noticed ones. We also tried to get a good grip on the challenges to managing virtualized memory. With the proliferation of cloud data centers the focus is going to be on efficient utilization of available resources to bring down the space, time and energy costs. Memory is one of the most important resources next only to the CPU, maybe, in terms of cost as well as utility. But as we saw, effective management and reaping immediate benefits of this management is not as easy as it is with the CPU. Hence better optimizations and newer strategies are bound to come in.

References

Basu, Arkaprava, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift, 2013, “Efficient virtual memory for big memory servers,” in *ACM SIGARCH Computer Architecture News*, Vol. 41 (ACM) pp. 237–248

Force, US Air, 2000, “Analysis of the intel pentiums ability to support a secure virtual machine monitor,”

Gandhi, Jayneel, Arkaprava Basu, Mark D Hill, and Michael M Swift, 2014, “Efficient memory virtualization: Reducing dimensionality of nested page walks,” in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on* (IEEE) pp. 178–189

Hwang, Jinho, Ahsen J Uppal, Timothy Wood, and H Howie Huang, 2013, “Mortar: filling the gaps in data center memory,” in *Proceedings of the 4th annual Symposium on Cloud Computing* (ACM) p. 30

Mishra, Debadatta, and Purushottam Kulkarni, 2015, “A survey of memory management techniques in virtualized systems,” Unpublished paper

Popek, Gerald J, and Robert P Goldberg, 1974, “Formal requirements for virtualizable third generation architectures,” Communications of the ACM **17**, 412–421

Salomie, Tudor-Ioan, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone, 2013, “Application level ballooning for efficient server consolidation,” in *Proceedings of the 8th ACM European Conference on Computer Systems* (ACM) pp. 337–350

Smith, Jim, and Ravi Nair, 2005, *Virtual machines: versatile platforms for systems and processes* (Elsevier)

Waldspurger, Carl A, 2002, "Memory resource management in vmware esx server," ACM SIGOPS Operating Systems Review **36**, 181–194

Wang, Xiaolin, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li, 2011, "Selective hardware/software memory virtualization," ACM SIGPLAN Notices **46**, 217–226

Wikipedia, 2015, "General protection fault — wikipedia, the free encyclopedia," [Online; accessed 21-April-2015], `http://en.wikipedia.org/w/index.php?title=General_protection_fault`

Zhao, Weiming, Zhenlin Wang, and Yingwei Luo, 2009, "Dynamic memory balancing for virtual machines," ACM SIGOPS Operating Systems Review **43**, 37–47

Zhou, Pin, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar, 2004, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, Vol. 38 (ACM) pp. 177–188