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

As mentioned in the introduction section the arrival of a new layer of software abstraction above the hardware relegated the native OS to the role of a guest OS. Especially on architectures like the x86 this posed a serious challenge. x86 architecture has this notion of different CPU privilege levels to provide protection. They are represented as *protection*

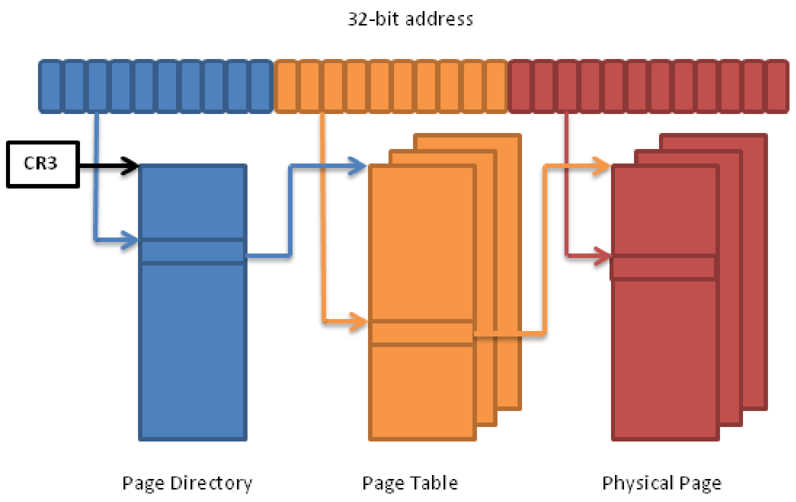

Figure 2.1: Illustration of *VA* → *PA* address translation.
Source: `corensic.files.wordpress.com/2011/11/virtualmemory.png`

rings. There are 4 rings and each ring correspond to a privilege level, 0 being the highest privilege level and 3 the lowest. The OS in a normal setting runs on *Ring 0* and user applications on *Ring 3*. *Rings 1 & 2* are not used in general but was originally intended by Intel™ for housing device drivers. They have more permissions than *Ring 3* but cannot access privileged instructions and any attempt to access such an instruction results in a general protection fault *(GFP)*(Wikipedia, 2015). This gives the OS complete control over all resources. But the introduction of a *Hypervisor* made it necessary to host it on *Ring 0* and this pushed the guest OS to *Ring 1*.

Rings 1 and 2 are in a way, ”mostly” privileged. They can access supervisor pages, but if they attempt to use a privileged instruction, they still GPF like ring 3 would. So it is not a bad place for drivers as Intel planned...

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 (**?**).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.

References

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

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

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`