Virtualization: Memory Virtualization

1 Introduction

Hello, and welcome to this lesson on Memory Virtualization.

When we explored virtual memory at the beginning of the course, we encountered the problem that processes expect each their own zero-based virtual address space, which need to be multiplexed on the available physical memory.

When we run multiple guest OSes on the same machine, we encounter the same problem again, but this time we have each guest OS expecting their own zero-based physical address space. This lesson will be about how to multiplex multiple physical address spaces on the main memory of the machine.

We will first explore the mechanics of how this can be done on available memory management software, by using so-called shadow pages.

We will then re-visit some of the virtual-memory management problems that we encountered earlier, such as page replacement algorithms, but this time applied to managing the physical memory of guest OSes on the available main memory of the machine. We will see that bad page replacement decisions can be very painful.

Virtualization is often used in situations where memory is severely over-committed, that is, where the sum of the memory allocated to the virtual machines exceeds the available memory on the machine significantly. We will look at a particularly clever approach, called "ballooning", which enables the hypervisor to control the memory footprint of the virtual machine while leaving the page replacement and other memory management decisions with the guest OS.

At the end of this lesson you will be able to describe how memory can be virtualized and some of the problems that one encounters in doing so. The discussion of memory virtualization will also give us the opportunity to explore some truly clever approaches to translate and manage what one may call "virtualized virtual memory".

Let's begin.

2 Virtual vs. Physical vs. Machine Memory

We remember from a very early lesson on memory that . processes expect a zero-based address space, where they have their executable, their stack, and so on. We call this the logical or virtual address space of the process.

From a system perspective, this address space is a collection of virtual pages, which are managed by a page table.

When we run this process in a system, the virtual pages are mapped onto a collection of physical pages, which we call "frames".

Since the system runs multiple processes, virtual pages from multiple processes have to be multiplexed on the physical pages.

In a virtualized environment, the physical memory is not the main memory of the machine, but rather the memory of the virtual machine.

For lack of a better term, the main memory of the machine is called "machine memory", and the frames of the main memory are called "machine pages". And because we want to run multiple virtual machines on the same host, the physical memory of the multiple guest OSes must be multiplexed on the available machine pages.

In the same way as processes expect zero-based virtual address spaces, guest OSes expect zero-based physical address spaces.

3 Shadow Page Tables

The problem now is how to efficiently translate the virtual addresses of the currently running process in a virtual machine to the the machine address of the host.

Let's focus on the a single virtual machine. And let's represent each process by its page table.

In a physical machine, the OS would switch page tables by loading the page table register, which is CR3 in the case of the x86. Since the virtual machine is not allowed to manipulate the real CR3 register, it manages a "virtual CR3" register instead. This virtual register points to the page table of the current process.

The hypervisor manages, for each process, a so-called shadow page table, which is the actual page table used by the memory management unit. In fact, the "real CR3" register points to the shadow page table of the running process.

While the page tables map the virtual address to the physical address, the shadow page table maps the virtual address to the machine address.

3.1 Discover the Page Table

How can the hypervisor keep track of the guest operating systems manipulations of the page tables? Let's start with the basic problem of discovering how many and what page tables are managed by the guest OS. The hypervisor needs to be able to discover the guest OS's page tables without any form of API to do this.

Let's assume that the guest OS just created a new page table. Until this page table is actually used, the hypervisor does not need to know about it. In order for the page table to be used, the guest OS has to load the page table into CR3.

This is a sensitive instruction, which is intercepted, either though an exception, or by code that was generated by binary translation. Instead of simply loading CR3, the hypervisor now executes a function, say "handle_CR3".

The hypervisor now knows the address of the new page table, and can therefore traverse it and generate the shadow page table.

Once the shadow page table has been constructed with all the mappings from virtual to machine addresses.

The virtual CR3 register can be loaded, and the real CR3 register as well. The memory management unit now uses the second shadow page table to translate virtual addresses to addresses in the main memory of the machine.

3.2 Switch the Page Table

If the guest OS later executes a process-level context switch, it has to load the page table of the new process into CR3. This again invokes the handle_CR3 function of the hypervisor, which loads

the shadow page table of the new process and the function returns.

3.3 Page Fault

How do shadow page tables handle page faults? Recall that the memory management unit uses the shadow page table and not the page table of the guest OS.

The process, refers to a virtual address with an invalid entry in the shadow page table.

A page fault exception occurs, and the page fault handler of the hypervisor takes over. The handler checks the page table to see if the page table entry there is valid. If it is not, this means that this is a bona fide page fault, and the hypervisor direct the page fault to the guest OS for handling. . The guest OS handles the page fault, and not the entry in the guest OS's page table for the current process is valid. . The page fault handler returns.

3.4 "Shadow Page Fault"

Sometimes it can happen that the guest OS updates page table entries without the hypervisor knowing it. This is possible because manipulations of the page table are memory operations, and cannot be detected as sensitive by the hypervisor. This is not a problem, because it does not really matter what is in the guest OS page table until the entry is referenced.

If it is, like here, where the change to the page table was not propagated to the shadow page table, another page fault exception occurs. and the page fault handler of the hypervisor takes over. Now the entry in the guest OS page table is valid. The Hypervisor therefore knows that this is a so-called "shadow page fault", where the entry in the shadow page table has be brought up to speed with the entry in the guest OS page table. . This is done, and the page fault returns. The address can now be referenced.

3.5 Shadow Page Tables: Conclusion

Shadow page tables are great because they eliminate the need for double book-keeping of virtual to physical mapping and then physical to machine. The fact that this can be done with legacy page table hardware, including legacy TLB's is impressive.

The problem with shadow page tables is two-fold.

First, every page fault, whether it is a bona-fide page fault or a shadow page fault, has to context switch into the hypervisor and then back. This is very expensive.

Second, every page table has to be duplicated, which increases the memory footprint significantly.

In the last few years, support has been added to hardware for memory virtualization. For example, both AMD and Intel have added some form of nested paging or extended paging, which all is based on having much larger, and virtual-machine-aware, TLBs support the mapping from virtual addresses to machine addresses. This allows to eliminate shadow page tables, and so significantly increases performance.

4 Issues with Page Replacement

Just as in virtual memory in traditional systems, we have to address replacement policies after one has solved the page fault mechanics. One key issue is page replacement.

Replacement policies are particularly interesting here because virtualization is often used in settings where resources have to be over-committed for the system to run profitably. The replacement

problem for the hypervisor is different than for the guest OS. If the hypervisor runs out of machine memory, is has to select victims among the physical pages that it has allocated to virtual machines. It then has to page out those physical pages to disk.

There are two issues. The first is that the hypervisor's page replacement algorithm has to pick a victim virtual machine and a physical page in that machine.

Picking a machine is easy, as the hypervisor may collect information about how utilized the memory of the various virtual machines is.

Picking a victim physical page on the other hand, is difficult, and only the guest OS knows what a good page to replace is.

If the hypervisor picks the wrong page, the effects can be serious.

Say the hypervisor picks a physical page as a victim. The page is evicted from memory to disk. Some time later, the guest OS may want to page out the virtual page stored in the physical page that we just evicted. To do so, the hypervisor has to first page back in the physical page for the guest OS to page it back out.

This causes two page faults per single fault. This is why this effect is called "double-paging".

5 Ballooning: Avoiding page-out "Physical" Pages

If physical pages have to be moved to disk, it is best to have the guest OS choose which ones. The problem is that there is no API for the hypervisor to communicate to the guest OS to shed physical memory. A clever approach that enables the hypervisor to trigger the guest OS to page out memory is "ballooning". The idea is as follows.

Let's have an indicator for the amount of memory pressure that the hypervisor experiences.

As we start virtual machines, each with a given amount of physical memory, the memory pressure for the hypervisor increases.

The pressure increases further as we start more virtual machines.

In order to reduce the amount of memory used by the virtual machines, the idea is to deploy special device drivers that control memory balloons. Balloons just use request physical memory from the guest OS and so make it inaccessible to the virtual machine. This memory is then made available to the hypervisor to allocate to other virtual machines.

If the hypervisor experiences memory pressure because it is running many virtual machines, it may inflate one ore more balloons. When the balloon in a virtual machine inflates, it requests more memory, thus causing memory pressure for the guest OS.

In response to the increased memory pressure in the virtual machine, the virtual memory manager of the guest OS kicks in and starts paging out pages, either directly, or by swapping out processes. As it does so, the memory pressure in the virtual machine is released, and the hypervisor memory pressure as well, until it reaches the target level.

Interestingly, the decision of which pages to page out is done in an informed fashion by the guest OS, and not by the hypervisor.

Ballooning is a wonderful idea, but it is not without problems.

There is a saying that ballooning works great as long as it works.

Ballooning drivers are at the mercy of the guest OS, however. If the guest decides to uninstall, or disable, the driver, ballooning becomes stops working.

Similarly, guest OS may carefully cap the amount of memory reserved by drivers, for example, to protect against buggy or malicious drivers. In such cases the effectiveness of ballooning is hampered as well.

6 Summary

We have come to the conclusion of this lesson on memory virtualization.

We learned about the for an additional layer of memory, which is often called "machine memory".

We also learned how to multiplex multiple physical address spaces onto a single machine address space, with the help of legacy paging hardware. The solution is to use shadow page tables that map virtual addresses to machine addresses rather than to physical addresses. This shadow page tables are hidden from the guest OS, and some interesting gymnastics has to be done to keep the guest OS's page tables synchronized with their respective shadow page tables. Shadow page tables are being rendered obsolete in real systems do the emergence of sophisticated memory management hardware that supports virtualized memory directly. Examples are AMD's nested page tables and Intel's extend page tables.

In addition to page fault mechanics we need to also deal with page replacement policies, just as in the case of virtual memory. We learned about the danger of double page faults.

A solution is to delegate the page replacement to the guest OS and control amount of memory that is available to the virtual machine. One way to do this is the use of ballooning device drivers.

We truly hope that you enjoyed this lesson on memory virtualization. This topic is a very dynamic one, as developments in OS design are complemented by dramatically increased hardware support. It is an exciting topic for further study.

Thank you for watching.