Memory Ballooning

In case of a cloud offering, the resources are best utilized by overcommitment. The case for overcommitment is that not every user/VM will be using all of the resources allocated to it at the same time. So, by allocating more resources to the VMs than are available, cloud can support more number of VMs.

The Problem with simple memory overcommitment

When a Virtual Machine starts, it is allocated physical memory page frames from the host machine by the hypervisor on demand, similar to the processes running on an OS. But unlike processes, once a memory page is no longer needed by the guest OS, it does not return it back to the hypervisor. So, once memory is given to a guest OS, it cannot be reclaimed by the hypervisor. Why does this happen?

When an Operating System is running in the virtualized environment, it must get the illusion that all the memory starting from physical address 0 belongs to it. Multiple Operating Systems should be able to run on the same physical memory. To do this, hypervisor must add an extra layer of virtualization to virtualize physical memory for each VM on top of the machine memory. Hypervisor maintains a mapping from the physical pages of a virtual machine to the actual machine memory pages, similar to the page tables of a process. The guest OS internally keeps per process page tables. So, it takes two translations for the virtual address of a process running on guest OS to be translated to actual physical address of the machine memory.

When a Virtual Machine starts, none of its physical memory pages are mapped to the actual machine pages. These are allocated on demand as and when the hypervisor intercepts all the page faults and TLB misses of the guest OS. But there is no way for the hypervisor to know when a physical page has been freed by the guest OS. There is no interrupt/trap instruction called by the guest OS which can be trapped by the hypervisor when a page is no longer needed by the guest OS and is freed.

What happens on a memory access by

a VM?

Since the code of the guest OS is directly running on the CPU, for each address, it will refer to the TLB for its translation. On a TLB miss, the flow will be different for hardware managed MMU and software managed MMU. Since x86 has hardware mananged MMU, let us discuss the case of hardware managed MMU. In normal case (without virtualization), the CPU gets the address of the page table of the current process from the CR3 register and walks the page-table to get the physical address. In case of virtualization, hypervisor has to maintain a shadow page table for each process of each of the virtual machine. The CR3 register points to this shadow page table of the process of the VM which is running. This shadow page table contains mapping from the virtual address of the VM directly to the machine page address. How are the shadow page tables built and maintained?

When a process needs memory, the guest OS kernel creates a mapping from the process virtual address to the guest physical address. Now when the process in the guest OS tries to access it, there is TLB miss. the TLB traverses the shadow page table and since the page is not present (I think the `present` bits of all the pages in the shadow page tables are set to 0 in starting), this generates a page fault. The page fault is trapped by the hypervisor(why not by the host OS? Or is it trapped by the host OS which passes it on to the hypervisor? If so, how?). The hypervisor checks the page tables of the guest OS. How does the hypervisor know the address of the guest OS page table? Accessing the CR3 register causes VMExit(hardware supported virtualization) or an interrupt(when there is no harware support for virtualization) and hence the hypervisor gets the address of guest OS's page table. The hypervisor finds a new machine page for that physical address and updates the shadow page table. Since this page fault handler was generated due hypervisor not having the correct entry, it should not be passed on to the guest OS. This whole process should be transparent to the guest OS.

What happens in case of a genuine page fault? Normal page faults can occur when `mmap` has been used to map a file to memory, or the OS has swapped out a page from the memory (Think of more such factors due to which page faults can occur). In that case, the hypervisor passes the page fault interrupt to the guest OS. Then the guest OS first finds a free physical page for it, reads the page from disk(this generates other interrupts which are trapped by the hypervisor and handled accordingly), the read data is copied to the free physical page (this will generate a TLB miss and a page fault if that physical page has no backing machine page) and the page table inside the guest OS is updated.

What happens when the guest OS tries to swap out a page? Hypervisor should know about this and set the present bit of the corresponding shadow page table entry to 0(how does this happen?), which would trigger a TLB miss, then page fault on the next access, which would be passed on to the guest OS by the hypervisor.

What happens when the host OS swaps out a machine page which was being used by a guest OS? Hypervisor runs along with the kernel and knows about this, and thus sets present bit to 0 in its mapping. Next TLB miss, will cause a page fault, which should not be passed on to the guest OS by the hypervisor. **How does the hypervisor know when to pass on the page fault to the guest OS and when not to?**

Now, the hardware supports virtualization and there is no need for shadow page tables, the MMU automatically walks both - the hypervisor mapping and the guest OS page table to get the correct machine memory address. Maintaining the mapping from physical memory to machine memory will go through similar steps as described above.

What can be the ways around this?

- The hypervisor periodically goes through the free list of the guest OS and removes those pages from its mapping. But how will the hypervisor know the address where the free pages list of the guest OS is stored? When when will this routine maintenance task take place?
- Guest OS calls an interrupt/trap instruction when it gets a free page. Will require modification of the guest code. There are better ways to do this by modifying the guest OS code.

Ballooning

The way memory reclamation is implemented in is through a balloon driver which resides in the guest OS. Hypervisor can talk to the balloon driver in the same way as a device talks to its driver in the Operating System (hypervisor emulates all the devices of the guest OS). On balloon inflation, this driver gives a list of free pages of the guest OS to the hypervisor and hypervisor can remove those pages from its mapping. These pages can now be used by any other VM on the same host or any other process on the host machine. This requires the balloon driver to be present in the host OS. Linux already has the virtio balloon driver in its source.

When the pages are taken away from the guest OS, its physical memory(total capacity of the guest OS) decreases. On balloon deflation, this driver return some memory pages back to the guest OS. The toal capacity of the guest OS increases.

##