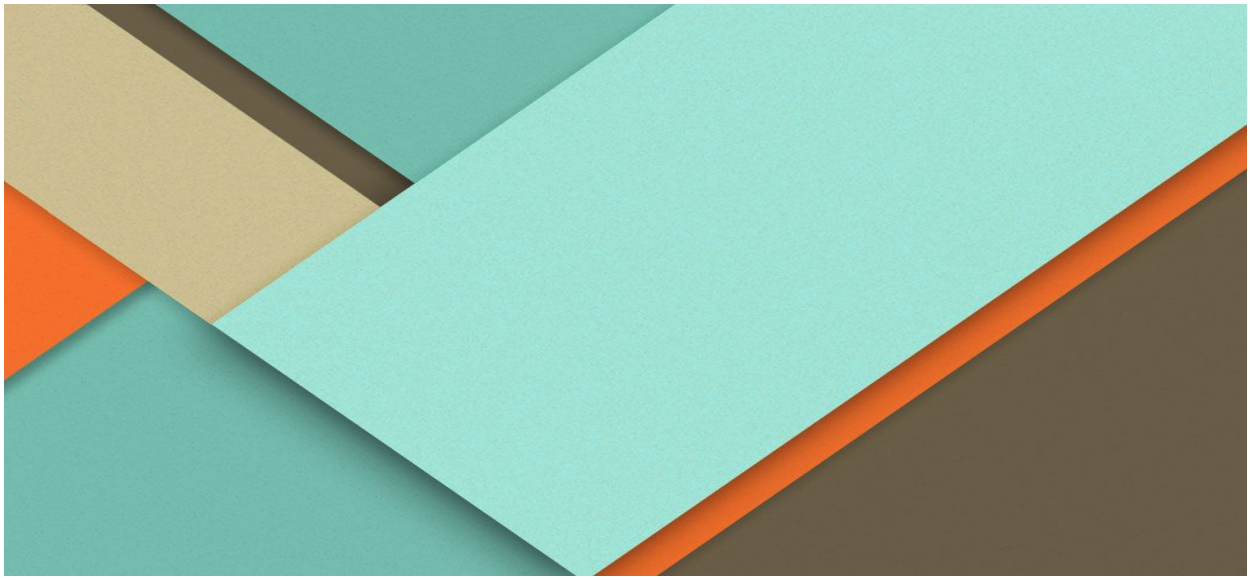

Cloud Computing

Design of Guest OS & Hypervisor

02.10.2017

Submitted by: [GROUP 15]

Aditi (2014CS10205)
Aditi Singla (2014CS50277)
Aditi Gupta (2017BSY7508)

Current Virtualisation Technology

1. Modern Hypervisors employ trap & emulate model. When a Guest OS executes a privileged instruction, the hypervisor traps it and takes over the control. It generates instructions to handle the trapped instruction and in a way emulates it. Here, the non-privileged instructions are not trapped and are executed as they should be.
2. **Problems with this technology :**
   a. This turns out to have a huge overhead, since we need to context-switch from Guest OS to hypervisor, which is expensive to do at each privileged instruction.
   b. The Current Privilege level of Guest OS can be seen in the hardware registers. So, this CPL cannot be kept hidden from the Guest OS, which can somehow detect its CPL value.
   c. There is a lot of book-keeping required to maintain 2 level page tables and so on, for managing memory and other resources.

Modifications

Since we are now designing the Guest Os and hypervisors from the scratch, we are suggesting a few modifications in all the three components of virtualization. The main idea is to make the guest OS like a process running on host.

1. Memory Virtualization :

   Issues -

   Since the guest OS is not aware of virtualisation, it works in its own address space, which then needs to be translated to the hypervisor address space, and that needs to be further mapped to the actual physical space. This requires a 2 level page table,and a Translation Lookaside Buffer (TLB) to cache the mappings. Now, this also means that the TLB would need to be flushed every time a context switch occurs.

   Solutions -

   - Like in allocating memory to processes (upper 2GB for OS, and remaining 2GB for process code & data), we can divide the total memory into some part for hypervisor, and allocate the remaining as contiguous blocks to the guest OS. This would greatly simplify book-keeping for both hypervisor and guest OS, since now we don't need a 2- level page table now. **Direct mappings** can be established to convert guest OS' physical address to host physical address.

- The hypervisor can use **large pages** to allocate guest OS. This is because small pages made sense while allocating to processes, but since it has to allocate guest OS' memory, giving them large pages will greatly reduce the required book-keeping and the load on the TLB (Translation Lookaside Buffer).

- Various other OS optimizations can also be employed by the hypervisor. Like, it can implement **copy - on - write**, i.e. copy contents of page only when something is written on the new copy. Also, **demand paging** can be used.

2. I/O Virtualization :

   Issues -

   In the current architectures, there are three types of I/O virtualization based on where does the hypervisor take control of the call - System Call, Device driver, IO operation. The issues with each of these are as follows:

   1. System Call Interface - The entire I/O action would need to be emulated by the hypervisor, which also means that it needs all the ABI routines to shadow the ones available to the users.
   2. Device driver Interface - The VMM can intercept the call at driver level, it will convert the virtual device info to the physical device and make a call to the driver program. This means that the VMM needs to be aware of the internals of each of the guest OS to do so.
   3. I/O Operation Interface - A single device call can actually involve multiple operations at I/O level, and hence it is really hard for the hypervisor to reverse engineer the complete I/O action.

   To avoid such complications, we propose a generic virtual device interface between the hypervisor and the guest OS.

   Solutions -

   The hypervisor can implement a uniform **virtual device interface**, where multiple virtual devices are mapped to each physical device. Now, for each physical device, each Guest OS gets a Virtual Device. The virtual memory addresses and the registers will be the same as the physical addresses.

   This means that the hypervisor will not need to trap each I/O call made by the guest OS's. It then just needs to have a **queuing system** at the interface, to execute the I/O operations from virtual devices at the corresponding physical device. The best ordering here would be First-In First-Out (FIFO) idequeue to maintain a list of outstanding requests.

   Here, the hypervisor would also need to implement some **synchronization directive** to handle race conditions on the physical addresses. This could arise

when multiple guest OS might try to write something on the same physical address. It can probably keep a simple lock to do so.

3. CPU Virtualization :

   Issues -

   Intel provides hardware assistance, where in the guest OS runs at ring 0, and the hypervisor is at root mode (ring -1). All instructions are trapped to guest OS normally, and the sensitive ones are trapped to the root level hypervisor. This incurs a lot of extra cost.

   Solutions -

   Privilege Levels : Now, since we are designing the OS from scratch, we are designing OS such that it is aware of virtualisation and hence, we can modify the privilege levels. The guest OS can run on privilege level 1, and the hypervisor can be assumed to run at level 0, since the OS is aware of its existence. Various kinds of instructions are handled as follows :

   - Normal System calls: These will go to the guest OS, which is at ring 1, by keeping the last two bits of the segment descriptor as 01.
   - I/O Calls: These will need to be handled by the hypervisor, because they need to be directed to the corresponding physical device. Hence, their last 2 bits of the segment descriptor are kept as 00.
   - H/W System Calls: Since these calls require hardware access and updates, they need to be handled by the hypervisor. They are directed to ring level 0 by setting their last 2 bits of the segment descriptor as 00.

   Hence, unlike traditional methods, there is no overhead of going to the interrupt vector of the guest OS and then directing the calls to the hypervisor. Handling the segment descriptor bits is way less expensive.

   Resources : This is easier to handle, since we just need to allocate resources depending upon guest OS' requirements. This is similar to how an OS allocates processing power to different processes, since guest OS can be assumed synonymous to processes.

   The VM's will have a continuous view of large memory, and the hypervisor can keep allocating physical memory space as and when the guest OS use more memory.

Conclusion

Since we have modified both the hypervisor and the guest OS, the design is now scalable and efficient. There are very few remaining overheads for virtualization, as compared to traditional virtualization.