COL 733: CLOUD COMPUTING TECHNOLOGY FUNDAMENTALS

Assignment 4
Specification/Design of Guest OS and Hypervisor

Group 11

Members:

- Ankit Shubham
  2014CS10158
- Ayush Gupta
  2014CS50281
- Kapil Kumar
  2014CS50736
- Deepak Bansal
  2014CS50435

Introduction

When the era of computing was picking pace, the hardware were not designed keeping virtualization in mind; virtualisation being relatively newer idea and it came into picture after a significant amount of time when even home PCs were common. But when people started realizing its potential, they tried to establish virtualization on existing hardware and os; often a software layer solution. Even though the aim was met and the software managed to emulate the virtualization behaviour, it was often too complex, not straight-forward, with a lot of redirections and inefficient. In this report, we are trying to design a system from scratch which suits the needs of virtualization.

Our approach

We looked into different modern resource virtualization techniques in the domains of memory, processor and I/O. We speculated why these complicated techniques were required in the first place and how could we replace them with a far simpler method. The answers to these were always found to be this: due to trying to modify a system that was not meant for virtualization. We then brainstormed on how to address these issues. In this report, we present the ideas that have undergone numerous revisions that make it suitable for virtualization. We have aimed to minimize the role of software and maximize the role of hardware, keeping the scheme as simple as possible.

Design

CPU Virtualization

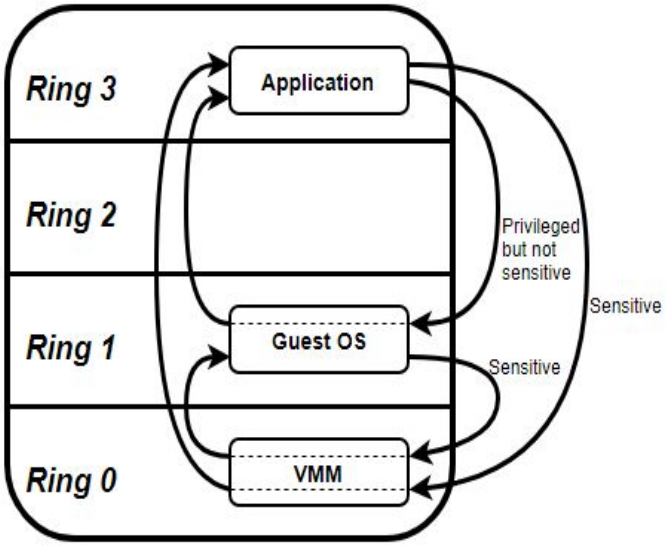
CPU Virtualization is achieved by a variety of techniques, viz. Binary translation, paravirtualization etc. But these seem to be a work-around; we focus on hardware-assisted virtualization. But for that, we first segment all the instructions that a vm will possibly execute into 3 kinds:

- Sensitive(kind 1)
- Privileged but not sensitive(kind 2)
- Neither privileged nor sensitive(kind 3)

The hardware is made such that it is able to handle these 3 kinds of instructions in separate ways. Following are the semantics of what should be done when each of these kinds of instructions are encountered:

- If a trap of kind 3 occurs, there is no context switch required and the current user will handle that trap.
- If a trap of kind 2 occurs, then there is a context switch to guest kernel mode
- If a trap of kind 1 occurs, then there is a context switch to vmm mode
- After handling the trap, the cpu jumps back to its previous mode and continues execution

Following diagram sums up this well:

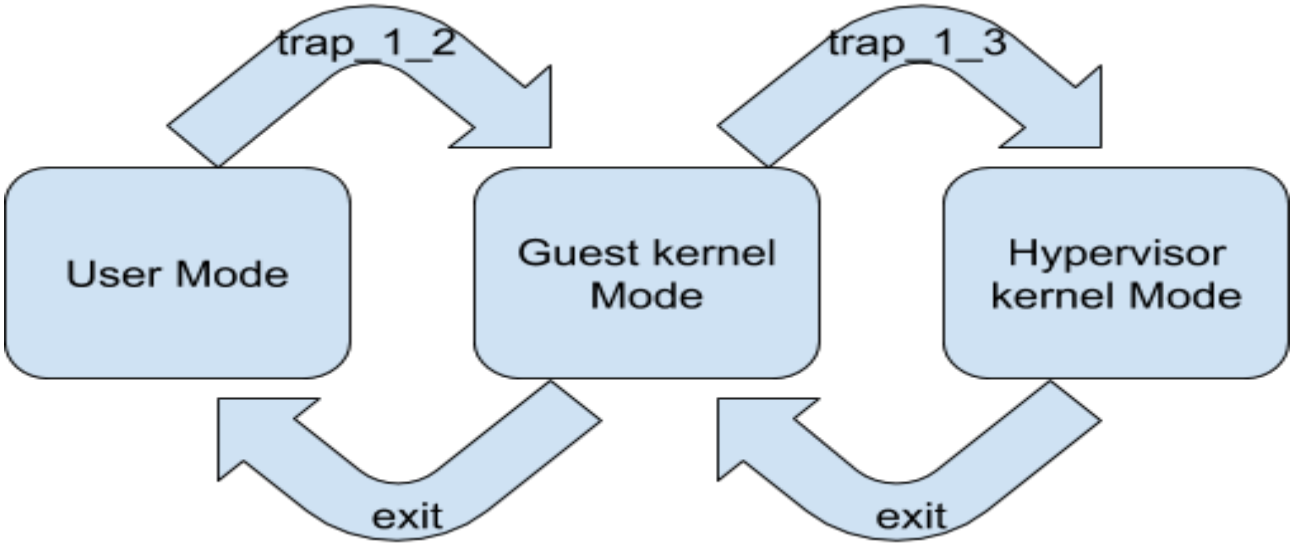


For better understanding, here is an example: trap_1_2 is an instruction of kind 2. Assume that currently, CPU is running in user mode and encounters trap_1_2. Hardware sees that it is of kind 2, so it changes the CPU mode to guest kernel mode and starts executing it. Below is the pseudocode for trap_1_2:

trap_1_2:

```
/*
*
*Here are some syscalls of type 1 and 2. Since this part is being executed in guest kernel mode, these are not trapped further
*
*/
trap_1_3; //a syscall of type 3. CPU now traps to vmm kernel mode
//cpu comes back to guest kernel mode
/*
*
*Here are some more syscalls of type 1 and 2. Since this part is being executed in guest kernel mode, these are not trapped further
*
*/
exit; // this function exits the CPU from its current mode and brings it to that mode from which trap_1_2 was called
```

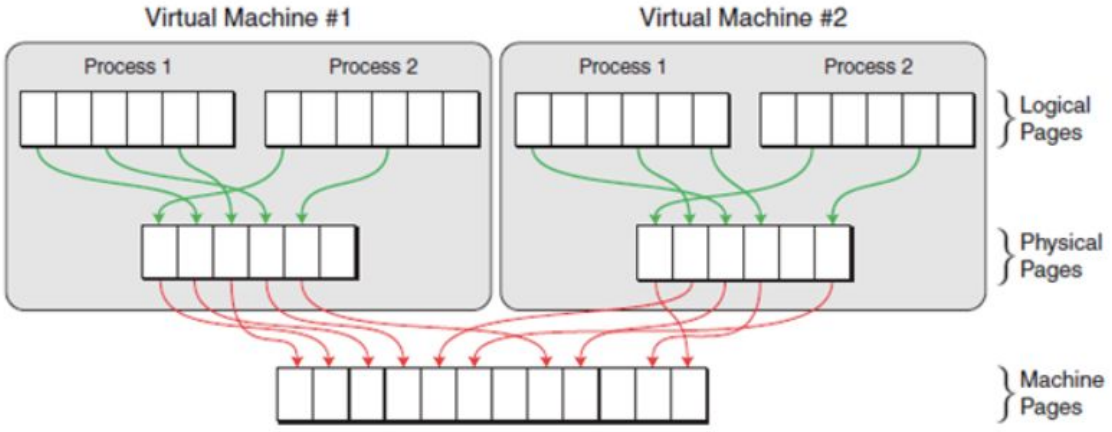
When trap_1_3 is called, which is of type 3, the mode of CPU is further changed to hypervisor kernel mode. Eventually, when 'exit' is called, the CPU gets back to the user mode. Following is the flow of CPU mode changes:



This approach has almost removed the dependency of software. There is no emulation required in this scheme(Unlike trap and emulate model).Hardware recognizes the kind of the instruction and suitably changes the mode of CPU for trapping it. There is no redirection or extra context switches here (unlike paravirtualization); hence greater efficiency and least overheads. Further, this scheme is in accordance with the Goldberg-Popek theory of virtualization.

Memory Virtualization

There is a 2 level of memory translation scheme in conventional vms. First level of conversion is from user space virtual address to guest os space's real address. Second level of conversion is from guest os space's real address to hypervisor space's physical address. Following diagram conveys this more clearly:



We could remove this 2 level of translation and replace it with a direct access mechanism. Following are the details:

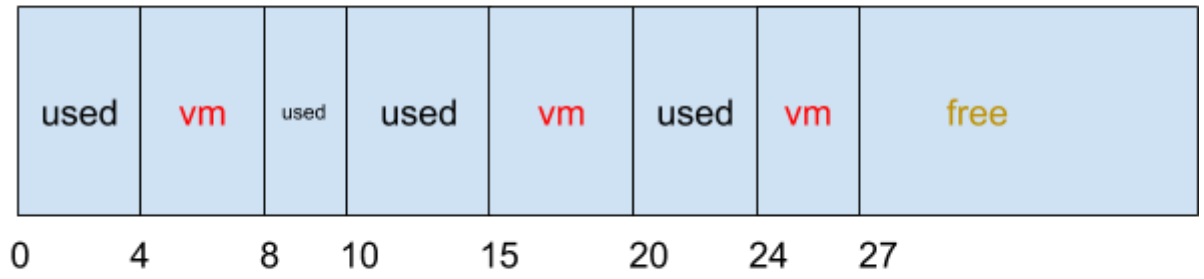
Maintain a priority queue of tuple(base,size) in hypervisor space with base as the priority; call it global memory queue and each vm maintains a priority queue of tuple(base, size) with base as the priority, call it local memory queue. Base represents the address in the physical memory from where a vm is allotted a continuous chunk of size 'size'. For example, a tuple (1024, 512) denotes that vm has been allotted a continuous physical memory from 1024th address to 1024+512 = 1536 th address(exclusive). Let's see the working of this scheme now.

When a new vm joins, it requests hypervisor for memory and piggybacks the required size, 'sz' along with it. Hypervisor traverses its global memory queue, glb_mem_q and looks for a segment which can accommodate the requested size, sz. If for any i, glb_mem_q[i+1].base - (glb_mem_q[i].base+glb_mem_q[i].size) > sz, then it inserts a new tuple (glb_mem_q[i].size,sz) into glb_mem_q. It then responds back to the vm with a success and piggybacks (glb_mem_q[i].size,sz). The vm enters this tuple into its local memory queue.

One obvious problem of this approach is fragmentation. But we have come up with a solution to this too. When a vm requests for a memory of size 'sz', we do not necessarily provide a continuous chunk; instead while traversing the global memory queue, we keep on allocating whatever the free space we encounter in the memory to the vm and move to next even if the size of that free space is smaller than sz. We do it till a total of sz memory has been allocated. Then essentially, we have a list of tuples(base, size) instead of a single tuple(base,size). Following example will make this clearer. Suppose a vm requests a

memory of size 12. Following is the state of the physical memory before and after allocation:

Before allocation

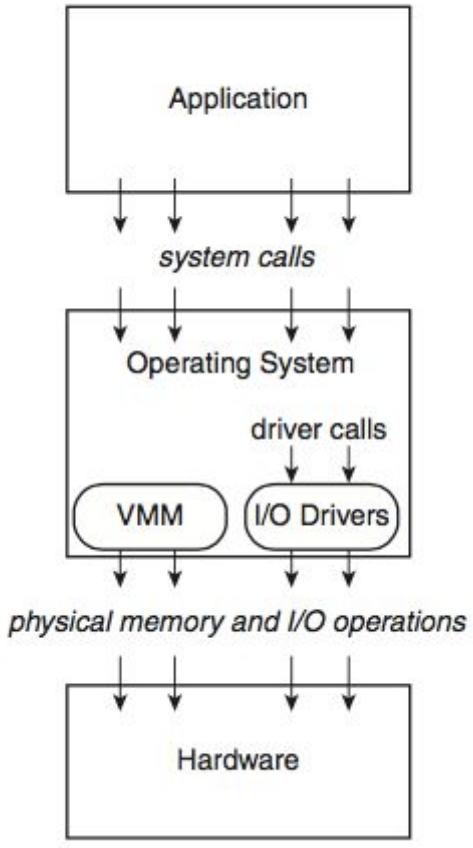


After allocation

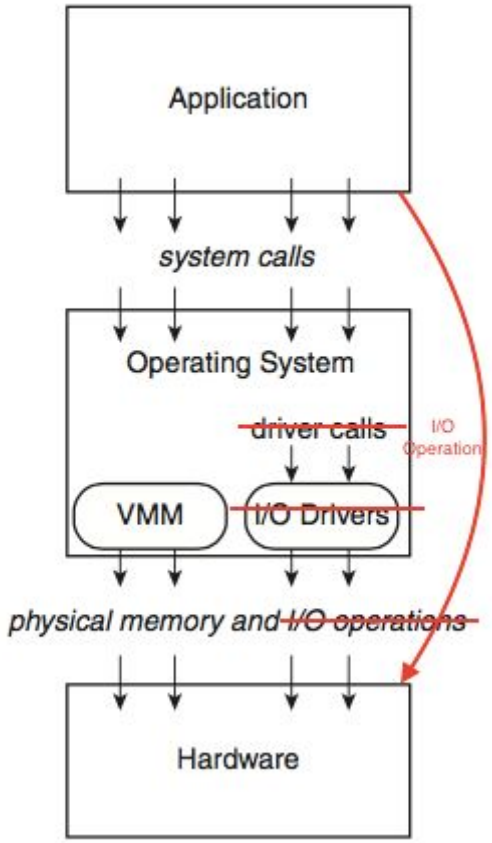
Note that vm has been aggressively allocated the physical memory. In totality, it is given the required size of 12 but not contiguously. Also, the tuples that are going to be added to its local memory queue as well as the global memory queue are :(4,4),(10,5),(24,3).
Same approach will be used if an existing vm demands for more memory.
Let's move to the case when a vm wants to shut down. It requests the vm that it wants to shut down. Hypervisor deletes all the tuples from the global memory queue which are present in the local memory queue of that vm. Finally it removes the local memory queue. All that memory is now freed up and available for reclaim. If a vm wants to shred off some memory, a similar approach will be taken keeping in mind the size of the memory that is being requested to shred off.
Clearly, this method of memory allotment is more direct and needs lesser redirections. This results in increased performance and lesser complexity of the system.

I/O Virtualization

We know about 3 levels at which I/O can be virtualized: at operation level, at device driver level and at system call level. Amongst these, we know that theoretically, the virtualization process could be made more efficient by intercepting the initial I/O request at the system call level. Then the entire I/O action could be done by the hypervisor. To accomplish this, however, the VMM would need ABI routines that shadow the ABI routines available to the user. According to the book, Virtual Machines: Versatile Platforms for Systems and Processes, "These routines would be different for each type of guest OS. Writing the ABI emulation routines would have to be part of the VMM development task and would require much broader knowledge of the guest OS internals than writing drivers. Furthermore, all interactions between the ABI emulation routines and other parts of the guest OS would have to be faithfully emulated. In general, this rather daunting task could be done only if the guest OS is well structured and understood intimately by the VMM developer." But anyway, we are designing the whole system from scratch which itself speaks that we have complete understanding of the guest OS. So, it can be assumed that we have a complete understanding of the system and thus, we enable I/O virtualization at system call level. Below is a diagram that depicts the conventional I/O virtualization paradigm:



And below is the diagram of what we are trying to achieve:

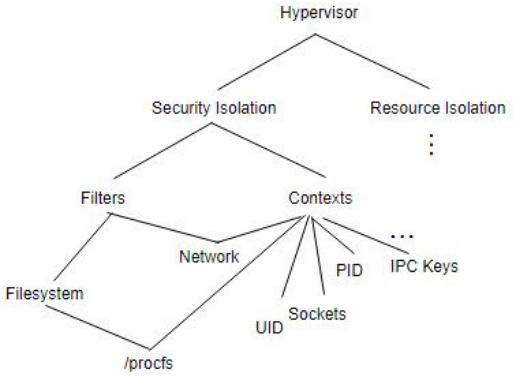


Hypervisor design

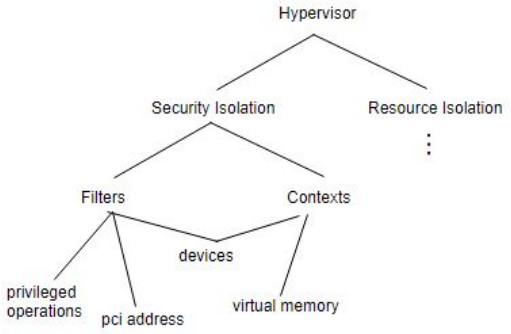
Today's operating systems are not designed keeping virtualization in mind. So, they provide relatively weak form of isolation/abstraction of the VMs.
Hypervisors provide higher degree of isolation. Current hypervisors provide isolation at hardware abstraction layer but our isolation scheme applies to software abstraction layer as well.

- Scheme for security isolation:
    - **Separation of name spaces (contexts) -** global identifiers (such as process ids, user ids, sockets, IPC keys etc.) of each VM are in completely different namespaces, i.e. they do not have pointers belonging to the namespace of other VMs. So, a VM cannot get access outside its own namespace and each VM has it own set of global identifiers.
      This eases the implementation of VM migration, resume, and checkpoint as we are using separate global PID space for each VM. So, for resuming VM, we can re-instantiate processes with the same PIDs at the time of checkpoint.
    - **Access controls (filters) -** Filters control the access of kernel objects, e.g. file system and network. They ensure, by runtime checks, whether the VM has appropriate permissions to access a kernel object or not. This prevents unwanted interaction of processes between different VMs.
- Resource isolation same as existing hypervisors - Allocation and scheduling of physical resources (storage, memory, cycles, link bandwidth) such that each VM has fair share of resources.

Our isolation scheme:



Existing hypervisors:



Conclusion

Overall, our aim was to minimize the redirections, overheads and complexity of the system and introduce simpler mechanisms that naturally yield an efficient system.
Apart from these design perspectives, we also realized that complexity issues in virtualization is not only due to incompatible hardwares but also due to lack of standardization. Today, any os can act as guest os. So, the pressure comes up on the hypervisor to see after all the corner cases(i.e. All the manners in which the guest os can behave and correspondingly hypervisor provides virtualization for all of them) and gracefully handle all the situation. If there is a standardization of guest os keeping the hardware compatibility too in the mind, virtualization will become a lot simpler to grasp and execute.
Another important issue is that guest os not knowing that it is being virtualized. We can remove this strict condition for the sake of improvement. For instance, in the memory virtualization, when we decided to keep a local memory queue in guest os space, it somehow conveys to the guest os that it is being virtualized (because if not, then why it is keeping some 'local memory queue'; it makes no sense to keep this if it is not virtualized). However, it greatly improved the performance too. So, apart from focusing more on hardware rather than coming up with software workarounds, we also need to focus on standardizing the guest OS and the hypervisor.

References

- https://dl.acm.org/ft_gateway.cfm?id=1273025&ftid=431460&dwn=1&CFID=990911311&CFTOKEN=83847667
- https://dl.acm.org/citation.cfm?id=2541946
- Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design), Ravi Nair and J Smith