Advanced Operating Systems - Assignment #1- Fall 2018

Group Members

Name	Roll Number
Aiman	18L-1839
Fareeha Idrees	18L-2084
Maliha Arshad	18L-2106

Executive Summary

You have to create a report on how KVM does the task of CPU scheduling. You are reminded that KVM is a type 1 hypervisor and it is loaded with a Linux kernel as a loadable module. That means, the kernel working along with KVM also completes the tasks of a normal kernel. This means that the kernel has to distinguish among the processes inside the host operating system and VM processes. In this light your report must answer following questions

Question 1 :

How are interrupts handle in a kernel which has KVM module loaded and activated. Describe both scenarios

1. When the interrupt occurs and the target process is in host OS
2. When the interrupt occurs and the target process is inside a virtual machine

Answer 1:

1. On Intel VT-x, all interrupts cause a trap into the host operating system. The host determines if the interrupt is for the host itself (in which case it will handle it, possibly causing KVM to sit unscheduled for quite some time); or for the guest, in which case, it will inform KVM about the interrupt. KVM will then, in turn, inject the interrupt into the guest. In either case, the host operating system takes control first.

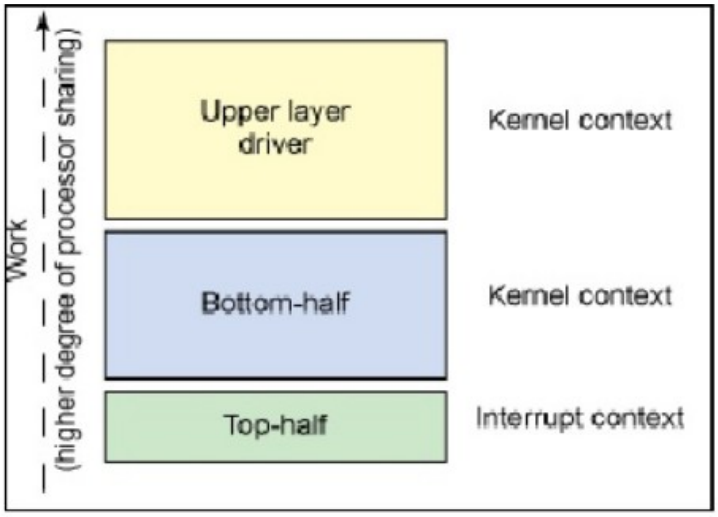
- **When the interrupt occurs and the target process is in host OS**, host kernel traps the instruction and exits into the VM's user space (some instructions may be handled in kernel mode).
- The VM's user-space initiates the I/O on behalf of the guest.
- The VM's user-space then returns to the kernel and the kernel resumes guest code

In all the operating systems, interrupt handling processes are divided into two parts. Since, this project concerns itself with KVM hypervisor, the terminology used will be in correspondence with Linux kernel. The two parts of the interrupt handlers are known as the top half and the bottom half.

The top half is the routine that is registered with request_irq() and the one which actually responds to an interrupt. During this time, interrupts to this device are disabled. Once the top half exits, interrupts are again enabled. Bottom half is executed in the form of a task let, i.e., it is scheduled by the top half to be executed at some later point of time. Thus, the top half and bottom half execute concurrently

and increase the speed of interrupt handling by making device available within a short interval of acknowledging the interrupt. In a typical scenario, the top half saves the device data to a device specific buffer, schedules the bottom half and exits. The bottom half, when processed later, awakens the sleeping process and invokes another I/O if required.

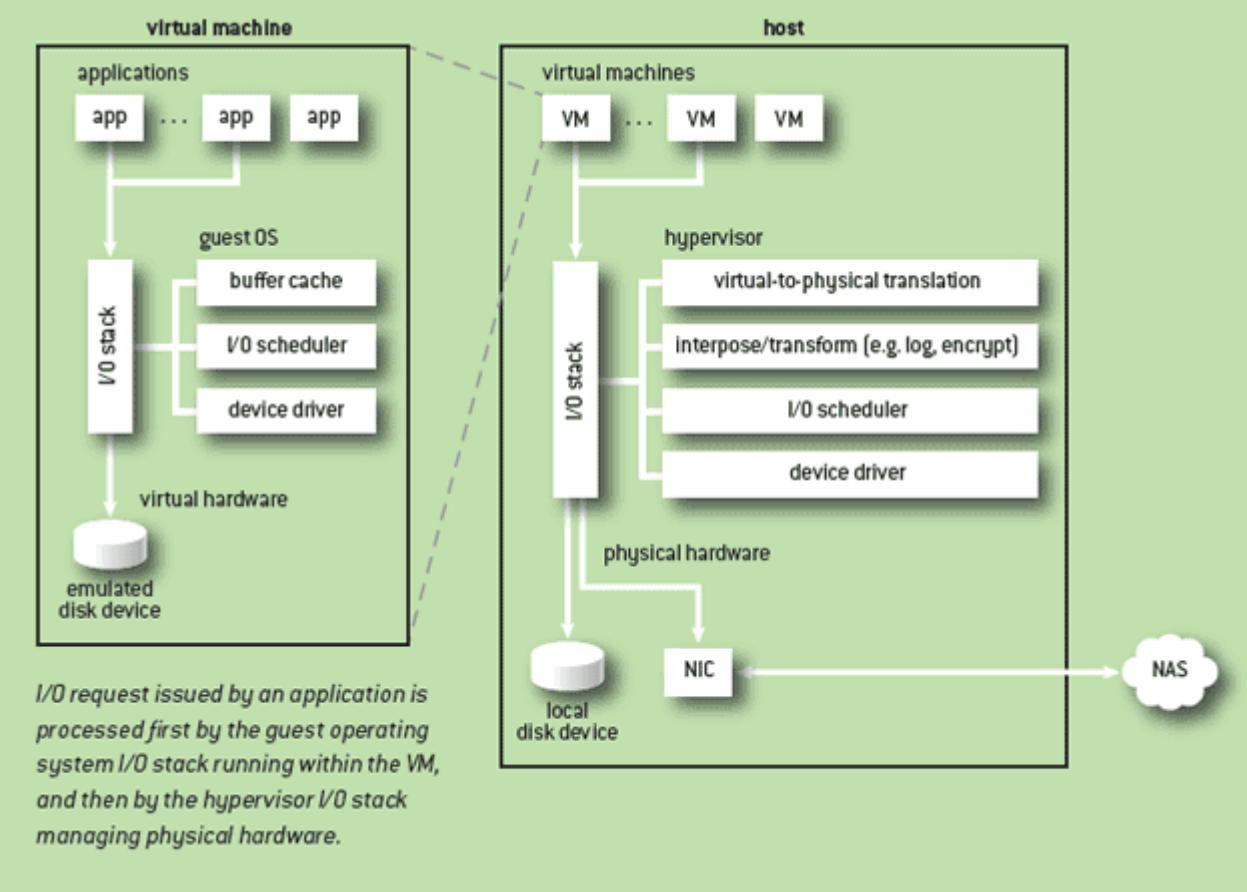
For e.g., in case of a network interrupt, the network interface generates an interrupt. In case of receiving, the top half of the handler receives the data and pushes it into a buffer. The bottom half then processes the data when scheduled. The processor is interrupted by the hardware to signal arrival or transmission of data. The interrupt handler routine determines the scenario through a status register present on the physical NIC.



2 **When the interrupt occurs and the target process is inside a virtual machine,** for instance, an application running within virtual machine issues an I/O request. The I/O stacks in guest OS processes this request. A device driver in the guest OS issues this request to a virtual device. This is a privileged instruction and hence, gets trapped by the hypervisor. Hypervisor schedules the requests from multiple virtual machines onto an underlying shared physical I/O device, usually via another device driver maintained by the hypervisor. When physical device finishes processing the I/O request, the two stacks must be traversed again but in reverse order. The actual device posts a physical completion interrupt which is handled by the hypervisor. Hypervisor determines which virtual machine is associated with the completion and notifies it by posting a virtual interrupt for the virtual device managed by the guest OS.

1. Guest issues an I/O instruction
2. Hypervisor traps the I/O instruction and forwards it to the privileged guest (some instructions may be handled internally)
3. The hypervisor's scheduler has to schedule the privileged guest
4. A guest switch is performed into the privileged guest

5. The privileged guest's interrupt handler is invoked, which causes an I/O process to be scheduled using the privileged guest's scheduler
6. A process context switch is performed
7. The I/O process initiates the I/O on behalf of the guest
8. The I/O process signals (through the privileged guest kernel and hypervisor) that the I/O initiation is complete
9. The hypervisor switches back into the original guest
10. The hypervisor resumes the guest code



Question 2 :

How KVM presents a CPU to a virtual machine. How it uses the VTx technology. What are properties of a VCPU offered by KVM?

Answer 2 :

The kernel-based virtual machine (KVM) is a creative virtualization solution based on Linux. It was

originally developed to utilize underlying hardware-based virtualization extensions. Unlike common hypervisors that perform basic scheduling, memory management themselves, KVM finds the similarity of modules between VMM and operating system kernel, thus resorting to existing Linux kernel for all common functions. In other words, KVM turns standard Linux kernel into a hypervisor simply by adding some virtualization capabilities to it.

KVM is implemented as two components: kernel-space device driver for CPU and memory virtualization and user-space emulator for PC hardware emulation.

The driver adds a character device (/dev/kvm) that exposes the virtualization capabilities to user space. Using this driver, a process can run a virtual machine (a "guest") in a fully virtualized PC containing its own virtual hard disks, network adapters, and display. Each virtual machine is a process on the host; a virtual CPU is a thread in that process. kill(1), nice(1), top(1) work as expected. In effect, the driver adds a third execution mode to the existing two: we now have kernel mode, user mode, and guest mode. Guest mode has its own address space mapping guest physical memory (which is accessible to user mode by mmap()ing /dev/kvm). Guest mode has no access to any I/O devices; any such access is intercepted and directed to user mode for emulation.

Standard Linux kernel can derive virtualization capabilities from a loadable kernel module. This kernel module, which is the key part of KVM, serves as a device driver and is responsible for providing CPU and memory virtualization features with the help of hardware extensions.

How it uses the VTx technology?

Although there are software techniques that introduce viable solutions to x86 virtualization, they have various limits that prevent them from fitting all scenarios. Therefore, hardware extensions are needed to be used so that faithful virtualization can be implemented.

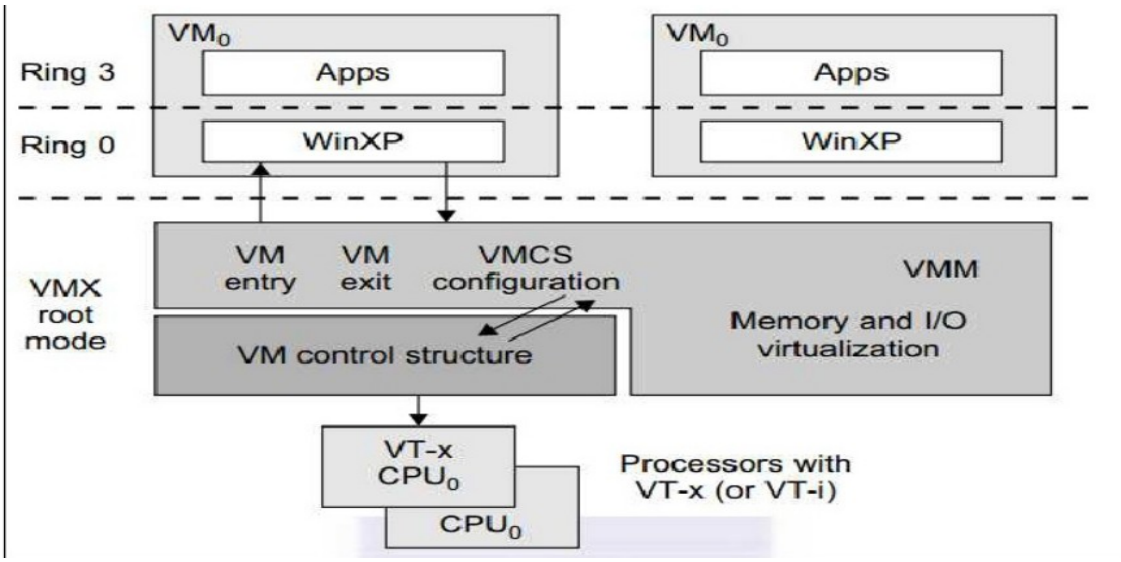
KVM requires a processor with hardware virtualization extensions because KVM will not work in CPUs without virtualization capabilities. **It runs on processors that support x86 Hardware Virtual Machine (VT/SVM instructions set)**. KVM adds a driver for Intel's (and AMD's) hardware virtualization extensions to the x86 architecture.

Intel VT are instruction set extensions that provide hardware assistance to virtual machine monitors. They enable running fully isolated virtual machines at native hardware speeds, for some workloads.

The advantage is that CPUs with Virtualization Technology have some new instructions to control virtualization. With them, controlling software (called VMM, Virtual Machine Monitor) can be simpler, thus improving performance compared to software-based solutions. When the CPU has support to Virtualization Technology, the virtualization is said to be hardware-based or hardware-assisted.

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. In general, the host operating system and VMM will run in root operation, while the guest operating systems along with their applications will run directly in non-root operation.

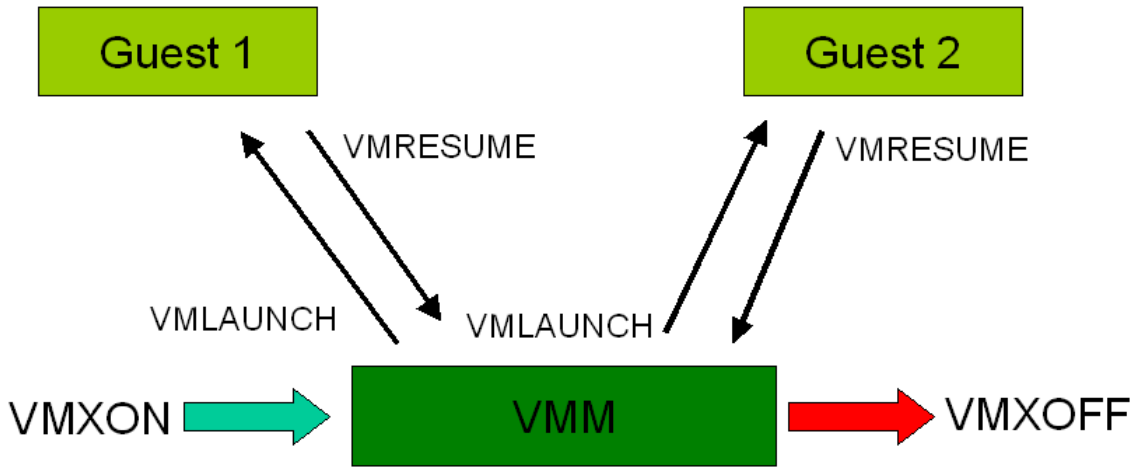
Since both root and non-root operations include all x86 privilege rings, guest code can run at its intended ring, which means no de-privileging is required. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits.



Processors with Virtualization Technology have an extra instruction set called Virtual Machine Extensions or VMX. VMX brings 10 new virtualization-specific instructions to the CPU: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, and VMXON.

Usually, only the virtualization controlling software, called Virtual Machine Monitor (VMM), runs under root operation, while operating systems running on top of the virtual machines run under non-root operation. Software running on top of virtual machines is also called “guest software.”

To enter virtualization mode, the software should execute the VMXON instruction and then call the VMM software. The VMM software can enter each virtual machine using the VMLAUNCH instruction, and exit it by using the VMRESUME instruction. If the VMM wants to shut down and exit the virtualization mode, it executes the VMXOFF instruction.



What are properties of a VCPU offered by KVM?

KVM kernel module provides various services like creation of virtual machines and virtual CPUs, allocation of memory to specified virtual machine, and injection of events. It exposes those services via a character device */dev/kvm* which can be used by the user-space emulator through *ioctl()* system call.

KVM kernel module executes as a bridge between user mode and guest mode. On the one hand, the user-space emulator can request execution of a virtual machine by issuing *ioctl()* system call. When the kernel module receives this request, it will make some preparation like loading VCPU context to VMCS, injecting pending virtual event and disabling interrupt, etc. It then switches control to guest mode by issuing special instruction offered by hardware extension. From then on, the virtual machine starts executing in guest mode.

On the other hand, if a VM exit caused by some special I/O instruction occurs later, the kernel module will take control to handle this exit. It first restores certain environment like enabling interrupt again and then handles this VM exits by switching back to user space for I/O emulation.

A virtual CPU is a thread in the QEMU-KVM process. KVM turns the single-threaded execution model of QEMU into a multithreaded model. Since every virtual machine on KVM is a process of the host Linux, a QEMU process therefore represents an instance of virtual machine. In other words, the host Linux views a virtual machine as a QEMU process.

While I/O thread is used to manage emulated devices, VCPU thread is used to run guest code.

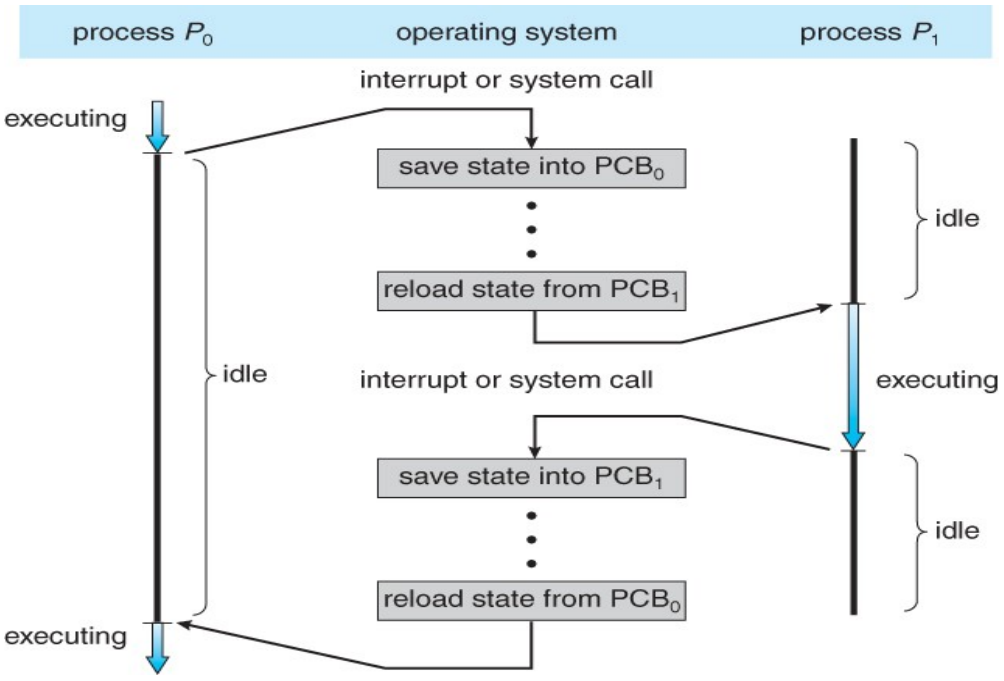
Question 3 :

How a process switch works, which is commonly termed as context switch. Describe both scenarios

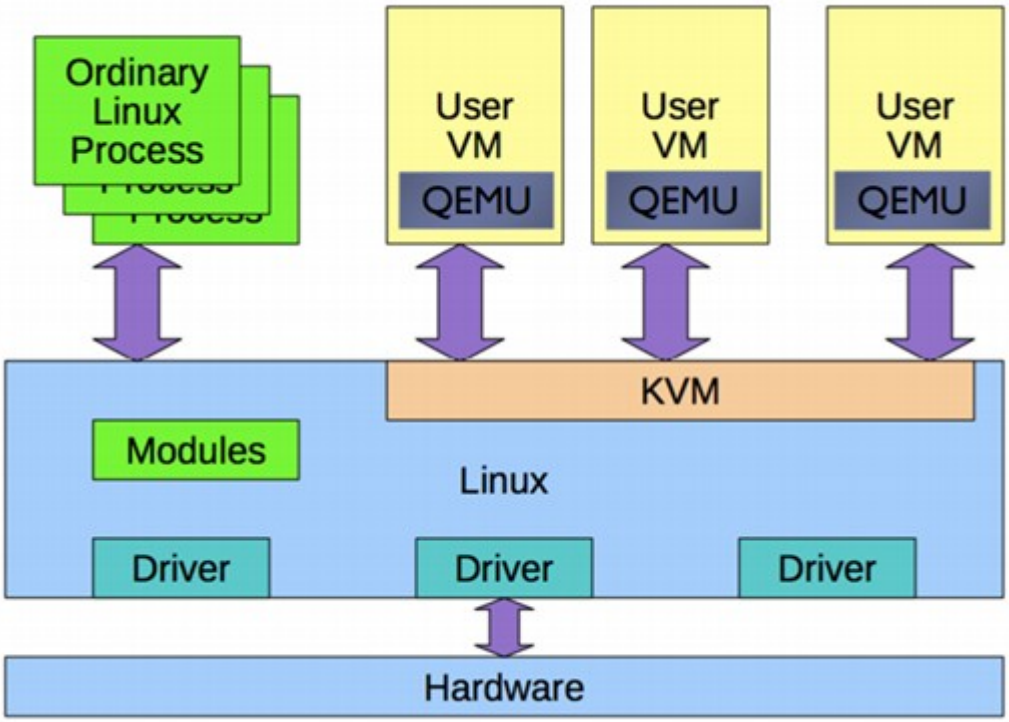
1. When a process switch occurs in the host operating system. This is simple, the text book contains the details
2. The challenging part is to explain how a process switch works when KVM is loaded. First address the topic for process switch in the host operating system
3. Address the issue of process switch inside a virtual machine.

Answer 3 :

1. To control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended. When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information. A state save of the current state of the CPU is performed, be it in kernel or user mode, and then a state restore to resume operations. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. The diagram below represents the working of a process switch in CPU.

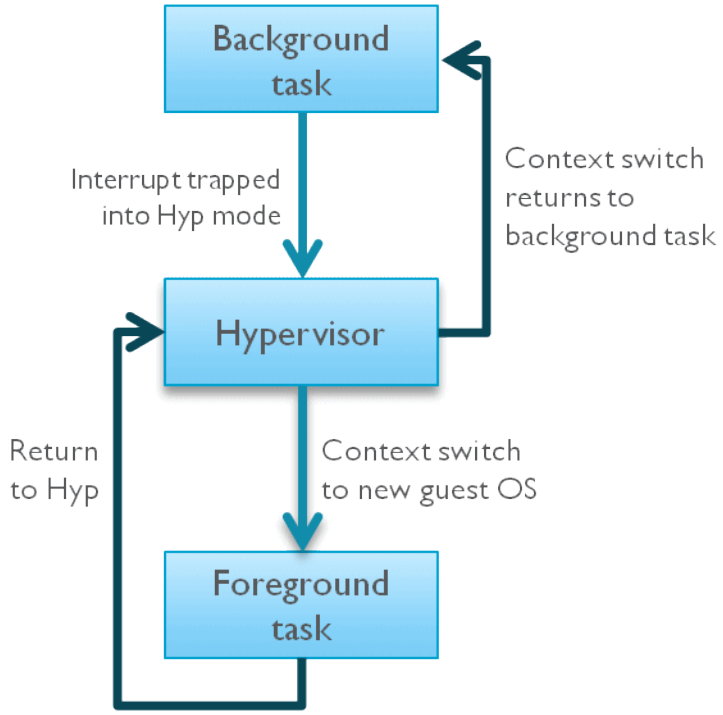


2. In a host operating system, process switching works simply by saving the essential register values and addresses in PCB of a specific process that is to be paused. Control transfers to the other process and its execution is resumed after reading values from its PCB. The Kernel-based Virtual Machine adds a virtual machine monitor (or hypervisor) capability to Linux. Using KVM, one can create and run multiple virtual machines. These virtual machines appear as normal Linux processes and integrate with the rest of the system. In KVM, the processor manages the process states for the host and guest operating systems, and it also manages the I/O and interrupts on behalf of the virtualized operating system. Every guest machine has its own kernel and is a process from the host point of view with a PID. The virtual machine executes in this guest mode which in turn has user and kernel mode in itself. KVM creates for each virtual machine a virtual CPU to hold the guest CPU state. When the guest operating system executes a privileged instruction, the hardware virtualization mechanism triggers an “exit” from the virtual CPU execution context to the VMM.



3. Virtual machines "guests" run in *user* state, in which a subset of instructions is permitted. In particular, the instructions that perform I/O, enable or disable interrupts, are forbidden, and generate an exception or trap when executed. This trap causes a context switch to the hypervisor. The hypervisor saves the current state. Every time a virtual machine executes a privileged operation, then at least two context switches need to deal with, once switching from guest to host, and then back again. This process happens every time the guest does I/O, context switches between its processes, sets or responds to a time or I/O interrupt, or one of its applications executes a system call. Thousands of clock cycles spent each time, at a

frequency that can add up to thousands of times per second. Context switching is overhead because it is cycles or the time that the processor is being used but no user code is executing, so no directly productive computing is getting done.



Question 4:

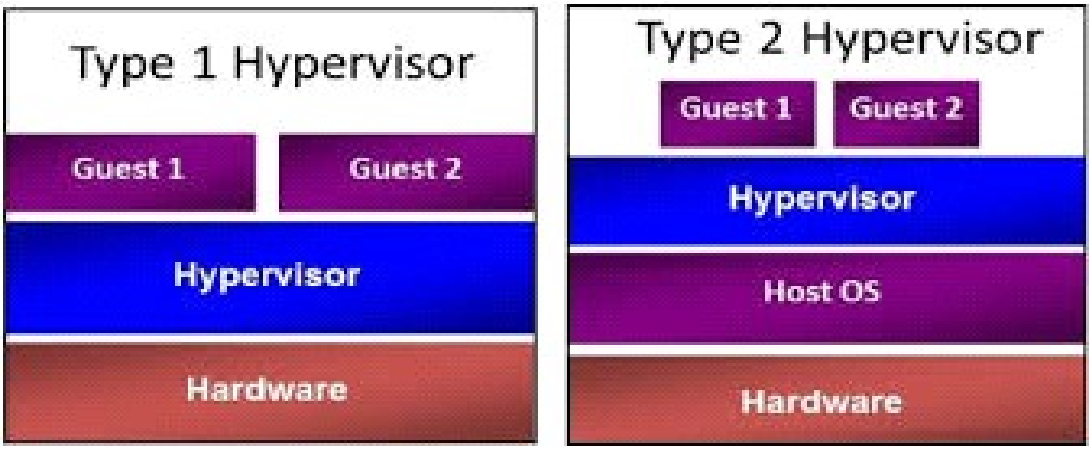
How CPU scheduling works in KVM enabled kernel. For ease take the example of Round Rabin Algorithm and describe its functions properly. You have to go at greatest length of details. For example, you must describing how a specific register is used for quantum definition, and what registers and CPU commands are used to schedule a timer interrupt. You must describe these things and other relevant details for both scenarios.

1. How CPU scheduling (RR algorithm) works for the processes in the host operating system
2. How CPU scheduling (RR algorithm) works for the processes inside virtual machine
3. What happens if host virtual machine has different scheduling algorithm than the virtual machine

Answer 4 :

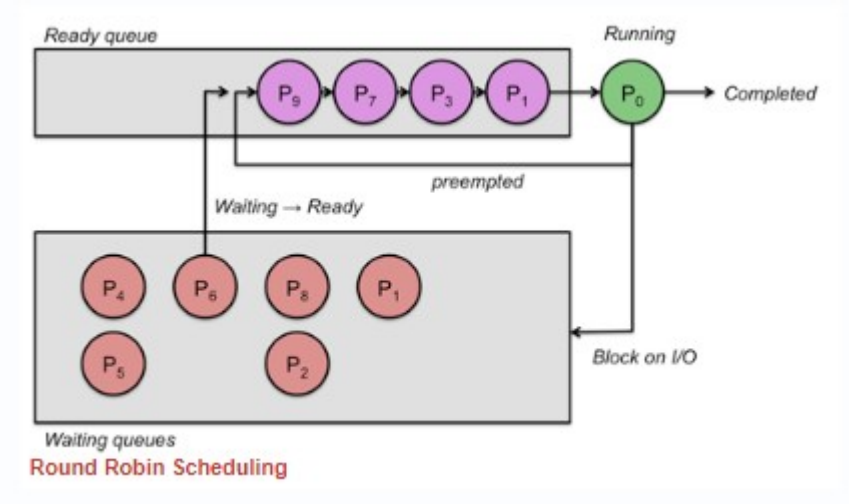
In a KVM enabled kernel there is an extremely high chance that the CPU will get scheduled by the basic built-in scheduling algorithm. If we talk about Linux then it basically uses CFS as its

scheduling algorithm. There are quite a lot of scheduling algorithms. Some of them schedules on the basis of time required for the job to get done while some are used to schedule on the basis of priority of the job.
Now, we have taken round robin algorithm for the sake of simplicity and explanation. Round robin basically is a preemptive algorithm and it works on the basis of first come first serve.
It have a pre-defined quantum number that allows it to divide the CPU time for each process. It does not matter how short or long a process may be, it is given the CPU time and CPU would be taken back when the time is over or may be before that if the process is complete before the time ends.
For scheduling a timer interrupt in CPU we would need to code it in the kernel. Timer interrupts are generated automatically and we can control them through the code in OS.



How CPU scheduling (RR algorithm) works for the processes in the host operating system

On Host OS level, all other virtual machines working are like processes. Host OS assigns equal time to each of the virtual machine and while that time is over the machine will have to be in waiting state while the other virtual machine will have the CPU. This specific time slot is known as time slice or quantum. There is a register called quantum register that saves the state of virtual machine and the state of I/O while the virtual machine reaches its end of the CPU burst time.



How CPU scheduling (RR algorithm) works for the processes inside virtual machine

While if we look at it on the level of the virtual machine then we can say that it works exactly same the only difference is that this time these are real processes and Virtual machine divides the quantum assigned to it into more chunks and assign each process with a specific time for it to execute its queries. It's like many circles in a circle. The outer circle gives lead to the inner circles one after the other for a specific time and same goes for the inner circle. The only difference is that there are no more circles in the inner circle.

What happens if host virtual machine has different scheduling algorithm than the virtual machine

It may sound like an issue but it doesn't really matters if there are two different scheduling algorithms working on each of Host and Guest OS. Because, each one of them is going to work on its own. They are not depending on each other it's just that Host is going to give access to the Guest but as it is going to give it anytime so it does not matter that what algorithm will be used. In fact in a single system more than one algorithm works on its own queues. So, as long as the work is getting done efficiently it doesn't matter what specific algorithm is being used.

References

https://lwn.net/Articles/658511/

http://www.academia.edu/15984262/Virtual_Interrupt_Handling_to_Reduce_CPU_Overhead_in_I_O_Virtualization_-_Full_Thesis_Report

https://www.hardwaresecrets.com/everything-you-need-to-know-about-the-intel-virtualization-technology/

https://ieeexplore.ieee.org/abstract/document/5711095/

Zhang, Jun, et al. "Performance analysis towards a kvm-based embedded real-time virtualization architecture." *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*. IEEE, 2010.