There are number of instructions OS has to use and those we don't use in user space applications.
Eg: modifying CR3

So what happens when both user applications and kernel run in user space in virtual machine?

What is the difference between application doing privileged operation?

When each instruction is called in user mode, hypervisor should know that it is in user mode and call the illegal instruction trap handler in the guest OS. Because such operation is illegal as in user mode, we can't call such privileged instructions.

If the instruction is called from the kernel space, hypervisor emulates the instruction and updates its internal copy of CR3.

These is one variable in virtual hardware state, that says that it is in user mode or kernel mode. Then whenever we do a system call that traps hypervisor, hypervisor decides should the instruction work or not—should it call illegal instruction handler in guest OS or should it emulate the instruction?

This way we have all the virtual machines running—
in complete isolation.

So when we have all the details like different conditions we have to trap on, all the page fault handlers, etc., we will have a virtual machine and each such virtual machine can run a complete operating system and would never know about other ones running next to them. We will have complete isolation between them, this is useful for a lot of things.

E.g. web hosting, where we can have VM as a web host. This will be a lot cheaper than having dedicated hardware for it.

In such cases, when you are paying a good amount of money for a web server, and if you would like to be able to administrate the web server that you are paying for, but you don't want anyone else administrating it at the same time. So we can have separate VM, and we can get to do whatever we want with your applications running on your VM and others can also do the same with their applications running on their VM.

There are instructions in the kernel that examine or modify state that needs to be virtualized, with some of the aspects of the Intel memory architectures like segment descriptors, etc. There are various instructions that going to be emulated in the kernel but which don't trap. If basically if you execute them in kernel mode, they do something & if you execute them in user mode, they just don't do anything. They don't trap or anything - they just do no-op.

And if you try and run kernel in user mode like one in virtual machine, whenever it goes through one of those instructions, nothing would happen, it won't trap into the hypervisor, it won't
modify any processor's state, it will just skip that instruction and that would be not right, because you are using that instruction because you want to do something with it.

So instead, to avoid such problem we can use Binary Translation technique.

VMware uses the binary translation technique which is very similar to the job of JVM.

In JVM, we emulate byte code stream but in background when you are emulating, you have a just in time compiler which takes heavily used pieces of code & translate them directly into machine code. So whenever you hit the section of code that's been translated you can run that as native code instead of emulating.

Binary translation is based on x86 to x86 translator. In fact some cases make exact copy of the original instruction.

The binary code that the kernel of guest OS wants to execute on the fly is translated & stored in a translator cache. User applications will not be touched by binary translator (CBT) as it knows it assumes that user code is safe. User mode applications are executed directly as if they were running natively.

Direct Execution

(user mode guest code)

---

User mode applications are not translated, but run directly.

---

Binary Translation (kernel mode guest code)

(user applications are not translated, but run directly.)

Binary translation only happens when guest OS kernel gets called.
The kernel code has to go through the x86 to slightly longer x86 code translation. We can say that the kernel of the guest OS is no longer running. The kernel code in the memory is nothing more than an input for GT, it is the GT translated kernel that will run.

In many cases, the translated kernel code will be the exact copy.

However, in many cases, GT makes translated kernel code a bit longer than the original code.

Overhead of Virtualization:
Every privileged instruction, by a VM causes a trap, as it is trying to execute a "resource management" instruction while running in a less privileged space. VM always intercepts all these traps and emulates the instruction.

To improve performance, we need to minimize number of traps, so that reduce time required to take care of various traps.

So, one other way we can do the virtualization is the way how Intel & AMD's new processors handle virtualization. Instead of fully virtualized processors, they have the architecture looks like:

```
User
Supervisor
Root

Guest OS
```

Root contains set of instructions which tells processor that which instructions in supervisor mode should trap.
The table in front tells the emulator which instructions to disallow & force a trap on. So if we want to traps on these instructions, and so whenever one of these things happen, you will get a trap into virtual mode nor.

Normally the root part is disabled by default in BIOS. It's not ever going to be active unless you initialize it.

**Disk Drive:**

Disk drive is a recording magnetic signal on the surface of the disk, & reading it back.

![Disk Drive Diagram]

Difference between CD and disk drive is CD has its data in spiral starting at middle & going out. But disk drive have it in circles. So each circle is a track & each track in broken up into the sectors & each sector is some number of data bits plus whole lot of error correcting coding plus some additional signals.

So what the performance characteristics of disk. So if you have a particular piece of data on a disk, there is two sources of delays - before we can read that data - associated with reading/writing data on computer's disk drive.

There is seek-time & there is rotational delay.

**Seek Time:** In order to read/write data in a particular place on the disk, the read/write head of the disk...
needs to be physically moved to the correct place. This process is known as seeking & time it takes for the head to move to the right place in the seek time. It varies depending on how far the head's destination is from its origin at the time of each read/write instruction.

Second source of delay is rotation delay. Time required for the addressed area of the disk to rotate into a position where it is accessible by the read/write head.

Maximum rotation delay is time it takes to do a full rotation (as the relevant part of the disk may have just passed the head when the request arrived)

Maximum rotational delay is simply the reciprocal of the rotational speed.

Eg. 7200 RPM is the rotational speed. Then maximum rotational delay will be $\frac{60}{7200}$ i.e. about 8 ms

Disk speed depends on disk capacity. On the disk, the disk arm moves close to the track & start reading the server information of the disk. Encoded in each track is some information about which track it is, & so when you get to where a track is, then it has to make these microscopic adjustments to find it. So if you have extremely fine tracks on it, you will get good disk speed.

Lastly, we have a disk characteristic affecting causing delay is difference in track speed. Data comes out of disk head at a certain speed, so in any generation of the disk technology, bits on it are of certain size, you can fit more of them around the outer size track than you can around the
Inside tracks, eg in CD-ROM, there it spins faster when it is reading from center, than it does when it goes to the outside.

If you are listening music from disk plate you will want that the bits come out at a constant rate. So we need disk drive moving at constant velocity. We don't want to change its speed everytime the head moves from track to track.

Disks are divided into zones, so you have 5-10 zones on the disk, where in first zone you have certain number of sectors in it. But there is a constant density of bits on each of these sectors. Limiting factor is typically how fast you can read bits out through the disk head is how they are close together without messing each other up. So you pack them in about the same density on each of the sectors in tracks.

Bit density remains constant so that the speed of data increases.

Logical protocols & controllers:
We deal with data on disk as series of disk sectors. Disk isn't like memory. It's a like - you send commands to it & you get responses.
The sectors on disk has logical block addresses. So if you have 16B disk, that means

\[ 2^{38}/2^{3} = 2^{35} \] blocks.

And so if you want 110th block, you need to send command saying "read 110th block" & you get back the data.

So there are these two fundamental commands read & write, who takes addresses and length as parameters.
There are also all whole bunch of complex instructions, e.g., for getting information about drive, hibernating drive, etc. But fundamental commands are: read & write.