Virtualization

Davide La Rosa

March 16, 2010

1 Introduction

Most of all the work in computer programming and also in computer science has to do with defining hierarchies of machines, this is both true when you do programming languages and when you design computer architectures, when you design chips and so on, because basically everything you do is broken down into simple problems. You have to translate human problems in some sense down to bits, because what we have at the bottom level is a boolean logic ad this is a huge gap. Of course everything is done by splitting the translation process into several levels, and from a more technical point of view, everything that we do when we design systems at high level programming models to lower level architectural systems is actually breaking down the complexity of the problem by defining an hierarchy of machines each one with a more rich set of functions and each one implemented on top of the simpler one.

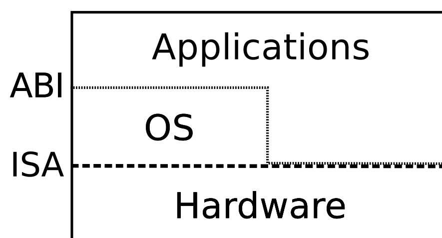
Virtual machine means that we are developing an abstraction of a machine and this abstract machine will be used to execute some program that we have at higher level. This particular layer is called virtual because we are looking at the execution of real code, that means we are on top of the real machine, the hardware (cpu, ram, disk and so on). On top of the hardware usually we have already some kind of more abstract layer which is the operating system because most popular application will not run unless you have an operating system. This already provides you a different level of abstraction because allows you to use input/output devices just like some kind of abstract things without having to care of the real reading/writing on a disk, flash ram or whatever and without having to look to the internal organization of the computer. If we look deeper in each level, we can find a lot of other layers inside them, but we do not want to go into the details. From the point of view of executing an application on a specific machine, the operating system completely shields the application from the hardware. On the other hand there is one thing that cannot shield your application from and this is the machine instruction set. There are some basic things that the applications have to deal with and these are the CPU instructions because they are written with them. Almost everything else can be dealt by the operating system: managing memory, managing input/output, and stuff like that. A few things like accessing the OS functions (call to a subroutine, software interrupt, trap instructions, privileged call) depend on the hardware we are using (x86 or Sparc, Mips, PowerPC) and this is visible in the application or program code.

The first problem that raises is interoperability, it means that if you have programs compiled to work on a specific architecture you cannot make them

work on a different one, even if the OS are the same (for example a program compiled for a Macintosh PowerPC cannot be easily ported to a Macintosh based on Intel x86). The interface between the hardware level and the levels upon it is called ISA (Instruction Set Architecture).

Another problem that comes into play is that when you design a system and the system is still used after five or ten years, then as the system ages, you actually have to preserve the ISA interface because otherwise people working with the operating system or developing applications will not be able to exploit new architectures. The vantage is that a new machine, possibly faster than the previous, is still able to run the same OS and applications. Usually, to exploit new functionalities at ISA level, we have to use specific libraries (that work as translators) and we can think at them as a part of the OS. Summarizing, when we develop a new architecture and you have to keep fixed the same ISA, you are going to have problems, either of performance or portability even with slightly changes. One solution that has come out already in the '60 is to provide an emulated machine in such way that whatever runs on top of it has no way of understanding that is not running on a real machine.

The concept of virtual machine is that we are adding between any two levels a particular part of software which provides to the upper levels the illusion of working on a specific machine.



Another definition that at this point comes into play is the Application Binary Interface (ABI): is what actually the application sees of the real machine and this includes a part of the ISA and other functionalities provided by the OS. When designing a virtual machine we have to address a boundary (for example ISA or ABI) and provide a complete emulation of it. We can think of a virtual machine as providing some kind of execution environment for code which is not natively executable on the machine and we can classify it in terms of what kind of programs can run inside the virtual machine. One first class of VM that we have is those designed to execute and host processes. Those VM provide an abstraction of the resources that is more or less the same as the OS provides. These kinds of VMs are already present in the most of OS because they are multiprogramming, that means they can run several processes at the same time. Another characteristic that a VM can provide is the emulation of different machines where they are not availables. This involves an improvement of the interoperability because we can have one process designed to be run on an old machine which is able to run and interoperate with other processes written for a newer machine.

1.1 Techniques

Different techniques can be used to obtain the emulation for either processes or a whole machine.

1.1.1 Interpretation

Interpretation means that, given our process code, we take one instruction at a time, we translate it on the fly and we execute them in some memory space which is the one originally designed for the process. This technique implies certain overhead due to the just in time translation of the instructions.

1.1.2 Translation

Translation is the opposite approach, we take longer parts of our program that has to be executed and we translate them into the language used on the level below the virtual machine. In other word it means that the program is recompiled to run directly on the underlying layer. This technique implies an initial cost to translate the programs, but then the recompiled code will run much more faster than the previous case.

1.1.3 Dynamic translation

Dynamic translation is half way between the first two techniques. In this case we are both providing the interpretation and the translation functionalities: we start executing the code interpreting the instructions that we are going to emulate but whenever we notice that a piece of code is used very often, we compile it and we keep it in some kind of cache. During the execution, therefore, some instructions will be interpreted and some other instruction will be translated.

2 Application level abstraction

A particular instance of virtual machine applied to processes is the support for high level languages. In this particular case we have at the upper layer some kind of language abstraction and we know that at some point we need to increment this language abstraction on top of some hardware (and possibly on the operating system). The target is: we want to compile our language into something which is not dependent on the hardware. The idea goes back at least to Pascal language, therefore in its design there was the choice of being completely independent from the hardware. This means that once the Pascal code had been analyzed by the lexical analyzer and ran through the compiler, the latter produces some kind of intermediate code called p-code. The p-code was basically a set of instructions easily reproducible on most of the cpu at the time extended with a small set of I/O functions, used by a stack machine (a very simple execution machine based on stack). If we don't want this approach, by further compilation we can obtain an executable code directly for our hardware and operating system. The same approach we can see it in Java, if we replace Pascal with Java and we change the name of p-code with byte-code we are basically at the same point. There are obviously different tradeoffs between

Pascal and Java in terms of complexity and size of virtual machines, and also with respect to security because when the byte-code was designed, it has been kept in mind security issues like signing the byte-code in a such a way that it cannot be tampered with. Actually, a limitation of the Java virtual machine is that it is designed to be completely interpreted and still we cannot compile the whole program due to the fact that certain things only happens at the boundary of the byte-code instruction (for instance checking interrupts). However, in general, a virtual machine can be designed with the full range of options (for instance the VM of Microsoft .NET framework which has been designed in order to allow the most extensive use of just-in-time compilation).

3 System level abstraction

A virtual machine at system level aims to emulate a whole system. This is necessary for example when we have an application written for a particular OS and that combination of application and OS don't work on our hardware. The issues addressed by this kind of VMs are:

- interoperability
- portability
- efficiency in resource usage
- isolation and security

In any hierarchy of machines, if we have programs running on top of a common implementing layer, there is always the risk of unwanted or hidden interactions (for instance two users on the same OS which are able to read each other the file they are owning). This risk is greater if the applications or users or whatever, are running on top of the same machine abstraction, for instance two Java thread running on the same JVM can in principle exchange any data. Providing improved isolation and security is the key for reduce at the minimum possibly unwanted interactions.

Must be noted that adding more levels of VMs don't give us a complete security because there still be the possibility that through the VM, some unwanted interactions may take place. It has been shown that there is some methods to understand if an application for instance is on a virtual or real machine, and if we know that we are on a VM there are mechanisms with which you can gain informations on what is running on the same VM (monitoring disk access time, network bandwidth, memory availability, ...).

Efficiency in resource usage means that by sharing the hardware resources, we are able to make a more efficient use of them. Writing a system level VM that completely emulate a machine at ISA layer means that the technique used (interpretation, translation or dynamic translation) strongly influence the performance of the emulation. There are a lot of details that have to be addressed: kinds of instructions that we want to emulate, number of registers that our machine will have, complexity of the code that emulates the instructions and so on.

The memory space of every VM has to be remapped into some space which is available to the hardware machine, so we will have a memory image for each

machine. A particular case is when the emulated architecture is the same as the emulating architecture (for instance running x86 code on a VM running on x86 machine), this very often allows to simplify a lot the emulation. Therefore most of the emulated instructions have the same meaning of those at the level below the VM. Modern processors allow the operating system programmer to setup registers for relocating all the spaces in the memory, these are special registers in the memory management unit.

3.1 Hypervisor

Usually the VM support when we talk about VM at system level, is the hypervisor. It is a part of the implementation of the VM mechanisms which provides all the resources and the functionalities which are needed by the guest OS. Since the guest operating systems are usually the same that we would use on a bare machine, of course there is a lot of work to do in order to trick those OSs into thinking that they are running on a real machine. Of course we don't want to modify these OSs, or at least as little as possible. There are different kinds of hypervisors:

Native hypervisor or bare-metal (type 1): It is designed to run directly on top of the hardware, so it is able to use the components inside the machine (pci-express bus, northbridge chip, disks an so on) and it provides these resources to the different instances of guests operating systems.

Hosted hypervisor (type 2): It is hosted in an operating system but it still provides to the upper level the image of the hardware as if the guest OS is the only one.

In the case of *native hypervisor* we need an OS which has a special role, it must know that it is not running on the hardware and has a special way to interact with the hypervisor in order to control the real hardware (for example to shutdown the real machine and not one of the virtual instances). This kind of hypervisor is made in this way: we have several VMs, inside these VMs we can run our applications, each VM doesn't know anything either about the other VMs or being running on top of any stack of software. One of this VM is running the control OS, it can be any kind of OS but this particular instance has a mechanism to communicate with the hypervisor.

In the case of *hosted hypervisor* we have on top of the hardware an host OS and an hypervisor which hosts several VMs. In this case of course, the host OS behaves also as control OS.

3.2 Protection rings

Currently cpu architecture provides several protection rings, that means the processor in any moment can be in different states. There are four rings in all and each instruction belongs to a specific ring. In this way different kinds of machine instructions are allowed at different execution level. Typically the OS kernel uses just one ring for its execution, the ring 0, and at that level it can do everything on the machine. Other rings like 1 and 2 can be used to execute device drivers and the last one, the ring 4 can be used to execute applications. What often happens is that only ring 0 and 4 are used to execute system and user

code. This make the development of an hypervisor easier, therefore the guest OS can be executed in ring 1 or 2 and the hypervisor will fake the instruction of the guest OS that must be run at ring 0. This implies an overhead due only to the emulation of the instruction at ring 0 called by the guest OS. In some cases, when the host OS uses all the rings available, this trick cannot be used and the hypervisor must exploit different mechanisms.

4 Kinds of virtualization

4.1 Paravirtualization

The paravirtualization mechanism doesn't change the instructions level security of the host OS, but to allow the guest OSs also to run instructions at ring 0, some parts of the their kernel are modified so that they do the same thing in an alternate way. An advantage of this technique is that doesn't require hardware support at all.

4.2 Full virtualization

In this case only the hypervisor runs in ring 0, so it is the only that has the access to the hardware. The control OS runs in ring 1 and each privileged instructions of the guest operating systems are managed by the hypervisor in order to execute them safely.

4.3 Full virtualization with hardware support

A new ring -1 is added below all the other protection rings, this means that we can have standard unmodified kernels for the guests OS running at ring 0 and the hypervisor running at ring -1. The hypervisor can automatically handle all the translations needed between ring 0 and ring -1 that really acts on the hardware. This technology is supported by most recent processors both from Intel and AMD.

4.4 Process level virtualization

Process virtualization uses the concept of containers to insulates processes so that they cannot see outside other application running on the same hardware, they cannot communicate with other processes and they cannot even know that other processes exist outside of the container. Container management is much like virtual machine management with less detail to provide.

4.5 Hosted virtualization

This is the most common case, the hypervisor is of type 2 and runs within the OS (VMWare, VirtualBox). The OS and the hypervisor run in ring 0, the latter is made to cooperate with the kernel without causing any problem even if this may require small changes in the host OS kernel. The virtual machine usually runs in ring 1.