An Introduction to Virtualization

It's hot. Yet again.

Microsoft acquired Connectix Corporation, a provider of virtualization software for Windows and Macintosh based computing, in early 2003. In late 2003, EMC announced its plans to acquire VMware for $635 million. Shortly afterwards, VERITAS announced that it was acquiring an application virtualization company called Ejascent for $59 million. Sun and Hewlett-Packard have been working hard in recent times to improve their virtualization technologies. IBM has long been a pioneer in the area of virtual machines, and virtualization is an important part of IBM's many offerings. There has been a surge in academic research in this area lately. This *umbrella of technologies*, in its various connotations and offshoots, is hot, yet again.

> The purpose of this document can be informally stated as follows: *if you were to use virtualization in a an endeavor (research or otherwise), here are some things to look at*.

History

Christopher Strachey published a paper titled *Time Sharing in Large Fast Computers* in the International Conference on Information Processing at UNESCO, New York, in June, 1959. Later on, in 1974, he clarified in an email to Donald Knuth that:

" ... [my paper] was mainly about multi-programming (to avoid waiting for peripherals) although it did envisage this going on at the same time as a programmer who was debugging his program at a console. I did not envisage the sort of console system which is now so confusingly called time sharing.". Strachey admits, however, that "time sharing" as a phrase was very much in the air in the year 1960.

> The use of multi-programming for spooling can be ascribed to the Atlas computer in the early 1960s. The Atlas project was a joint effort between Manchester University and Ferranti Ltd. In addition to spooling, Atlas also pioneered *demand paging* and *supervisor calls* (that were referred to as "extracodes"). According to the designers (1961): *"... the Supervisor extracode routines (S.E.R.'s) formed the principal 'branches' of the supervisor program. They are activated either by interrupt routines or by extracode instructions occurring in an object program."* A "virtual machine" was used by the Atlas supervisor, and another was used to run user programs.

In the mid 1960s, the IBM Watson Research Center was home to the M44/44X Project, the goal being to evaluate the then emerging time sharing system concepts. The architecture was based on virtual machines: the main machine was an IBM 7044 (M44) and each virtual machine was an experimental image of the main machine (44X). The address space of a 44X was resident in the M44's memory hierarchy, implemented via virtual memory and multi-programming.

IBM had provided an IBM 704 computer, a series of upgrades (such as to the 709, 7090, and 7094), and access to some of its system engineers to MIT in the 1950s. It was on IBM machines that the Compatible Time Sharing System (CTSS) was developed at MIT. The supervisor program of CTSS handled console I/O, scheduling of foreground and background (offline-initiated) jobs, temporary storage and recovery of programs during scheduled swapping, monitor of disk I/O, etc. The supervisor had direct control of all trap interrupts.

Around the same time, IBM was building the 360 family of computers. MIT's Project MAC, founded in the fall of 1963, was a large and well-funded organization that later morphed into the MIT Laboratory for Computer Science. Project MAC's goals included the design and implementation of a better time sharing system based on ideas from CTSS. This research would lead to Multics, although IBM would lose the bid and General Electric's GE 645 would be used instead.

Regardless of this "loss", IBM has been perhaps the most important force in this area. A number of IBM-based virtual machine systems were developed: the CP-40 (developed for a modified version of IBM 360/40), the CP-67 (developed for the IBM 360/67), the famous VM/370, and many more. Typically, IBM's virtual machines were identical "copies" of the underlying hardware. A component called the virtual machine monitor (VMM) ran directly on "real" hardware. Multiple virtual machines could then be created via the VMM, and each instance could run its own operating system. IBM's VM offerings of today are very respected and robust computing platforms.

Old Problems

Robert P. Goldberg describes the then state of things in his 1974 paper titled *Survey of Virtual Machines Research*. He says: *"Virtual machine systems were originally developed to correct some of the shortcomings of the typical third generation architectures and multi-programming operating systems - e.g., OS/360."* As he points out, such systems had a dual-state hardware organization - a privileged and a non-privileged mode, something that's prevalent today as well. In privileged mode all instructions are available to software, whereas in non-privileged mode they are not. The OS provides a small resident program called the *privileged software nucleus* (analogous to the *kernel*). User programs could execute the non-privileged hardware instructions or make supervisory calls - e.g., SVC's - (analogous to *system calls*) to the privileged software nucleus in order to have *privileged* functions - e.g., I/O - performed on their behalf. While this works fine for many purposes, there are fundamental shortcomings with the approach. Consider a few:

- Only *one* "bare machine interface" is exposed. Therefore, only one kernel can be run. Anything, whether it be another kernel (belonging to the same or a different operating system), or an arbitrary program that *requires* to talk to the bare machine (such as a low-level testing, debugging, or diagnostic program), cannot be run alongside the booted kernel.
- One cannot perform any activity that would disrupt the running system (for example, upgrade, migration, system debugging, etc.) One also cannot run untrusted applications in a secure manner.
- One cannot easily provide the illusion of a hardware configuration that one does not have (multiple processors, arbitrary memory and storage configurations, etc.) to some software.

We shall shortly enumerate several more reasons for needing virtualization, before which let us clarify what we mean by the term.

A Loose Definition

Let us define "virtualization" in as all-encompassing a manner as possible for the purpose of this discussion: *virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.*

Note that this definition is rather loose, and includes concepts such as quality of service, which, even though being a separate field of study, is often used alongside virtualization. Often, such technologies come together in intricate ways to form interesting systems, one of whose properties is virtualization. In other words, the concept of virtualization is related to, or more appropriately *synergistic* with various paradigms. Consider the multi-programming paradigm: applications on *nix systems (actually almost all modern systems) run within a virtual machine model of some kind.

Since this document is an informal, non-pedantic overview of virtualization and how it is used, it is more appropriate not to strictly categorize the systems that

we discuss.

> Even though we defined it as such, the term "virtualization" is not always used to imply partitioning - breaking something down into multiple entities. Here is an example of its different (intuitively *opposite*) connotation: *you can take N disks, and make them appear as one (logical) disk through a virtualization layer.*
>
> Grid computing enables the "virtualization" (ad hoc provisioning, on-demand deployment, decentralized, etc.) of distributed computing: IT resources such as storage, bandwidth, CPU cycles, ...
>
> PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. PVM is widely used in distributed computing.
>
> Colloquially speaking, *"virtualization abstracts out things."*

Why Virtualization: A List of Reasons

Following are some (possibly overlapping) representative reasons for and benefits of virtualization:

- Virtual machines can be used to consolidate the workloads of several under-utilized servers to fewer machines, perhaps a single machine (server consolidation). Related benefits (perceived or real, but often cited by vendors) are savings on hardware, environmental costs, management, and administration of the server infrastructure.
- The need to run legacy applications is served well by virtual machines. A legacy application might simply not run on newer hardware and/or operating systems. Even if it does, if may under-utilize the server, so as above, it makes sense to consolidate several applications. This may be difficult without virtualization as such applications are usually not written to co-exist within a single execution environment (consider applications with hard-coded System V IPC keys, as a trivial example).
- Virtual machines can be used to provide secure, isolated sandboxes for running untrusted applications. You could even create such an execution environment dynamically - on the fly - as you download something from the Internet and run it. You can think of creative schemes, such as those involving *address obfuscation*. Virtualization *is* an important concept in building secure computing platforms.
- Virtual machines can be used to create operating systems, or execution environments with resource limits, and given the right schedulers, resource guarantees. Partitioning usually goes hand-in-hand with quality of service in the creation of QoS-enabled operating systems.
- Virtual machines can provide the illusion of hardware, or hardware configuration that you do not have (such as SCSI devices, multiple processors, ...) Virtualization can also be used to simulate networks of independent computers.
- Virtual machines can be used to run multiple operating systems simultaneously: different versions, or even entirely different systems, which can be on hot standby. Some such systems may be hard or impossible to run on newer real hardware.
- Virtual machines allow for powerful debugging and performance monitoring. You can put such tools in the virtual machine monitor, for example. Operating systems can be debugged without losing productivity, or setting up more complicated debugging scenarios.
- Virtual machines can isolate what they run, so they provide fault and error containment. You can inject faults proactively into software to study its subsequent behavior.
- Virtual machines make software easier to migrate, thus aiding

application and system mobility.

- You can treat application suites as *appliances* by "packaging" and running each in a virtual machine.
- Virtual machines are great tools for research and academic experiments. Since they provide isolation, they are safer to work with. They encapsulate the entire state of a running system: you can save the state, examine it, modify it, reload it, and so on. The state also provides an abstraction of the workload being run.
- Virtualization can enable existing operating systems to run on shared memory multiprocessors.
- Virtual machines can be used to create arbitrary test scenarios, and can lead to some very imaginative, effective quality assurance.
- Virtualization can be used to retrofit new features in existing operating systems without "too much" work.
- Virtualization can make tasks such as system migration, backup, and recovery easier and more manageable.
- Virtualization can be an effective means of providing binary compatibility.
- Virtualization on commodity hardware has been popular in co-located hosting. Many of the above benefits make such hosting secure, cost-effective, and appealing in general.
- Virtualization is fun.
- Plenty of other reasons ...

Variations

Generically speaking, in order to virtualize, you would use a layer of software that provides the illusion of a "real" machine to multiple instances of "virtual machines". This layer is traditionally called the Virtual Machine Monitor (VMM).

There are many (often intertwined) high-level ways to think about a virtualization system's architecture. Consider some scenarios:

A VMM could itself run directly on the real hardware - without requiring a "host" operating system. In this case, the VMM *is* the (minimal) OS.

A VMM could be hosted, and would run entirely as an application on top of a host operating system. It would use the host OS API to do *everything*. Furthermore, depending on whether the host and the virtual machine's architectures are identical or not, instruction set emulation may be involved.

From the point of view of how (and where) instructions get executed: you can handle all instructions that execute on a virtual machine in software; you can execute *most* of the instructions (maybe even some privileged instructions) *directly* on the real processor, with certain instructions handled in software; you can handle *all* privileged instructions in software ...

A different approach, with rather different goals, is that of *complete* machine simulation. SimOS and Simics, as discussed later, are examples of this approach.

Although architectures have been designed explicitly with virtualization in mind, a typical hardware platform, and a typical operating system, both are not very conducive to virtualization.

As mentioned above, many architectures have privileged and non-privileged instructions. Assuming the programs you want to run on the various virtual machines on a system are all *native* to the architecture (in other words, it would not necessitate emulation of the instruction set). Thus, the virtual machine can be run in non-privileged mode. One would imagine that non-privileged instructions can be *directly* executed (without involving the VMM), and since the privileged instructions would cause a trap (since they are being executed in non-privileged mode), they can be "caught" by the VMM, and appropriate action can be taken (they can be simulated by the VMM in software, say). Problems arise from the fact that there may be instructions that are non-privileged, but their behavior depends on the processor mode - these

instructions are *sensitive*, but they do not cause traps.

> One of the most popular architectures, IA-32, is not virtualization friendly. The analysis in a paper titled *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor* reports at least seventeen instructions on the Pentium that make it "non-virtualizable". IA-32's privileged instructions cause a General Protection Exception when executed in non-privileged mode. Instructions like `STR` can be problematic: `STR` can be executed at any privilege level, but it tells you the security state of the machine (the value it retrieves has the Requestor Privilege Level, or `RPL`).
>
> The IA-32 TLB (unified code and data in 386/486, separate in the Pentium) is hardware managed. In contrast, architectures such as Alpha, MIPS, PA-RISC, and SPARC use software managed TLB's, which are easier to virtualize than hardware page tables. An IA-32 TLB entry cannot be *tagged*, say with an address space identifier (ASID), which would make it easier (and less expensive, in terms of TLB flushes) to manage the address spaces of the VMM and its virtual machine kernels.

Architecture amicability aside, there are various other problems to solve.

When a process on a guest system running on a (hosted) virtual machine invokes a system call, it should not be handled by the host. The host should notify the guest operating system. One solution is for the virtual machine monitor to use `ptrace()` to trace process execution and identify system call entry. The VMM can then *nullify* the system call (say, by "converting" it to `getpid()`, or to an invalid system call), which is executed by the host. Upon system call exit, the VMM notifies the guest system kernel (through a signal, say), which can take appropriate action.

A similar situation exists in case of page faults.

> Mach is capable of running Unix as an application program. To achieve this, Mach uses a Unix Server for providing BSD system services/ resources, and a *Transparent System Call Emulation Library* that executes within the address space of a Unix task. Now, Mach supports system call redirection - you can have a certain set of system calls to be handled by user-space code within the calling task. When a user-level Unix task issues a system call, the Transparent Emulation Library intercepts the call (using the redirection facility), and transforms it into a remote procedure call to the Unix Server (not always though - sometimes the Emulation Library can handle the call).

When a typical operating system kernel running on real hardware has nothing to do, it runs its idle thread, or loop. When the same kernel runs on a virtual machine, this behavior is undesirable, because the virtual machine is *wasting* its processor time. The virtual machine could have a mechanism to suspend itself, instead of running the idle loop. For example, the Denali Isolation Kernel uses a purely virtual instruction (`idle-with-timeout`) for this purpose.

Along similar lines, the virtual machine monitor would not know when a memory page is no longer being actively used by a virtual machine.

Depending on how much, and how virtualization is done, there will be more such issues. Figuring out yet more optimal ways to virtualize (particularly in the face of hard-to-virtualize hardware) is an active area of research.

Emulation and Simulation

A virtualization framework may make use of emulation or simulation, perhaps because the guest and host architectures are different, or even otherwise.

In the context of software, an emulator reproduces the behavior of one system on another. It executes, or strives to execute, the same programs as the "original" system, and produces the same results for the same input. It is important that the user of an emulator is not supposed to care *how*. Software emulators abound for old and new hardware architectures, video game

consoles, etc.

In the context of computing, a simulation is an imitation of some real system. A simulator can be informally thought of as an "accurate emulator". The ARMn is a multiprocessor *cycle-accurate* simulator that can simulate a cluster of ARM processor cores connected by custom communication schemes. The VCS Verilog Simulator from Synopsys can do gate-level ASIC simulation, useful to semiconductor people. There is exists an instruction-accurate simulator for the picoJava processor core, which is essentially an accurate (reasonably) software model of the real thing, and so on.

A related family is that of in-circuit emulators: a combination of hardware and software that lets you run code on actual hardware while providing flexible and powerful debugging facilities, etc.

> Today the Turing machine has become the accepted formalization of an effective procedure [Hopcroft and Ullman, *Introduction to Automata Theory, Languages, and Computation*]. Alonzo Church hypothesized that the Turing machine model is equivalent to our intuitive notion of a computer. A Universal Turing Machine can compute any function that any Turing machine can compute (Turing proved this). Often, the Church-Turing thesis is (mis)understood to imply that *the Universal Turing Machine can simulate the behavior of any machine*. Nevertheless, given that an arbitrary computer is equivalent to *some* Turing machine, it follows that *all computers can simulate each other*.

Examples

We have seen that in the original, traditional sense, virtualization provides multiple execution environments (virtual machines), each of which is identical to the underlying computer. Each virtual machine looks like a "real" machine to its user, whereas in reality, it is an isolated (from others) environment running on the *really* real machine under the supervision of a Virtual Machine Monitor (VMM). Recent years have introduced several new connotations for the phrase "virtual machine" (as some of the examples will indicate). This section provides brief overviews of several frameworks (methodologies, projects, products, concepts) related directly or indirectly to virtualization.

ABI/API Emulation

Rather than creating virtual machines to run entire operating systems, API emulation can be used to create execution environments for running alien programs on a platform.

Sun used WABI (Windows Application Binary Interface) to make Solaris more appealing to those needing Windows applications. The WABI software sits between an application and the operating system, intercepts the applications Windows calls, and translates them to "equivalent" Unix calls. On x86, the guest instructions were run directly on the processor, while they were emulated and/or binary translated on SPARC. WABI can also use an optional DOS emulator to run DOS applications.

> Sun later had the SunPC software that emulates a PC hardware environment on Solaris (SPARC). The software-only SunPC emulated a 286, but you could install a SunPC Accelerator Card (that had a coprocessor such as a 133 MHz AMD 5x86), and run Windows 95, although one session at a time since real hardware was used.
>
> A later product is the SunPCi III, a coprocessor card with features such as: mobile 1.4 GHz AMD Athlon XP 1600+ processor, up to 1 GB PC2100 RAM, AGP 8x equivalent 24-bit graphics, 10/100Base-T Ethernet, up to 3 USB 2.0 ports (two on an optional daughterboard), and a FireWire port (on the optional daughterboard). The A: drive is the physical floppy drive, while C: and D: are emulated via files on Solaris. Other drives can be mapped to local or networked file systems, or the local optical drive.

Lxrun is software for executing Linux `a.out` and ELF binaries (x86 only) on x86 Unix systems such as SCO OpenServer, SCO UnixWare, and later, Solaris. This is achieved by "remapping" Linux system calls on the fly. You need the Linux

shared libraries that the application requires, as well as the Linux dynamic loader. Lxrun is thus a *system call emulator*. There are various caveats as to what kind of applications will not run, etc.

Newer versions of the real-time LynxOS have Linux ABI compatibility.

Similarly, FreeBSD provides binary compatibility with a few Unix like systems, including Linux. Over time, FreeBSD has evolved to actually include a process file system (`linprocfs`) that emulates a subset of Linux's `procfs`.

Wine is software that lets you run Windows *applications* on Linux, FreeBSD, and Solaris. Wine is x86 only, and does *not* emulate a processor.

It is perhaps not a widely known fact outside the Solaris community that Microsoft had versions of Internet Explorer and Outlook Express for Solaris (SPARC). This was achieved not by porting them to Solaris, but by using API emulation. Mainsoft, the software company behind that effort, now has a product called Visual MainWin that allows for applications developed on Windows using Visual Studio to be run on Solaris, Linux, HP-UX, and AIX. It *recompiles* the applications from source on the deployment platform, using the latter's compilers.

There are far too many other examples to be enumerated here.

Bochs

Bochs is an open source x86 emulator written in C++. It is a *user-space* emulator, and emulates the x86 processor, several I/O devices, and a custom BIOS. Bochs is highly portable, and rather slow (not surprising since it *emulates* every instruction and I/O devices): the primary author of Bochs reports 1.5 MIPS on a 400 MHz Pentium II.

Nevertheless, Bochs is extremely flexible and customizable.

Chorus

The Chorus system's kernel provides a low-level framework on top of which distributed operating systems could be implemented. For example, System V Unix was implemented on Chorus this way, by making use of System V specific emulation assist code in the Chorus kernel.

chroot()

Many frameworks, particularly those targeted at hosting providers, make use of `chroot()` for filesystem sandboxing, either within the kernel, or in user-space. FreeBSD's jail uses `chroot()`. Virtfs is a relatively simpler `chroot()` based solution for Linux.

Denali

The Denali *isolation kernel* is an operating system, essentially an IA-32 virtual machine monitor , that allows for untrusted services to be run in isolated (protected) "domains". Denali does *not* aim to allow for unmodified operating systems to run on it: an operating system *must* be ported to the Denali architecture (for example, the processor architecture as presented by Denali is not x86, but a version modified for virtualizability and scalability).

Dis

Dis is the virtual machine designed for the Inferno operating system. Although Dis is conceptually similar in Java to many respects, the inventors describe key differences (such as an instruction set that matches existing processor architectures more closely, a less lazy garbage collection, etc.) in *The design of the Inferno virtual machine*.

Disco

Disco was the outcome of a Stanford University project with the goals of extending modern operating systems to run efficiently on large-scale shared

memory multiprocessors without a large implementation effort. Disco is essentially a VMM, implemented as a multi-threaded shared memory program, sitting atop the hardware and allowing multiple virtual machines. It virtualizes all resources of the underlying machine. An instance of a virtual machine has a MIPS R10K processor, main memory with contiguous physical addresses starting at zero, a specified set of devices such as disk, network interfaces, periodic interrupt timers, clock, and a console. The execution of a virtual processor is emulated via direct execution on the real processor. The MIPS TLB can be reloaded by software, and each TLB entry is tagged with an address space identifier, so that the TLB does not have to be flushed on a MMU context switch. Disco adds special device drivers (such as for UART, SCSI, Ethernet, etc.) into the operating system, and intercepts all device accesses from a virtual machine. Disco was used to run Silicon Graphics IRIX 5.3.

Ensim

Ensim has done a lot of pioneering work in the area of virtualizing operating systems on commodity hardware. Ensim's Virtual Private Server (VPS) technology allows you to securely partition an operating system in software, with quality of service, complete isolation, and manageability. There exist versions for Solaris, Linux, and Windows.

FreeBSD

The FreeBSD "jail" mechanism allows you to create an isolated existing environment via software means. Jail uses `chroot(2)`, and each jail has its own "root". Processes in a jail do not have access to or visibility of files, processes, or network services in *other* jails. A jail can be restricted to a single IP address.

The jail feature is implemented by making various components of the FreeBSD kernel "jail aware", such as the `pty` driver, the system call API, the TCP/IP stack, and so on.

> In 1998-99, I worked on the ECLIPSE operating system at Bell Labs. There was a great interest in Quality of Service then. ECLIPSE was derived from FreeBSD and included support both for quality of service (even for legacy applications), and a layer to manage it. ECLIPSE had fair-share schedulers for CPU, network, and disk. Protocols like NFS, WWW, and FTP were QoS aware. A pseudo filesystem (the *reservation* filesystem) was used to provide the user-level API for managing resources.
>
> While retrofitting QoS in an existing operating system (such as FreeBSD) is a good idea, one cannot deny that it cannot be perfect due to the architecture of the existing system. Typically you associate resource guarantees (or weights) with an execution context (say, a "domain", in which processes can run). One now needs to *tag* processes that belong to this domain, which is great except there is plenty of activity in the kernel that doesn't traditionally have access to process context. Modifying data structures to propagate the tags is unclean, and may break compatibility (say, because you changed the size of the `proc` structure). Thus, unless you design an operating system with QoS in mind, interactions in the system are complex enough that it is extremely difficult to charge all activity to its rightful owner.
>
> In light of the problem described above, ECLIPSE also included Signaled Receiver Processing (SRP) to alleviate a certain set of issues: protocol processing of received packets in BSD Unix is interrupt-driven and may cause scheduling anomalies that are unacceptable in systems that provide QoS guarantees. SRP is an alternate mechanism that generates a signal to the receiving process when a packet arrives. The default action of this signal is to perform protocol processing asynchronously. However, a receiving process may catch, block or ignore the signal and defer protocol processing until a subsequent receive call. In any case, protocol processing occurs in the context of the receiving process and is correctly charged. Therefore, SRP allows the system to enforce and honor QoS guarantees. Note that this is not the same as Lazy Receiver Processing (LRP).

Hive

Hive is an internally distributed system consisting of multiple independent kernels, or *cells*. The idea is to improve reliability by containing faults within a cell, thus not affecting processes running on other cells. Each memory page in hive has a small write permission bitmap, which allows the system to discard corrupt page upon fault detection.

HP-UX Virtual Partitions

In Hewlett-Packard's own words: *"Hewlett-Packard has created a family of flexible, powerful, and far-reaching partitioning solutions - the HP Partitioning Continuum for Always-On infrastructures. The solutions provide hard partitions, virtual partitions, and resource partitions ..."*

The HP Virtual Partitions (VPAR's) provide operating system and application (including name space) isolation. A VPAR runs its own copy of HP-UX (potentially different versions), and can be dynamically created, with a specific set of resources assigned to it. Within a VPAR, you can further create *resource* partitions.

There is a virtual machine monitor (the vPar Monitor) that sits on top of the hardware (it is booted on the real hardware instead of HP-UX) and assigns ownership of hardware resources to virtual machine instances (the vPar's). Note that a vPar accesses physical memory and I/O hardware *directly*, without involving the Monitor. A vPar is not, however, aware of the hardware resources that are not assigned to it.

Linux/RK

Linux/RK is an Linux-based *resource kernel* implementation. The effort focuses on incorporating quality of service (with respect to CPU, physical memory pages, network bandwidth, and disk I/O) in a portable manner.

LPAR

IBM's Logical Partitioning (LPAR) allows you to run multiple, independent operating system images of AIX and Linux on a single server (that supports such partitioning, such as the pSeries family). The minimum resources needed for a pSeries partition are: one processor, 256 MB memory, an I/O slot and its attached devices. The AIX Workload Manager (WLM) is used for resource management.

IBM introduced Dynamic Logical Partitioning (DLPAR) in AIX 5L Version 5.2. DLPAR allows you to dynamically add and remove resources from active partitions.

Other server families, such as the iSeries, and other IBM operating systems, such as OS/400, also support logical partitioning. There is a primary OS/400 partition, that loads a hypervisor (known as "the Hypervisor"), which provides partition control, mediation, and isolation. You can then have further OS/400 partitions, as well as Linux partitions.

Note that the POWER4 architecture has features that help in virtualization (such as a special Hypervisor mode in the processor, the ability to include an address offset when using non-virtual memory addressing, support for multiple global interrupt queues in the interrupt controller, and so on. The firmware of these machines is also specialized for virtualization.

> IBM has a number of server offerings. The zSeries is IBM's mainframe range that can run operating systems such as z/OS, z/OS.e, z/VM, VSE/ESA, a transaction processing OS (TPF), and Linux. The iSeries are midrange servers running OS/400 and Linux. The pSeries exist in various ranges, and run AIX and Linux. There are other server solutions for clustering, storage, etc.

Mac-on-Linux

Mac-on-Linux, or simply MOL, is a virtual machine implementation that runs under Linux on most PowerPC hardware, and allows you to run Mac OS (7.5.2 to 9.2.2), Mac OS X, and Linux. Most of MOL's virtualization functionality is implemented as a kernel module. A user process takes care of I/O, etc. There's

even an (very limited) Open Firmware implementation within MOL.

MAE

The Macintosh Application Environment (MAE) was an X application that ran on RISC machines (such as SPARCstation running SunOS and HP 9000/700 running HP-UX) and provided a virtual Macintosh environment. MAE emulated the Motorola 68LC040 processor, with native execution whenever possible for performance. MAE was based on System 7.x.

Microsoft Virtual Server

Microsoft has had its share of virtualization in the past. Windows NT had several subsystems, or execution environments, such as the virtual DOS machine (VDM), the Windows on Win32 (WOW) virtual machine for 16-bit Windows, the OS/2 subsystem, the POSIX subsystem, and the Win32 subsystem. Note that while the OS/2, POSIX, and Win32 subsystems are server processes, DOS and Win16 run within the context of a virtual machine process. They all are dependent on the NT executive for basic operating system mechanisms though.

> The VDM was essentially a virtual DOS (derived from MS-DOS 5.0 code base) running on a virtual x86. On x86, a trap handler was present to handle privileged instructions. Windows NT also ran on MIPS, so an x86 emulator had to be there in the MIPS version.

Similarly, Windows 95 used virtual machines to run older (Windows 3.x and DOS) applications. There was a System virtual machine that ran the kernel, GDI, etc. The System virtual machine had an address space shared by all 16-bit Windows programs, and a separate address space for each 32-bit Windows program.

Microsoft has included virtualization as a key component of its server offerings for the Enterprise with the acquisition of Connectix in early 2003. As with Virtual PC, the idea is to run multiple operating systems simultaneously on one machine.

Microsoft, and many enterprise software vendors, have also been making their *applications* virtualized. Microsoft's SQL Server 2000 has multiple instance capability. Microsoft's Exchange Server, File/Print Servers, IIS Server, Terminal Server, etc. also don't really need virtualization support in the operating system. There are pros and cons of virtualizing within an application, but there are scenarios in which anything stronger, or lower level, is overkill, or not optimal.

Nemesis

Nemesis is an operating system designed at the University of Cambridge Computer Laboratory to support quality of service. The Nemesis kernel is extremely small and lightweight, and most of the operating system code executes in the application process itself. The kernel has a scheduler and some other code for low-level CPU management. There is a *single* global page table (a single address space), although per-process memory protection is still there. Since the kernel performs much less work on behalf of an application, there is much less scope for the "wrong" process being charged for somebody else's work (often referred to as *QoS crosstalk*).

Related ideas can be found in Bell Labs' Pebble Operating System, the V++ Cache Kernel, and the MIT Exokernel Operating System. Yet another example is that of the "Separation Kernel" that John Rushby talks about in his 1981 paper titled *Design and Verification of Secure Systems*. Rushby's paper discusses the specific case of the "Secure User Environment" (SUE): a minimally small and very simple kernel, providing a fixed, small number of *regimes*, each of which runs a fixed, small program.

Plex86

Plex86 in its current life aims to provide a lightweight x86 virtual machine for running Linux. It does *not* handle instructions that cannot be virtualized (thus, it does not do any binary rewriting or code scanning). It also does *not* model

any I/O devices.

The Linux kernel needs minor modifications in order to run under Plex86.

Programming Language Virtual Machines

Programming languages are often implemented using virtual machines. Benefits of doing this include isolation (the virtual machine is a sandbox) and portability. The UCSD P-System was very popular in the 70s and the early 80s. It was a virtual machine running *p-code* (akin to bytecode), with UCSD PASCAL being the most popular programming language. The operating system itself was written in PASCAL.

The Java Virtual Machine (JVM) is another well known virtual machine. The JVM is an abstract computer: there is a Java VM *specification* that describes the "machine" (in terms of things such as a register set, a stack, a heap that's garbage collected, a method area, an instruction set, etc.) A JVM *implementation* for a particular platform (such as x86/Linux, x86/Windows, SPARC/Solaris, and so on) represents, among other things, a software implementation of the above specification. Note that it is also possible to implement the JVM in microcode, or even directly in silicon. The picoJava, for example, is a Java processor core. You can compile a Java program on any platform X and run it on any platform Y, given X and Y support JVM implementations. Unless Y is a "Java processor", its instruction set would be different from the (platform-independent) *bytecode* produced by the Java compiler. A JVM could *interpret* the bytecode one (Java) instruction at a time, or use JIT (Just-In-Time), a JVM-integrated optimization (it usually *is* faster, but not always) that takes the bytecode and *compiles* it into *native* code for the machine it is running on.

Note that the Java virtual machine doesn't really care about the Java *programming language*: it only knows the format of the `class` file, that contains JVM instructions (bytecodes), a symbol table, etc.

The JVM is not a *multi-user* virtual machine, although there have been research efforts to re-architect the JVM for safe multi-tasking with multi-user support.

Microsoft's .NET CLI is another example, and so is Parrot. There are many more popular and/or successful programming language virtual machines.

QLinux

QLinux is an operating system that extends Linux to support quality of service. It was a result of joint work by the Universities of Massachusetts (Amherst) and Texas (Austin). QLinux includes a hierarchical start-time fair queueing (H-SFQ) CPU scheduler, an H-SFQ packet scheduler for network, the Cello disk scheduler, and Lazy Receiver Processing to incorporate fairer accounting of protocol processing overheads.

QLinux is very similar to ECLIPSE: the latter is based on FreeBSD, uses different scheduling algorithms, and uses a pseudo filesystem as a management interface.

Shade

Shade is a virtual machine that emulates a target system's ABI by *dynamically* cross-compiling the target machine code to run on the host system. Shade is also a program profiler: it can be used to (programmatically) trace/profile the programs that it executes.

Shade was a result of joint research by Sun Microsystems and University of Washington, Seattle.

SILK

Scout is a modular operating system targeted for small network appliances. It is communication-oriented, and incorporates several well-known network architecture improvements under one roof. It supports assigning of resources/ limits to multiple data flows, that can be explicitly scheduled (so as to provide QoS guarantees). Incoming packets are "early demultiplexed" to these flow queues, and they are also dropped early if the queues become full. The single

abstraction that captures these ideas is termed the *path* (a single TCP connection encapsulating a flow of data is a path, for example). SILK stands for Scout in the Linux Kernel. Scout exists as a Linux kernel module in SILK. It includes its own CPU scheduler and threads package (the Linux scheduler is still there). SILK runs as a highest priority real-time kernel task (which is different from LRP).

Simics

Simics is a platform for *complete system simulation*. It began life as `gsim` in 1991, which itself was based on the `g88` which was written by Robert Bedicheck while at Tektronix. In the words of its project members: *"[Simics] attempts to strike a balance between accuracy and performance. That is, it is sufficiently abstract to achieve tolerable performance level with, at the same time, sufficient functional accuracy to run commercial workloads and sufficient timing accuracy to interface to detailed hardware models."* Simics model various processors and devices accurately enough to be able to run unmodified operating systems.

Simics is available for `sparc-sun-solaris`, `x86-any-linux`, and `x86-microsoft-windows`. Simics can run various Linux ports (ARM, MIPS, PowerPC, SPARC, x86, AMD64, IA64) on top of Simics hardware simulations, as well as VxWorks (PowerPC), OSE (PowerPC), Solaris (SPARC), and various Windows (x86, AMD64).

SimOS

SimOS is a *complete machine simulator* developed at Stanford. It is capable of modeling complete computer systems (CPU, caches, multiprocessor memory buses, network devices, disk drives, other I/O devices, ...), although it allows you to control the level of simulation detail. Note that it is very similar to Simics. The SimOS project started in 1992 with the simulation of the Sprite system on SPARC hardware. The next implementation of SimOS simulated the (MIPS based) hardware of an SGI machine in enough detail to support IRIX. This implementation ran on an SGI machine as a host computer, allowing a direct-execution mode.

SimOS has also been extended to model a Digital Alpha processor, on which a port of Digital Unix can be run, and to the PowerPC, on which AIX can be run.

SimOS can optionally use Embra, a processor simulator that uses dynamic binary translation to generate code sequences that simulate the given workload.

> Transitive is a company offering products based on dynamic binary translation. Quoted verbatim from its web site: *"Transitive was founded in 2000 to create a software solution that enables applications written for one processor based system to be easily transported to another processor based system. The software solution, known as dynamic binary translation, is designed to allow system vendors the maximum flexibility in moving software applications to the optimum hardware platform."*
>
> Transitive says it has working solutions based on MIPS, x86, ARM/Xscale, PowerPC, and Itanium.

SimOS runs on `mips-sgi-irix-5.x`, `mips-sgi-irix-6.x`, `alpha-dec-digitalunix`, `sparc-sun-solaris`, and `[5|6]86-any-linux`.

> Complete system simulators like Simics and SimOS have different primary uses than, say, something you would use to run an additional operating system for productivity. The variable level of detail these simulators can reproduce can be used to design and develop processors and devices, debugging and developing operating systems (since the sequence of events leading to an error can be captured in greater detail), testing for reliability and fault tolerance, studying memory behavior (since various kinds of memory and memory spaces can be simulated) etc.

Solaris

Sun introduced static partitioning in 1996 on its E10K family of servers. The partitions, or domains, were defined by a physical subset of resources - such as a system board with some processors, memory, and I/O buses. A domain could span multiple boards, but could not be smaller than a board. Each domain ran its own copy of Solaris. In 1999, Sun made this partitioning "dynamic" (known as Dynamic System Domains) in the sense that resources could be moved from one domain to another.

By the year 2002, Sun had also introduced Solaris Containers: execution environments with limits on resource consumption, existing within a *single copy* of Solaris. Sun has been improving and adding functionality to its Resource Manager (SRM) product, which was integrated with the operating system beginning with Solaris 9. SRM is used to do intra-domain management of resources such as CPU usage, virtual memory, maximum number of processes, maximum logins, connect time, disk space, etc.

The newest Sun reincarnation of these concepts is (tentatively) called "Zones": a feature in the upcoming Solaris 10. According to Sun, the concept is derived from the BSD "jail" concept: a Zone (also known as a "trusted container") is an isolated and secure execution environment that appears as a "real machine" to applications. There is only one copy of the Solaris kernel.

> While working for Ensim Corporation, I started the Solaris Virtual Private Server Project in late 1999. By the end of 2000, we had a virtualized version of Solaris, rather similar to the Solaris 10 Zones feature. There is only one instance of the kernel, but the operating system is divided into multiple isolated execution environments via a thin software layer, implemented mostly as a set of kernel modules. Each instance is visible as a "normal" operating system to applications within it, and is capable of running arbitrary complicated existing applications unmodified (such as the Oracle database server), with quality of service, and in complete isolation from applications on other instances.
>
> Each instance can be managed (administered, configured, rebooted, shutdown, etc.) independently of others.
>
> Specific virtualization components include (but are not limited to):
>
> - Virtualized system calls
> - Virtualized uid 0 (each instance has its own "root" user)
> - Fair share network scheduler
> - Per-virtual OS resource limits on memory, CPU and link
> - Virtual sockets and TLI (including port space)
> - Virtual NFS
> - Virtual IP address space
> - Virtual disk driver and enhanced VFS (each instance sees its own physical disk that can be resized dynamically, on which it can create partitions)
> - Virtual System V IPC layer (each instance gets its own IPC namespace)
> - Virtual `/dev/kmem` (each instance can access `/dev/kmem` appropriately without compromising other instances or the system)
> - Virtual `/proc` file system (each instance gets its own `/proc` with only its processes showing up)
> - Virtual syslog facility
> - Virtual device file system
> - Per-instance `init`
> - Overall system management layer
>
> Note that this was product quality software and all work was done

without ever having seen the source code for Solaris.

Sphera

Sphera is a hosting automation and management software company. One of their products (now marketed as an underlying technology) is VDS, or Virtual Dedicated Server, that partitions a physical machine into multiple execution environments in software.

SWsoft

SWsoft announced its acquisition of Plesk (maker of Plesk Server Administrator) and Yippi-Yeah! E-Business (maker of Confixx, a control panel popular in Europe). SWsoft's own product, Virtuozzo, allows you to create Virtual Private Servers (or VPS, a terminology originally coined by Ensim). Virtuozzo instances can be dynamically partitioned, and have quality of service guarantees.

TCP/IP Stack Virtualization

It may be worthwhile in certain scenarios to virtualize just the network stack, rather than an entire system. Isolated multiple network stacks (each with its own port space, routing table, packet filters, parameters, etc.) could be provided, either within the kernel, or running as user processes. Each stack could be given resource limits or guarantees.

This approach has been used in academic projects as well as by virtualization companies in their software.

User-Mode Linux

User-Mode Linux, or simply UML, is a *port* of the Linux kernel to the abstract `um` architecture. In other words, UML is the Linux kernel ported to run on *itself*, that is, the system call interface. UML runs on Linux as a set of Linux user processes, which run *normally* until they trap to the kernel.

UML originally ran in what is now referred to as the `tt` (trace thread) mode. In this mode, a special trace thread `ptrace`s UML threads, gets notified upon system call entry/exit, nullfies the original call (say, to `getpid()`), and notifies the UML kernel to execute the intended system call. Since the UML kernel and its processes both are in the "real" user space, the processes can read from and write to the kernel's memory. UML makes the relevant memory read-only temporarily, which hampers performance greatly. Modifications to the Linux kernel exist (the `skas` mode, for "Separate Kernel Address Space") that address many of these issues.

You can even compile a version of UML that can be nested inside another UML.

UMLinux

UMLinux is a framework for evaluating the behavior of networked Linux machines in the presence of faults. The faults themselves are injected via software in various locations such as the memory, CPU registers, block devices, and network interfaces. UMLinux is similar to User-Mode Linux (UML), but since the emphasis was on studying dependability behavior, UMLinux had memory protection of the user mode kernel (which UML did not, initially). Furthermore, UMLinux (the virtual machine, the "guest" kernel, and all the guest processes) is implemented as a *single* process on the host system.

Virtual PC

Microsoft acquired Connectix, the maker of Virtual PC, in early 2003. Connectix was founded in 1988. The Virtual PC product was introduced in 1997, and has been the only viable x86 virtual machine solution for the Macintosh. The Windows product (Virtual PC for Windows) was introduced later. There is a version even for OS/2.

While the Macintosh version uses an optimized CPU emulator, Virtual PC for Windows exploits the fact that host and the guest have the same architecture:

there is a Virtual Machine Monitor (VMM), or Hypervisor, that runs directly on the underlying hardware alongside the host operating system. I/O (such as disk and network) is handled in the user space via the host operating system. Certain devices are entirely simulated in software (the BIOS, PIC, DMA controller, IDE/ATA controller, real-time clock, buses, the keyboard, I/O, and memory controllers, programmable timers, etc.) Several other devices are partially implemented in software, and rely on their real counterparts (input devices such as the keyboard, mouse, joystick, etc., video controller, floppy drive, network interface, audio hardware, optical drive, hard disk drive, etc.)

VMware

VMware, recently acquired by EMC, was founded in 1998. Its first product was VMware Workstation (1999). The GSX Server and ESX Server products were introduced in 2001.

VMware Workstation (as well as the GSX Server) has a *hosted* architecture: it needs a host operating system (such as Windows or Linux). In order to optimize the complex mix of performance, portability, ease of implementation, etc., the product acts as both a virtual machine monitor (talking directly to the hardware), and as an application that runs on top of the host operating system. The latter frees the VMM from having to deal with the large number of devices available on the PCs (otherwise the VMM would have to include device drivers for supported devices).

VMWare Workstation's hosted architecture includes the following components: a user-level application (**VMApp**), a device driver (**VMDriver**) for the host system, and a virtual machine monitor (**VMM**) that is created by VMDriver as it loads. Thereafter, an execution context can be either native (that is, the host's), or virtual (that is, belonging to a virtual machine). The VMDriver is responsible for switching this context. I/O initiated by a guest system is trapped the the VMM and forwarded to the VMApp, which executes in the host's context and performs the I/O using "regular" system calls. VMware uses numerous optimizations that reduce various virtualization overheads.

GSX Server is also hosted, but is targeted for server deployments and server applications.

VMware ESX Server enables a physical computer to be available as a pool of secure virtual servers, on which operating systems can be run. This is an example of dynamic, logical partitioning. Moreover, ESX Server does not need a host operating system (like VMware workstation) - it runs directly on hardware (in that sense, it *is* the host operating system). ESX server was inspired by work on Disco and Cellular Disco, which virtualized shared memory multiprocessor servers to run multiple instances of IRIX. As mentioned earlier, the IA-32 architecture is not naturally virtualizable. Certain "sensitive" instructions must be handled by the VMM, and cannot be simply executed in non-privileged mode because they don't cause a General Protection exception. ESX Server solves this problem by dynamically rewriting portions of an operating system kernel's code to insert traps at appropriate places - in order to catch such sensitive instructions. ESX Server can run multiple virtual CPUs per physical CPU. Multiple physical network interface cards can be logically grouped into a single, high-capacity, virtual network device.

> Since virtualization-unfriendliness of IA-32 is a long standing issue, many approaches have been used to address it. Scanning code dynamically and inserting an illegal instruction before each instruction of interest is one option (which would then cause traps). You can also replace such instructions with subroutine calls.
>
> Almost all common x86 operating systems do not use all four privilege modes provided by IA-32, which has been exploited for schemes to protect a guest operating system kernel from its user level processes.

z/VM

z/VM, a multiple-access operating system that implements IBM virtualization technology, is the successor to IBM's VM/ESA operating system. z/VM can support multiple guest operating systems (there may be version, architecture, or other constraints), such as Linux, OS/390, TPF, VSE/ESA, z/OS, and z/VM itself. z/VM includes comprehensive system management API's for managing virtual images.

The real machine's resources are managed by the z/VM Control Program (CP), that also provides the multiple virtual machines. A virtual machine can be defined by its architecture (ESA, XA, and XC, that refer to specific IBM architectures), and its storage configuration (one of V=R, V=F, and V=V, refers to how the virtual machine's guest real storage is related to the host real storage).

Others

As mentioned in the beginning, the overview presented by this document is not strictly limited to virtualization. There are numerous other systems not listed above that could be discussed in the context of this document. It would be impractical, if not impossible, to cover them all. *Some* systems not discussed above include:

- Cellular IRIX
- Flask, Fluke, the OSKit (the Flux Research Group at University of Utah)
- Hurricane
- L4
- Mach
- Palladium (a project at SUNYSB, *not* the Trusted Computing architecture)
- QEMU CPU Emulator
- SPIN Modula-3 Operating System
- twoOStwo
- VINO
- VServer (Linux)
- Xen

References

TBD, even though this is the most important section of this document!