ghaff@
illuminata.com

The Future of the Operating System?

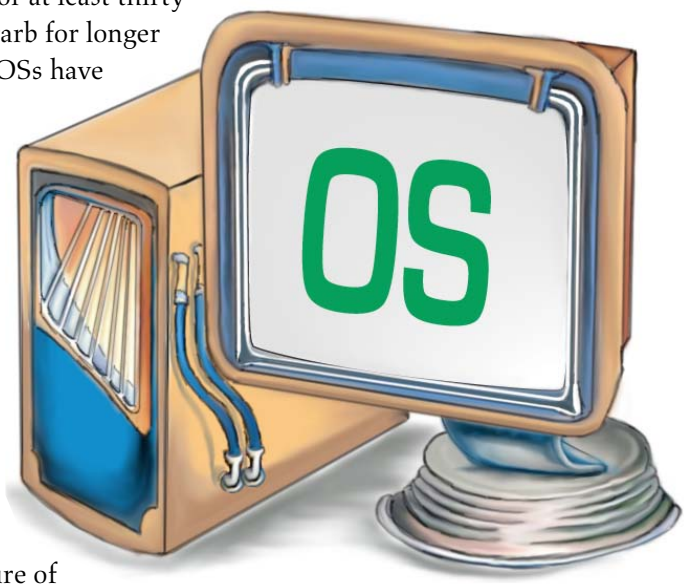
Research Note

Gordon Haff
11 January 2008

Often, when something has been around for a long time, we start to think of it as just immutably part of the way things are. Part of it is habit and familiarity. Are a keyboard and mouse truly the best way to interact with our (non-gaming) computers? Maybe yes, maybe no. But radically different input devices tend to feel odd. And most people don't like odd.

It's also the case that the whole technology ecosystem resists fundamental changes to the way we do things. Technology products plug together in certain ways and in particular places. The whole vendor landscape often has to go through wrenching change to alter these relationships. Consider, for example, the evolution from the largely vertically-integrated computer industry that was the model until the mid-1980s or so to today's much more horizontally-oriented one.[1]

The operating system (OS) is an example of technology that has existed in essentially its current form for at least thirty years—and in recognizable garb for longer than that. During that time, OSs have gotten more sophisticated, gained nicer interfaces, and generally subsumed more and more functions that were once handled by separate products over time. Today's OSs are also far more likely to have been written by someone other than the system manufacturer than was the historical norm.



However, the basic architecture of an OS hasn't much changed. It's software that runs on top of a computer system[2] and provides services for applications—which are what users actually care about. This historical structure continues to have a lot of implications for how

[1] That is, today, chips, software, servers, and so on are more likely to come from a variety of sources (even if someone—such as a system vendor—integrates them all together for customers).

[2] This note concentrates on, and gives examples from, the server world, but most of the same principles apply to client devices as well.

applications are developed and qualified, how systems are managed, and how the vendor landscape is populated.

But we're starting to see changes that affect how the operating system relates to other pieces of the hardware and software stack. Which means that everything that's largely the result of the OS's being like-it-always-has-been is increasingly subject to change.

An Historical Perspective

The first computers didn't even have operating systems. Users ran programs that controlled the entire machine and explicitly told the hardware how to perform calculations or do other tasks. The first operating system is generally considered to be the GM-NAA I/O system created in 1956 by Bob Patrick of General Motors and Owen Mock of North American Aviation for the IBM 704 mainframe. A lot of things were left as an exercise for the users in the early days. GM-NAA I/O, its successors like SHARE, and other early OSs were all written by customers. Even after vendors started to develop OSs on their own—for example, IBM released the FORTRAN Monitoring System (FMS) and then IBSYS based on the earlier SHARE work—OSs and other software tended to be fairly specific to a given model of computer.

Over time, operating systems became more and more sophisticated. This evolution was, in part, to help programs make use of increasingly sophisticated hardware. For example, the operating systems that IBM developed for the System/360[3] added the concept of multitasking—the ability to rapidly switch among multiple tasks—as a way of coping with the disparity between an increasingly speedy CPU and relatively slow storage peripherals.[4] Another thread of OS development began in the early 1960s when the Compatible Time-Sharing System (CTSS) was developed at MIT. The successor to CTSS, MULTICS, was a commercial failure, but it paved the way for important future products and technologies, including Unix.[5]

The history of Unix can (and has) filled books. There was also a plethora of operating systems for various minicomputers that evolved somewhat in parallel with Unix (and would largely be displaced by it). These included Digital's VMS (now owned by HP and called OpenVMS), Data General's AOS/VS, HP's MPE, Prime's PRIMOS, and so forth. However, by about the late 1970s a certain pattern had emerged that wouldn't really change until Windows and, later, Linux started to see widespread commercial use. Namely, each vendor had one, or a few, operating systems that ran across one or more of their product lines and that were specific to their own hardware. Even the adoption of Unix didn't alter this reality much—given that the "Unix Wars" resulted in a plethora of Unix versions that were similar—but not completely compatible.

Windows and Linux changed all that—in some respects. Now, one could obtain third-party operating systems that could run across gear sourced from multiple vendors. However, both a cause and a consequence of these third-party OSs is that there's much less variety in the underlying hardware.[6] For example, Windows runs only on x86, which is also far and away the most common processor architecture used to run Linux.

Yet, even the advent of mass market OSs for the server market hasn't really changed the fundamental nature of the operating system—which still has much in common with VMS and its relatives and, to an even greater degree, the proprietary Unixes. This probably shouldn't be especially surprising. After all, Linux is very much a member of the family of Unix operating systems; its innovation stems far more from collaborative

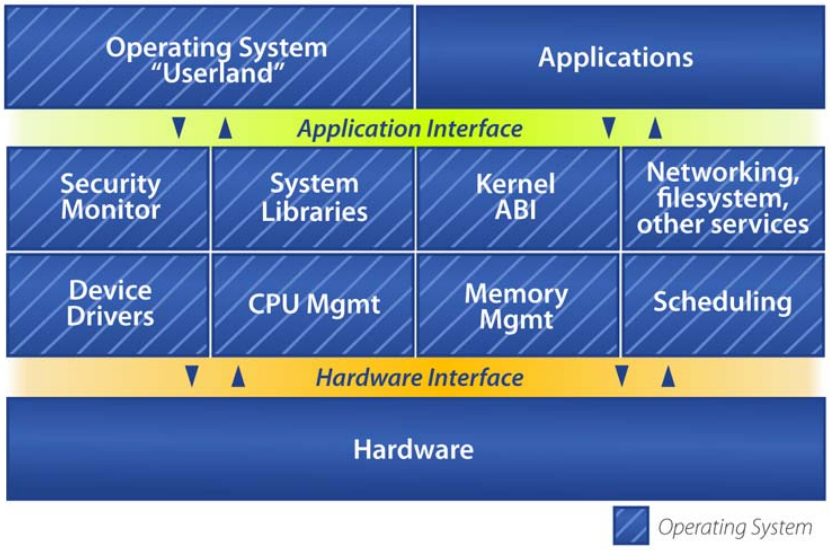
[3] As famously chronicled by Frederick Brooks in The Mythical Man-Month.

[4] A performance issue that remains with us today in spite of immensely faster components everywhere.

[5] See our *z/VM: Teddy Bears and Penguins*.

[6] Of course, there are many factors at play here including the greatly increased capital requirements to build modern microprocessors and the fact that middleware and other software is also now more likely to come from an independent software vendor than the company that built the server.

development and acceptance by a wide range of vendors than any fundamental advance of operating system design. Windows NT and its successors were also clearly inspired by operating systems like VMS.[7] Again, Windows has always been notable—and remains so today—far more because of the volume economics associated with it than because it's an especially interesting or unique product from a technical perspective.



What is an Operating System Anyway?

So what is this operating system thing that hasn't changed much, exactly?

I could approach this question from an architectural perspective. The core of an operating system is its "kernel"—the part that handles the management of computing processes and their use of memory. Over the years, OS kernels have grown to also encompass file systems, networking stacks, and access control mechanisms. However, we often use the "operating system" term to also include everything from graphical user interfaces (GUI) to file manager utilities to Windows Solitaire. This expanded set of functions and utilities also sometimes goes by the "operating environment" moniker, although that terminology has never caught on in a widespread way. By way of concrete example, "Linux" properly refers only to the Linux kernel but, in practice, is widely used to encompass all the "stuff" that comes in a typical distribution.[8]

However, for reasons that will become clear, I think it's more useful to view operating systems in the context of the basic functions that they provide.

The first function is to communicate with, and abstract, the underlying hardware. In this, the OS works hand-in-hand with various system hardware and standards for the hardware itself. Thus, higher levels of the software stack don't especially need to concern themselves with which chips QLogic decided to use in its latest disk controller; instead they talk to a device driver in a well-defined, largely standardized way. The OS is also very involved with handling a lot of the basic management of processes and memory so that, for example, application programs don't typically worry about where exactly on the physical server their instructions are executing, or on what disk drive, platter, and sector their data are stored.

Another function of an operating system is to provide services to applications. These services might include a TCP/IP networking stack or system libraries for programs written in a compiled language such as C++. One important aspect of these services is that they stay relatively consistent and stable. Put another way, they constitute a sort of "contract" for applications that run on top. There's effectively a promise that if you write a program to the defined operating system interfaces things will still work even if you install a new device driver for a new type of storage device or if you double the number of processors.[9]

Finally, operating systems as we usually think of them provide a wide range of utilities and other

[7] Dave Cutler was heavily involved in the design of both as described in G. Zachary Taylor's Show Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft.

[8] I discuss this in the context of OpenSolaris and "Project Indiana" in our *A Better Linux Than Linux*.

[9] This is a bit idealized, of course. Operating systems do change over time—especially when making major technology transitions—but backward compatibility is still an important goal.

applications to manage various aspects of system operation, provide convenient ways to carry out various administrative tasks, tune performance, and generally offer a handy general-purpose toolset to augment more task-specific or in-depth software obtained from third parties. These are the broader set of components associated with the operating environment described earlier; some—such as GUIs—also are effectively part of the application contract to at least a subset of applications.

The Virtualization Game Changer

With that as prelude, it's apparent that the OS isn't one thing but, rather, several things that are unified more for reasons of past practice and convenience than technology imperative. System vendors originally built operating systems the way they did because they had to build all the software below the application layer anyway. Even after third-party operating systems became important parts of the market mix, they largely mimicked the existing structure.

Now, consider this.

The first function of the OS that I described—abstracting and communicating with the hardware—sounds a lot like another piece of technology that's become popular of late: the hypervisor layer that underlies virtual machines.

Because the hypervisor is the code that is actually sitting directly on top of the server and controlling it, the hypervisor implements a lot of the low-level hardware resource management that the OS historically had to handle on its own. A hypervisor also adds capabilities that most operating systems didn't traditionally have—specifically the ability to itself host a number of isolated OS instances and their associated applications.[10] Partly as a consequence, a hypervisor also presents a more abstracted view of hardware to the higher layers. For example, "virtual CPUs" and "virtual NICs" don't need to correspond to the quantity of physical hardware actually present. They also strive to be as generic as possible—although, in practice, the desire to completely abstract the underlying hardware has to be balanced against the desire to take advantage of existing driver software, vendor-specific hardware acceleration, and other features.

One significant implication of all this is that the "operating systems" that sit on top of a hypervisor don't necessarily have to be OSs in the traditional vein, because the hypervisor is already handling many of the tasks associated with scheduling and other functions associated with the physical hardware. Rather, the OS running atop a hypervisor can major in just providing system services for applications.

This is hardly a theory any more. Linux on the mainframe, running under z/VM, uses precisely this approach. Linux scalability has improved greatly over the past decade, but still can't hold a candle to the biggest of the Big Iron operating systems. But under z/VM, it doesn't have to. The z/VM foundation (and a lot of support by and for System z hardware) does almost all the "heavy lifting." Linux primarily provides the application abstraction—the environment that developers and customers want to use, so that they bring their applications on board.[11]

Taken to its logical extreme, one can even imagine specialized OSs that provide a well-defined subset of services for specific applications—the idea being that a stripped down OS tailored to a specific purpose could have a smaller footprint and be more efficient than the historical general-purpose OS that had to be prepared to do anything. This sort of architecture isn't wholly theoretical even today. BEA's LiquidVM—a virtualization-enabled version of BEA's JRockit JVM—can run directly on top of VMware's ESX Server without a conventional OS present. BEA claims performance boosts of up to 40 percent relative to running a JVM on top of a standard operating system.

[10] See our *The Server Virtualization Bazaar, Circa 2007* for a more detailed discussion of virtual machines and other forms of virtualization.

[11] See our *z/VM: Teddy Bears and Penguins*

Virtual Appliances

There's also another aspect to how virtualization plays into the future of the operating system: virtual appliances.

Normally, we install applications on top of operating systems whether that operating system is running on bare metal or within a VM. Installation typically involves copying files, creating directory structures, adapting to the physical and software environment of the host, telling operating system resources (such as menus, environment variables, or registry settings) about the new program, and giving the user the opportunity to customize some settings. Installation can be a complicated and time-consuming process—all the more so when software depends on other software (perhaps even specific versions of that other software) to function. For example, a Windows program might require a specific version of the .NET framework. Linux or Unix installations often require various language compilers or interpreters, databases, templating systems, and so forth.

Virtual appliances offer an alternative. The idea is that a software vendor can take its application, the operating system it runs on, and all of the required supporting programs, libraries, and what have you; configure the whole mess properly; and then write it out to disk, ready to be fired up as a self-contained, ready-to-run virtual machine. Thus there's no need for the end-user to install anything other than the simple virtual appliance. The app comes packaged together with whatever else it needs to run—including OS.

No doubt appliances have been oversold at times—never more so than during the first Internet boom when some envisioned that hardware appliances would replace general purpose servers for running everything from databases to email.[12] And, even with infinitely "softer" and more transportable virtual appliances, the fundamental appliance concept implicitly assumes that appliance images don't have to routinely be modified, patched, or tweaked, in which case the "just fire it up in a virtual machine" promise starts to ring a bit hollow. Still, some types of applications (such as firewalls and web servers) can indeed be quite standard—at least within a given enterprise—and others have such truly ugly dependency sets that *anything* providing at least some simplification is welcome.[13]

What makes virtual appliances relevant to this discussion, however, is that they effectively subsume the OS. In a virtual appliance, you can think of the operating system as just part of the "stuff" that the application needs to run on top of a hypervisor. At least in principle, the user shouldn't really need to care about the details of that OS any more than you care what operating system your Linksys router or your car's GPS is running.

In this view, the operating system in a virtual appliance is a JeOS ("Just enough OS"), a customized operating system that's been tailored to the needs of the specific application. At the same time, a JeOS running atop a hypervisor could also potentially jettison the parts that the hypervisor makes redundant. The advantage of this approach is that the OS footprint can often be trimmed considerably. For example, software appliance vendor rPath cites being able to cut down Zimbra's CentOS-based[14] desktop software download from 2.1 GB to a few hundred MB by tailoring the OS. In this case, the main motivation was to reduce the download size. However, stripping out unnecessary components can also reduce memory footprint and even eliminate services that could be vectors for various types of security attacks.

Cutting Out the Dependencies

Up to now, I've focused on virtualization's effect on the future of the OS—both from an architectural perspective and because it has significant potential to change how applications get deployed. However, there are some other parallel developments that are driving change as well.

[12] See our *The End of Cobalt and the Appliance Era that Never Was*.

[13] See our *Virtual Appliances Evolve* for more background about how virtual appliances are coming to market.

[14] CentOS is a Linux distribution that is based on Red Hat Enterprise Linux.

Historically, applications were very much tied to a given OS running on a given variety of hardware. In the early days, applications written in assembly language—essentially a human-readable version of the machine language that a processor actually executed—were highly specific. However, programs written in so-called high-level languages (such as FORTRAN and C++) also require tweaking even for similar platforms (such as different flavors of Unix) and may require substantial "ports" in more extreme cases. In any case, even when the source code is similar, the final compiled binaries—the code that executes on the system—are different for each OS and processor variant with concomitant implications for support effort and cost.

This 1:1 relationship between OS and application version has had a significant effect on the whole structure of the computer industry over the past 10 to 15 years. As ISVs like Oracle evolved to become a more and more important part of the landscape, they effectively picked winners and losers by deciding which of a limited number of platforms they would support with their applications. Declining application availability on lower-volume server and OS types was perhaps the most significant factor that either drove second- and third-tier system vendors out of the business or forced them to adopt "industry standard" platforms. This whole dynamic still largely rules today. However, we're seeing an increasing swath of applications written either in "managed code" and "scripting languages," or that are dynamically translated and rehosted.

"Managed code" includes languages such as Microsoft's C# running on Microsoft's .NET framework and Java running atop Java Virtual Machines (JVM). Both .NET and JVMs provide abstractions of the underlying machine and operating system services, insulating programs from the "bare metal." So long as a Java program, say, doesn't bypass the JVM to talk directly to underlying system services, Java code is mostly blissfully ignorant of the OS. Perhaps things are often not quite as straightforward and perfectly universal as Java's "write once, run anywhere" mantra would suggest. Nonetheless, these runtime systems provide a very high degree of program transportability, even across divergent CPU architectures—from x86 to Itanium, RISC, mainframes, and even custom processor designs found in cell phones or Azul's Compute Appliances.[15]

Scripting languages such as Perl, PHP, Python, and Ruby are widely-used for Web-centric applications. Programs written in scripting languages can often run completely unchanged across a variety of different OSs and even processor architectures, so long as OS has the necessary pieces of software infrastructure (interpreters, databases, and so forth). They often use "bytecode interpreters" much as Java and .NET do,[16] but it's the supporting infrastructure—the modules of Perl, say, or the Rails framework for Ruby—that does much of the hard work of insulating programs from lower level system resources. As with Java and .NET, that's not to say that there aren't gotchas and quirks to overcome, but there are nonetheless far fewer dependencies than when an app is compiled for a specific operating system.

Dynamic translation (and dynamic rehosting) runs binaries from one operating system, runtime environment, and CPU choice on a target system that makes potentially very different choices. This approach has been tried many times over the years, with disappointments and more than a few outright failures. But the approach has recently been significantly advanced by Transitive.[17] It got its big break by providing the technology behind "Rosetta" that allows PowerPC-based Apple OS X applications to run unchanged on Intel-based Macs. The IBM System p Application Virtualization Environment (System p AVE or, informally, pAVE) also uses Transitive's QuickTransit to let 32-bit x86

[15] The Common Language Runtime (CLR) that supports .NET is arguably a little less general than Java's JVM, running mainly on x86 and Itanium processors. But that is really a business decision on Microsoft's part, rather than a technical difference.

[16] Some even piggyback. Jython and JRuby, for example, respectively implement Python and Ruby over Java's JVM infrastructure.

[17] See our *Transitive's Translations*.

Linux applications run atop Linux-on-POWER. Transitive can not only recode CPU instructions, but also library calls. That is, it can dynamically rehost from one operating environment to another. It's not easy, but there are many data points that now show it not just works, but works well, at least as a transition mechanism.

The significance of all this is that the OS and the application were historically very much intertwined. The OS mattered to the app and the app vendors because every app version was married to an operating system version. In a world of PHP and Java, that relationship is far weaker. To be sure, because of support concerns, software vendors will continue to limit the number of platforms for which they'll certify their applications. Nonetheless, this new reality of applications using interfaces that are largely independent of the OS and even the underlying hardware opens the door for running those apps in ways that don't depend on one of the handful of today's widely-used operating systems. We see an extreme example in BEA's LiquidVM. However, we also see it in the panoply of Linux distributions running LAMP stacks with applications that don't require the same sort of certifications that are a major raison d'être for the enterprise Linux distributions from Novell and Red Hat.

Will the OS Matter?

At the same time, there's both inertia and pragmatic arguments in favor of general purpose operating systems—even in virtual appliances. These include the issues associated with transitioning from the physical world to a virtual one, the fact that ISVs don't necessarily want to be in the operating system business, and the reality that operating systems aren't wholly invisible.

Virtual appliances are an interesting development and they could very well bring major changes to the way that we deploy software. But it will take some time. They're still in their infancy, and are mostly used today to get demo software up and running quickly. Thus, it's inevitably going to be less appealing to an ISV to go off and create a unique, optimized operating system for a small slice of their market—especially when they're also busy figuring out licensing and other issues associated with virtual appliances. Tools like rPath's rBuilder aim to streamline the process but, nonetheless, it's some degree of extra upfront development and back-end support work to add "yet another platform" to their development and qualification matrix.

Nor do a lot of ISVs want to suddenly assume the responsibility for updating and supporting a lot of foundational components that are someone else's problem today. Users also don't necessarily trust some small ISV to keep some customized Linux distro up-to-date with security patches. Just because Oracle wants to own the whole software stack doesn't mean that everyone wants to.[18] The promise of the JeOS approach is that by building the software as a unified structure, things will be more reliable and easier to support—making the extra effort worthwhile—but it's too early to truly know how well this will be borne out in practice.

Finally, it's fine conceptually to talk about operating system details being irrelevant, but that's an idealization and oversimplification. Even if you're "just running an application," you often still need to make trips to the command line or to operating system tools that manage user access or performance or whatever. The difference between Linux distributions may not matter too much—at least until you have to use some distro's unique packaging system to add some new software component. But Linux and Solaris, AIX and HP-UX, (much less Windows and OS X) differ significantly from each other.

These are among the reasons that Red Hat has itself introduced an appliance version of its Linux, the Red Hat Appliance Operating System (AOS). The idea is that applications that are certified for Red Hat Enterprise Linux—which has more certified apps than any other Linux distribution—

[18] See our *Oracle: Just Say No to Operating Systems*.

will carry those certifications over to AOS. Red Hat hopes that this will be a more natural bridge for both ISVs and end-users than bringing in something special just for appliances.

I expect that we will see a variety of approaches as virtual appliances become more mainstream. In some cases, the optimization and efficiency of a crafted virtual appliance will win out, but, in many others, the familiarity of a standard and accepted operating system will be preferred.

Conclusion

2008 will not be the year that the operating system changes in a radical way. Indeed, it probably won't happen to any substantial degree in this decade. However, what we will see in 2008 is server virtualization continuing its march to become just part of "how computing is done."

The consequences of that will affect how we do many things in computing—including provisioning applications, optimizing power and cooling, and managing workload changes. And it will ultimately affect what the operating system looks like. If one accepts that hypervisors will essentially fold into the hardware over time,[19] it seems inevitable that we'll also evolve the operating system to be a better architectural match for that environment. We already see some early glimpses of various possible futures that differ from today's general purpose OS model, whether BEA's LiquidVM or rPath's customized Linux-based appliances.

The OS will change, but it will be mostly a matter of slow evolution rather than rapid seismic shifts.

---

[19] See our *Will We See an Embedded VMware "ESX Light"?* and *The Server Virtualization Bazaar, Circa 2007*.