Notes on Virtualization

Version 9.0

Andrew Herbert
6th October 2015

Introduction

The story of virtualization and the important role it has played in Operating Systems over the past 50 years. The systems and papers mentioned in the text are the author's choice of exemplars for the concepts being discussed, they not an exhaustive list of all related research and products, nor a statement about precedence over other systems.

What is "Virtualization"?

In broad terms, "virtualization" in the context of operating systems principles relates to operating systems that give the illusion of efficiently running multiple independent computers known as "virtual machines". Depending upon the specific requirements and means of virtualization, the virtual machines be directly executed by (i.e., exactly mimic) the underlying physical machine, or they may comprise a more abstract system, parts or all of which are simulated by the physical machine.

Origins

The term "virtual" originated with "(dynamic) virtual memory" to express the abstraction of having a large contiguously addressed store available to programs, implemented by swapping portions of data and program between main store and magnetic drum. Some of the pioneers of operating systems (Maurice Wilkes, Brian Randell, Bob Nelson) have suggested the use of the word "virtual" was inspired by optics, in the sense of "virtual" versus "real" images.

The term "virtual" was then extended to include "virtual devices" with the advent of input-output spooling and file systems. A typical virtual device interface would replace the hardware input-output instructions and device interrupt handling structures with a more convenient abstract interface with functions such as OPEN, CLOSE, READ, WRITE, CREATE, and DELETE. Similarly some referred to multi-programmed processes as being "virtual processors", e.g., in his 1966 PhD thesis about The Compatible Time Sharing system (CTSS) – . - F. Corbató, M Daggett, R. Daley, "An experimental time-sharing system", *Proc. Spring Joint Computer*

Conference, May 1962 – Jerry Saltzer defined “process” as “program in execution on a virtual processor.”

The phrase “virtual machine” came to mean the complete environment of a virtual processor defined in terms of an instruction set architecture (which could itself be fully or partially virtual) with virtual memory based addressing and a virtual device interface for input/output. With the advent of computer networks, the definition further widened to include uniform, location-independent interfaces to services available throughout a distributed system.

In terms of exploring the history of virtualization, it is useful to explore four related strands of development:

1. Partitioning machine resources into efficient, isolated copies mimicking the parent machine, some other physical machine, an entirely abstract machine or some combination thereof.

2. The structuring of operating systems as a hierarchical layers of abstract machines,

3. Providing an application execution environment independent of the details of the underlying computer hardware,

4. Uniform, location-independent interface to services available throughout a distributed network.

From Paging to Virtual Machines

The concept of “dynamic virtual memory” was invented for the University of Manchester Atlas computer where it was called “paging”. The term “dynamic virtual memory” was coinded by IBM. The concept was simple – to enable more programs to be held in store to maximize the opportunity for multi-programming, divide programs into small fixed size blocks called “pages”, stored on a magnetic drum and only copy those pages active at any moment in time into “page frames” in the physical store (i.e., there was an assumption of locality in program execution and data access, allowing a job scheduler to ready a set of jobs whose total addressable memory needs exceeded the size of the physical store available). When the physical store became full, idle pages would be reclaimed (after writing back the contents to the drum if necessary). The page frame could then be used to bring in a newly active page from the drum. Paging was intended to automate (and do better than) previous manually programmed “overlay” systems for swapping in and out program units.

The advent of paging immediately spawned interest in designing algorithms for the optimal selection of pages to replace when physical store was full. Around 1965 a group at IBM Yorktown Heights led by Dave Sayre and Bob Nelson built an experimental machine, the M44/44X, based on the IBM 7044, for the purpose of studying the performance of virtual memory. The Atlas designers had believed, and it was quantitatively confirmed by the IBM team, that manual overlaying was a serious hit to programmer productivity; in their experiments, they documented 2x to 3x improvement in programmer productivity enabled by dynamic virtual memory.

Each user process in the M44/44X system was described as a "virtual [IBM 7044] machine", although it was not a full virtualization of the hardware in the sense of later systems: in essence the virtual machine monitor allowed different processes organize their own virtual memory management system – an early example of the principle of policy/mechanism separation.

Using the M44/M44X system, the IBM team compared paging algorithms, page sizes, and load controllers to prevent thrashing, leading to Belady's famous 1966 paper – L. Belady, "A study of replacement algorithms for a virtual storage computer", *IBM Systems Journal*, **5**, 2, 1966, pp 78-101. Belady had the first glimmerings of the principle of locality when he noted that references to memory were not random and exploiting usage bits in page tables led to better performance. This was followed shortly (and independently) by Peter Denning with the "working set" model at SOSP-1 in 1967 – P. Denning, The working set model for program behaviour, *Comm. ACM*, **11**, 5, 1968, pp. 323-333.

Virtual Machine Monitors

The virtual machine concept was taken further by two operating systems CP/40 and CP/67, for the IBM System 360 developed by the IBM Cambridge Scientific Center around 1967 (CP standing for "control program" and the numbers representing the target models) which achieved full virtualization – i.e., the ability for the host machine to run multiple virtualized copies of itself. There were three drivers for this work: (1) to implement time sharing by running each user's session as a virtual machine instance; (2) to be able to run copies of other IBM operating systems on the same machine simultaneously to obviate the need to rewrite applications for every possible IBM/360 operating system; (3) to allow IBM programmers to develop and test systems software using the time-sharing facilities available application developers.

(IBM has learned of time-sharing from collaboration with Project MAC at MIT on CTSS. There were a surprising number of operating systems for the IBM/360 range that reflected the capabilities and typical usage of the various models: the smaller models were not capable of running the functionally richer operating systems designed for the larger models.)

The virtual machine concept was further consolidated in the VM/370 operating system for the IBM/370 range that succeeded the IBM/360 range.

VM/370 had three principal components. First, the Virtual Machine Monitor (or hypervisor in modern parlance) which supervised the execution of virtual machines, assignment of physical devices to virtual machines, virtualization of card readers and printers (via a spooling system) and the mapping of virtual discs to physical discs – each physical disk was divided into so-called "mini-disks" that could be assigned to individual virtual machines (and shared by multiple virtual machines if appropriate); second, the Conversational Monitor System (CMS), essentially a single user operating system to run inside a virtual machine and the Remote Spooling and Communication Service (RSCS) that provided networking facilities between machines running VM/370. Together these three provided the elements of a flexible time-sharing system. Because the hypervisor provided a "pure", i.e., full emulation of the IBM 370 instruction set architecture (ISA), and users could run any of the IBM operating systems in a VM/370 virtual machine in place of CMS should they wish. (This was often used as a migration aid when updating a system from an older operating system to a newer one, e.g., from OS360/MVT to OS/360MVS when moving from a non-paged machine to a paged machine. Following on from the original M44/44X work, a virtual machine could be put in an "extended mode" to allow the guest operating system control its own dynamic virtual memory management).

Seeing the benefits to IBM, various other manufacturers explored virtual machine operating systems for their hardware, although it was not always possible to fully virtualize their ISAs. In 1974 G. Popek and R. Goldberg published their paper on "Formal requirements virtualizable third generation architectures", *Communications of the ACM*, **17**, 7, 1974, pp. 412-421. They define virtual machine monitor as providing an identical essentially identical to the parent machine, and that programs run in this environment with at most only minor decreases in speed. They identify non-privileged "(physical store) location sensitive" and "mode (supervisor/user) sensitive" instructions as potential barriers to being able to fully virtualize an ISA. Pointing out that few third generation architectures met their requirements, Popek and Goldberg coined the term "hybrid virtual machine monitors" for system in which all privileged mode instructions in the virtual machine system were simulated to overcome the problems with sensitivity rather than executed directly, but they saw this as compromising their "only minor decreases in speed" criterion. This did not deter others from exploring hardware modifications and/or system compromises to side step machine limitations, e.g., S. Galley "PDP-10 virtual machines", *Proc. ACM Workshop on Virtual Machines*, 1973, pp. 30-33 and G Popek & C. Kline, "The PDP-11/45 virtual machine architecture: a case study", *Proc. Fifth ACM Symposium on Operating Systems*, 1975, pp. 97-105.

After the first wave of virtual machine research in the decade 1965-75 virtual machine monitors were rarely mentioned at SOSP for nearly 20 years, although the related concept of having a small kernel allocate resources to operating system services running in processes (i.e., in a virtual machine) was taken forwards in work on microkernel and related operating system structures.

Later Developments

Virtual machine research resumed in the mid 1990s. Two projects exploited virtual machine monitors as a means to monitor (and control) the behaviour of operating systems by hosting them in virtual machines. A third project revisited the implementation of virtual machine monitors in the context of the then emerging scaleable multiprocessor workstations.

T. Bressoud and F. Schneider explored the use of a virtual machine monitor or hypervisor to provide fault tolerance: the role of the hypervisor being to coordinate state machine replication by intercepting and re-ordering incoming events to replicated VMs (T. Bressoud and F. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems*, **14**, 1, 1996 pp. 80-108). Bressoud and Schneider proposed this approach as an alternative to hardware-based fault tolerance, observing that fault tolerant hardware was often a generation behind commodity hardware because of the additional design complexity, whereas as software based approach could use the latest (and therefore faster) processors. In practice the software-based system ran at half the speed of a single replica due to the communications and coordination overheads, thus comparable to the performance ratio between success generations of processors typical at the time, making the result a "draw".

In similar vein, the ReVirt project at Michigan explored the use of a virtual machine monitor to capture and replay the execution of a system as a means of "intrusion detection" – G. Dunlap, S. King, S. Cinar, M. Basrai and P. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay", *Proc. 5th Symposium on Operating Systems Design and Implementation*, 2002.

In essence both these projects were following on in the tradition of using virtual machine technology by IBM in the early days to develop and debug new operating systems.

By contrast the 1997 DISCO project revisited the use of a virtual machine monitor to run alternative operating systems side-by-side – "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," E. Bugnion, S. Devine, K. Govil and M. Rosenblum, *ACM Trans on Computer Systems*, **15**, 4, 1997, pp. 412-447. As suggested by the paper title, one aim was to be able to run multiple instances of a commodity operating system (IRIX) to fully utilize the multiple processors available. In addition the DISCO VMM provided a communications abstraction based on TCP/IP to enable parallel computations to be distributed across multiple virtual machines (which could in turn be assigned to separate physical processors). In this respect DISCO looked like a distributed system implemented on a (shared memory) multi-processor. Importantly, because the DISCO VMM had control over all the resources of the multiprocessor it could optimize their use and take advantage of sharing to reduce the performance overheads of the system.

DISCO was implemented on MIPS R1000 processors, which could not be fully virtualized, and so it was necessary to develop a hybrid approach. DISCO shadowing physical processor state as a data structure in the (virtual) address space of each virtual machine and using binary rewriting to replace non-privileged sensitive instructions with new code that accessed and updated the shadow state, notifying the VMM when appropriate.

DISCO also included a virtual memory system that aimed to optimize physical memory utilization by remembering the disk block from which each page frame was loaded. If several virtual machines read the same block the same frame would be mapped into their virtual address spaces. If a virtual machine wrote to a shared page, the shared reference would be replaced by a reference to a new frame allocated to a private copy of the page for that VM. The DISCO authors also comment on the opportunity to use a virtual machine approach to load balance by migrating virtual machines between processors and fault tolerance through replication in the manner reported by Bressoud and Schneider.

Modern Virtual Machine Monitors

By the late 90s, the PC industry had re-created for itself the same problem as that faced by the mainframe vendors – the need to run applications tied to different operating systems (e.g., MacOS, Linux, Windows) alongside one another and to carry legacy applications on an older version of an operating system to the latest one. A number of vendors introduced desktop virtualization systems, including for example VMWare's Desktop for Windows in 1991. These systems are different from the early virtual machine monitors in that they ran as applications on a host operating system (e.g., Windows, MacOS or Linux) and were able to use the services of the host operating system to implement virtual file systems and virtual network interfaces. Generally the VMM application would load a driver of similar into the host OS kernel to provide the system interface to the hosted VMs and similarly applications and drivers for the guest OS to better integrate with the host OS for file sharing, window management and so forth. To distinguish these hosted VMMs are often described as being "type 2", in contrast to "type 1" virtual machines than run on "bare metal".

Until the appearance of the "VMX extensions" in 2006 virtualization systems for the x86 architecture necessarily had to adopt hybrid techniques such as the simulation and/or shadowing and binary rewriting used DISCO and this imposed a performance overhead. To overcome this others explored "paravirtualization", e.g., the Xen system – P. Barham, et.al., "Xen and the Art of Virtualization," *Proc. Symposium on Operating Systems Principles 2003*, October 19-22, 2003).

In a paravirtualized system the hosted operating system is modified to cooperate with the virtual machine monitor through a dedicated interface to manage changes to privileged virtual machine state. This improves performance, but of course requires access to the internals of the guest operating system that may not be possible in the case of proprietary systems. In general the modified code sits at the lowest level of the guest operating system (e.g., for Xen, in the "hardware adaptation layer" of Windows) so the main bulk of the guest operating system remains unchanged.

With the advent of VMX these issues have become less important, however the efficient statistical multiplexing of the host's resources across a set of executing virtual machine remained problematic. This was addressed by Waldspurger for VMware's type 1 "ESX Server" virtual machine through two features, "ballooning" and "content-based page sharing" – C. Waldspurger, "Memory resource management in VMware ESX Server", *Proc. Fifth Symposium on Operating Systems Design and* Implementation, Dec. 2002, pp. 181-194. Ballooning is a strategy for managing paging – the VMM allocates itself space in the virtual address space of each VM and by inflating (or deflating) the size of its region provoke the guest operating system to reduce (or increase) its memory usage, allowing the VMM to implement a global policy across all running VMs. Content based page sharing looks to identify file sharing by comparing the content of pages and if they are identical have them share the same physical page frame – an evolution of the page sharing used in DISCO.

With these advances, virtual machine systems are nowadays a widely used technology: but with a rather different set of drivers than the early systems:

- "Client" virtual machine systems
    - Desktop coexistence, e.g., running Windows alongside MacOS
    - Desktop migration e.g., for system upgrade, user relocation
    - Desktop backup and recovery
- "Server" virtual machine systems
    - Server consolidation – reducing operating costs by co-hosting server systems on the same physical box
    - Statistical resource multiplexing – provisioning virtual machines large enough to cope with intermittent peak loads
    - Multi-tenanted systems – using virtual machines to isolate systems running on behalf of different users
    - Fault containment and recovery – using virtual machines as the unit of failure and recovery
    - Managing server workloads through VMM APIs for (virtual) server provisioning and control.

Interestingly in all these cases, modern uses of virtual machines are more about dividing up large physical systems into a large number of smaller virtual machines, whereas in the early days the intent was to share the resources of a small physical machine across a number of larger virtual machines!

It is also worth noting that a number of current systems support a simplified virtual machine concept known as “containers” where the focus is virtual memory management and isolation, in many ways a step back to the initial work at IBM on the M44/44X system!

Structuring Operating Systems as a Hierarchy of Abstract Machines

Early operating systems were structured as monolithic programs running in privileged mode and written using assembly code. As additional functionality such as filing systems and time-sharing was added this structure became unwieldy. The operating system occupied too much store, the internal structure became increasingly complex, especially when dealing with concurrent activities and coding blunders in one area of the system could lead to disastrous effects elsewhere. It became necessary to impose finer grained structure on the operating system. An important early concept was to think of the operating as a hierarchical series of layers each providing a more abstract (and hopefully simpler and safer) machine to higher-level functions, with the user process as the most abstract machine of all.

An early example of the approach was the operating system for the X.8 written by Edsger Dikstra and colleagues at the Eindhoven Technical High School – Dijkstra, E.W. (1968), "The structure of the 'THE'-multiprogramming system", *Comm. of the* ACM, 11, 5 pp. 341–346. This was a simple batch operation system for running Algol 60 programs. It was organized as four layers. The first layer abstracted out details of concurrent processing and interrupt handling through providing the concept of self-contained sequential processes coordinated by semaphores for control access to shared resources (i.e., devices, regions of storage). The second layer abstracted out the details of dynamic virtual memory paging to give each process its own virtual address space transparently swapped between physical store and magnetic drum. The third layer abstracted out the details of the operator’s console. The fourth layer abstracted out the details of input/output including buffer management. The fifth and final layer was the Algol system with facilities for job submission and execution.

The layering idea spread out quickly in a number of directions. It influenced microkernel designs such as VENUS where the lowest levels of abstraction were implemented by microprogramming (B. Liskov, “The design of the Venus operating system”, *Comm. ACM*, 15, 3, March 1972, pp. 144-149) and later in the 1980s Comer’s XINU, initially developed for pedagogic purposes (D. Comer, Operating System Design - The XINU Approach, Prentice-Hall, 2nd Edition, 2015, ISBN 9781498712439).

The Provably Secure Operating System (PSOS) developed by SRI exploited layering to make the task of formally verifying the security of an operating system tractable in the days before we had industrial strength automated theorem provers and model checkers. Only downward call and upward return were allowed. This disciplined the

PSOS designers to clearly define the abstract objects at each level and arrange the hierarchy so that objects of one level could be only composed from objects at lower levels. The SRI team constructed formal proofs of the security relationships between the modules of their operating system layer-by-layer so that they could give cast iron security assurances to the end users. They called the approach "The Hierarchical Design Methodology". Specifications were written a notation called SPECIAL. (R. Feiertag and P. Neumann, "The foundations of a provably secure operating system (PSOS)", *Proc. National Computer Conference*, AFIPS Press, 1979, pp. 329-334.)

Compared to Dijkstra's five layers, PSOS nominally had 17 layers in the specification as follows:

1. Capabilities (for memory protection)
2. Registers and other storage
3. Interrupts
4. Clocks
5. Arithmetic and other basic procedures
6. Primitive input-output
7. System processes and system input-output
8. Pages
9. Segments and windows
10. Abstract object manager
11. Directories
12. Creation and deletion of user objects
13. User processes and visible input-output
14. Procedure records
15. User input-output
16. Environments and name spaces
17. User request interpretation.

However in the implementation several these were grouped together into 9 levels which interestingly correspond closely to the operating system reference model developed by Peter Denning and colleagues nearly 10 years later and listed below: P. Denning, W. Tichy, and J. Hunt. "Operating Systems," in *Encyclopedia of Computer Science* (A. Ralston, E. O'Reilly, D Hemmendinger, Eds.) 1999. Republished in 4th Edition, Nature Publishing Group, Grove's Dictionaries (2000), 1290-1311.]

1. Low-level input-output
2. Threads
3. Memory management
4. Inter-process communication
5. Processes
6. Stream input-output
7. (File System) Directories
8. Command shell
9. Graphics server.

Type Extension

The notion of layering abstractions fits naturally with the concept of hierarchy in object-oriented systems and later operating systems picked up on this their approach to system structuring. A notable piece of work was that by Habermann and colleagues at CMU on FAMOS, an attempt to build a family of related operating systems from re-usuable components (modules) – A. Habermann, L. Flon, L. Cooprider and P. Feiler, "Modularization and hierarchy in a family of operating systems", *Comm. ACM*, **19**, 5, May 1976, pp. 266-272. In their paper they explore the relationship between programs structured as modules (as defined by Parnas – D. Parnas, "On the criteria to used for decomposing systems into modules", *Comm. ACM*, 15, 12, Dec. 1972, pp. 1053-1058 – and the use of abstraction layers for designed, noting that often modules have to be split to straddle layers by comprising a lower layer and upper layer portion, and arranged so that the upper layer can use abstractions defined by other modules at layers in between the two. (There is an implied example of this the PSOS layers above – primitive input-output is at layer 5 to enable basic access to devices, whereas user input-output is at layer 14 and able to draw all the abstractions in between to unify devices and files.) The FAMOS paper also gives a useful summary of virtualization techniques, illustrating how a virtual processor can improve on the raw hardware by wrapping up fault handling, interrupts etc behind more convenient to use abstract functions.

In addition to those using layering to structure operating systems, others used layering to improve existing systems: for example, in the early 1970s a version of Multics was built around the concept of a security kernel to enable a more formal analysis of the security of the system and to introduce support for a model of computer security based on sensitivity levels and compartments, originating from MITRE and anticipating the later U.S. Department of Defense "Orange Book" or more formally, "Trusted System Security Evaluation Criteria". In their 1977 SOSP paper, the MIT team described how they moved large chunks of functionality out of the Multics supervisor and in effect added additional layers of abstraction to the operating system and more thoroughly exploited Multic's hardware protection rings to ensure the integrity of the layering – M. Schroeder, D. Clark, J. Saltzer, "The Multics Kernel Design Project*". Proc. Seventh ACM Symposium on Operating Systems Principles*, published in ACM Operating Systems Review, **11**, 5, Nov. 1977, pp. 43-56. The authors report that they used "type extension" as the means to rationalize complexity. In their terms "type extension" meant structuring all modules of the operating system as "object managers" and ensuring there were no cyclic dependencies between them. The authors noted that this process was somewhat of a tussle and required the introduction of new abstractions and further levels of indirection in the system structure compared to the previous Multics architecture, mirroring Haberman's approach in FAMOS.

Type extension provides a uniform basis for choosing levels of abstraction in an operating system. As we shall see subsequently, combined with capability-based addressing type extension can be augmented with fine-grained protection and enables uniform location-independent object-sharing between processes, and indeed in the case of distributed operating systems, between machines. Type extension was of particular importance in capability systems, and will be revisited in the last section on uniform naming.

In recent years there has been a backlash against layering leading to operating systems with fewer layers and more functionality in each layer. For example, the original Microsoft Windows NT had a layered oriented organization, but due to poor performance layers were reduced and more closely integrated in Windows NT 4. By contrast Comer reports that using layers led to a simpler and more efficient design for XINU than might otherwise had been the case, but it required some careful thought to design the layering structure and interfaces between the layers to achieve this. Those who chose to implement an operating system using an object-oriented language of course have the benefit of linguistic support and object-oriented design principles to help them in this task.
The network protocol world strongly believes in the principle of layering, e.g., the International Standards Organization seven layer "Reference Model for Open Systems Interconnection", ISO/IEC 7498-1. Protocol designers build protocol stacks on the idea that a given level on one machine sees the same level on another machine as a "peer" and can pass messages directly to its peer, with all the lower-level details like routing and signalling being invisible. Vint Cerf repeatedly insists that the Internet could not scale were it not organized as layers. We also see virtualization in the networking world with concepts such as virtual private networks (VPNs) and virtual local area networks (VLANs).

Hardware Independent Application Environments

The process abstraction in modern operating systems defines the environment for the execution of programs: as Saltzer defined in his thesis, *ibid*, a process [is a] program in execution on a virtual processor." The virtual processor has become increasingly abstract over time, to the extent that many programs today are written for entirely abstract machines such as the Java Virtual Machine.

In the very earliest computers such as the Cambridge EDSAC programs (programmes) were written in machine code and the application environment consisted of the hardware instruction set and a library of pre-written subroutines for input, output and mathematical functions. With the fast pace of technology development computer manufacturers rapidly introduced successor machines and invariably these came with new instruction sets requiring the manufacturer to rewrite the systems software and users to rewrite their applications to the new architecture. To overcome this it was common for the newer machine to be able to emulate its predecessor, there would be an instruction that switched the machine between instruction sets. For example some models of the IBM System/360 included hardware emulation of the earlier IBM 1400 and 7000 series machines. Instruction set emulation was often facilitated by microprogramming (introduced by EDSAC 2) and was perhaps an early example of the RISC versus CISC debate – the basic hardware was optimized for fast execution of microcode which simulated higher-level application oriented instruction sets.

A similar problem arose with the introduction of "ranges" of mainframes. All the machines in a range were supposed to be compatible in terms of instruction set architecture (ISA), but were engineered differently depending upon the target price/performance ratio and optimization for different kinds of computing, e.g., data processing versus scientific computing. Typically a data processing machine would not have a hardware floating point unit, instead floating point orders would be trapped and executed by software. Thus, depending upon the machine model some parts of the ISA could be virtual.

Some machines allowed the operating system designer to extend the instruction set architecture by software. For example in the Ferranti Atlas, the uppermost ten bits of a 48-bit machine instruction denoted which operation should be performed. If the most significant bit was set to zero, this was an ordinary machine instruction executed directly by the hardware. If the uppermost bit was set to one, this was an "extracode" and was implemented as a special kind of subroutine jump to a location in the fixed store (ROM), its address being determined by the other nine bits. Some extracodes were used to call mathematical procedures that would have been too complex to implement in hardware, for example sine, logarithm, and square root. Others were designated as "supervisor" functions, which invoked operating system procedures. Typical examples would be "Print the specified character on the specified stream" or "Read a block of 512 words from logical tape N".

It is interesting to note that in the early days there was no discussion of using dynamic binary translation to replace inconvenient instructions or implement extracodes, whereas today binary translation is widely used, particularly in the context of sandboxing dangerous code such as device drivers or browser scripts. I attribute this to the fact that in early systems addresses spaces were much smaller and addressing modes more restrictive than in modern architectures. We forget how liberating the move to 32 then 64 bit, sparse addressing has been.

The next evolution in the application environment came with the move to writing programs in high-level languages, as exemplified by UNIX – B. Kernighan and R. Pike, *The Unix Programming Environment*, Prentice Hall, 1984. UNIX applications were written in the C language and the interface to the operating system was instantiated through a standard library of C subroutines. Moving the operating system interface up to this level had the benefit that the subroutines could be structured to offer convenient and consistent abstractions to the user, whereas lower level operating system calls could be simplified to the minimum set required to support the library. In effect a lot of code that in earlier systems had been inside the operating system boundary was lifted into the user process with a significant saving in the number of (costly) system calls made. Compilers for other languages also targeted the UNIX standard libraries as their interface (indeed often they were compiled first to C, then from C to the target machine code).

Since the programming language defined the application environment it made applications easy to port between versions of UNIX on different processors simply by recompiling them and enabled alternative operating system structures to be explored by the operating systems community. It was these advantages that established UNIX as the dominant operating system of the workstation era and its continued existence to the present day.

Less frequently commented upon is the simplification that UNIX bought to virtual devices. Prior to UNIX most operating systems had record-based input-output and programmers had to juggle with device block sizes, fixed versus variable length records, blocking factors, etc., etc. One only has to read a COBOL language specification to see what a tangle this could be. UNIX took the view that the stream of bytes was the abstraction of input-output most useful to applications and that all the issues of blocking factors and so on should be dealt with by the library subroutines. The abstraction unified devices, files and pipes. (UNIX took the idea from Multics where files were "outform" memory segments and it was a matter for programs to decide how to put any finer grained structure on a file, and for simple text, "stream of bytes" was sufficient. However because files were segments their length was limited by the maximum segment size and large data sets/documents were awkwardly handled by splitting the content over several files. UNIX did not copy the Multics single level store concept and in consequence files can be much larger than segments and were read by device READ and WRITE functions rather than by memory mapping. Later versions of UNIX do have memory mapping, but is more generally used to access apportion of a file using a moveable "window".   Ironically in the CAP operating system we also used windowing to support long files, but our "stream manager" abstraction enforced record structuring, probably a consequence of the University Computing Service having an IBM mainframe at that time.)

In the last decade we have seen with languages like Java and associated object frameworks a move to running applications in an entirely virtual environment implemented by a portable simulator and relying on techniques such as just-in-time compilation for faster execution. In the era of plentiful processing and memory resources, developers electronic commerce and social media applications programmers are willing to burn these resources in return for rapid applications development, deployment and portability. These developments have mostly been reported outside of SOSP, although systems issues to do with sandboxing unchecked code and scripts downloaded from the Internet have been an important area of study.

Stepping back in time, with the advent of local area networks of workstations in the late 1970s there was much interest in extending the UNIX application environment to workstation clusters. Early systems focussed on file sharing by allowing remote file systems to be mounted locally, e.g., SUN's Network File System (NFS) and

Newcastle University's "Unix United" – J. Black, L. Marshall and B. Randell, "The Architecture of Unix United", *Proc. IEEE*, **75**, 5, May 1987, pp. 709-718. These systems gave location transparent access to files but system administration remained local to each machine, presenting challenges to managing access control (user ids, group ids) consistently across the cluster.

The Locus system was more ambitious, aiming to deliver a single system image across a cluster of workstations. Locus provided location transparent file sharing, transparent process migration for load balancing and replication transparency for fault tolerance – G. Popek, B. Walker, J. Chow, D. Edwards, D. C. Kline, G. Rudisin, and G. Thiel, G., "LOCUS a network transparent, high reliability distributed system", *Proc. Eighth Symposium on Operating Systems* Principles, 1981, Published in *ACM SIGOPS Operating Systems Review*, **15**, 5, Nov. 1981, pp. 169-177.

Locus was a successful system but was limited by scaling constraints to tens of nodes at most. The initiative moved towards more loosely coupled systems such as MIT's Athena system (J. Arfman and P. Roden, "Project Athena – supporting distributed computing at MIT", *IBM Systems Journal*, 31, 3, 1992, pp. 550-563) and CMU's Andrew system (J. Morris, M. Satyanarayanan, M. Connor, J. Howard, D. Rosenthal and F. Smith, "Andrew: a distributed personal computing environment", *Comm. ACM*, **29**, 3, Mar. 1986, pp. 184-201) designed to scale to thousands of nodes.

A specific challenge for single image distributed UNIXes was the handling of the file block cache and file-to-memory mapping. On a single machine all processes share the file block cache, and if the same block is memory mapped by two or more processes they see each other's updates. This behaviour is relied upon by some important UNIX applications and databases. In the case of a distributed UNIX, the block cache has to be extended to become a distributed shared memory and this brings its own implementation challenges with respect to consistent updating and scalability.

Uniform, Location Independent Interfaces to Operating System Services

Protection is a central concern in operating systems design – it should not be possible for one process, whether by accident or design, to access resources belonging to another. In early systems this was accomplished by using process local names in systems calls – for example UNIX file descriptors are local to a process and are mapped by the operating system to inodes, the system internal names for files. This was fine when applications ran in a single process, but with the advent of inter-process communications it was sometimes desirable for a process to share resources with or donate then to one of its peers. Without system support this was impossible.

Capability-based addressing

Capabilities were originally invented by Jack Dennis and Earl van Horn in 1966 as an abstract concept for discussing protection in operating systems – J. Dennis and E. Van Horn, "Programming semantics for multiprogrammed computations", *Comm. ACM*, **9**, 3, March 1996, pp. 143-155. They were a adaptation of the early concept of "codewords" originating with the Rice Institute computer (E. Feustel, "The Rice research computer – a tagged architecture", Spring Joint Computer Conference, 1972). In the Rice machine data was tagged and codewords were pointers to data structures called "descriptors". The tag indicated the type of the descriptor that might be for a physical resource such as a block of store, a device, etc, or for an abstract resource implemented by the operating system. To achieve protection, creating or editing code words was essentially a privileged operation. Dennis and Van Horn simplified the idea by restricting codewords to reside in "C-list" (capability list) segments sidestepping the need for tagging and also proposed the inclusion of access control bits. (John Iliffe who developed the codeword further in his "Basic Language Machine" argues that capabilities are an over simplification of codewords, losing their object-oriented data abstraction facilities [private communication, 2015]).

Bob Fabry of the University of Chicago proposed a hardware implementation of capabilities and observed they could be used as context-independent addresses for virtual objects and therefore passed between protection domains to allow controlled sharing of resources – R. Fabry, "Capability-based addressing", *Comm. ACM*, **17**, 7, July 1974, pp. 403–412.

This idea was readily exploited capability-based operating systems such as the Cambridge CAP Computer which added to its hardware supported capabilities for memory segments abstract "software capabilities" to represent virtual objects. Hardware supported protected procedures acted "type managers" for these virtual objects – each type manager had the capability to unpick software capabilities of its type to access hidden information about the abstraction. Thus a "file" software capability had within it the system internal name of the file (the number of its first block on disc, i.e., equivalent to a UNIX inode) which only the file manager could unpick – R. Needham and R. Walker, "The Cambridge CAP computer and its protection system", *Proc. Sixth ACM Symposium on Operating Systems Principles*, published in *ACM Operating Systems Review*, **11**, 5, Nov. 1977, pp. 1-10. Likewise the HYDRA system from CMU with its software mechanisms for "rights amplification" to enable a type manager to unlock the content of an virtual object (E. Cohen, D. Jefferson, "Protection in the HYDRA operating system", *Proc. Fifth ACM Symposium on Operating Systems Principles*, published in *ACM Operating Systems Review*, 9, 5, Nov. 1975, pp. 141-160).

A little later, in the Amoeba system, Andy Tanenbaum showed how the capability-based addressing concept could be applied to build a distributed operating system: A. Tanenbaum, S. Mullender and R. Renesse, "Using sparse capabilities in a distributed operating system", *Proc. Sixth International Conf. on Distributed Systems*, IEEE, pp. 558-563, 1986. Amoeba capabilities were protected with a cryptographic checksum to prevent them being forged and this idea has been widely adopted ever since. In the

distributed case, a capability becomes a location transparent name for a system wide virtual object.
A different approach was taken by Plan 9 – in this Bell Labs system all system resources were abstracted as files, thus the UNIX directory system could be used for naming and access control, with the UNIX stream model used for invoking services through the familiar OPEN, CLOSE, READ, WRITE library functions. A custom protocol called P9 was developed to allow these operations to extend across a network. (https://en.wikipedia.org/wiki/Plan_9_from_Bell_Labs).

Invoking Remote Services

There was a huge debate in the late 1980s about the design of protocols for invoking remote services. One camp was in favour of synchronous remote procedure call (RPC) the other for more asynchronous message passing. The argument for RPC was based on it being a natural extension of the threads and procedures/monitors model of most contemporary concurrent programming languages. Work by Andrew Birrell and others at Xerox PARC demonstrated that RPC could be extended to "network objects" and that a bare metal implementation of RPC could match the performance of asynchronous protocols – A. Birrell, G. Nelson, S. Owicki and E. Wobber, "Network objects", *Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993, pp. 217-230; A. Birrell and B. Nelson, "Implementing remote procedure calls", *ACM Trans. On Computer Systems*, **2**, 1, Feb. 1984, pp. 39-59. Others argued that message passing layered over an optimized byte stream transport protocol such as TCP/IP (IETF RFP 793, "Transmission control protocol specification") was better especially for bulk transfer of data, as for example in a remote windowing systems such as X-Windows.

The same debate had arisen five years previously in the context of designing operating systems – was threading and shared memory monitors a better model than message passing for inter-process communication? Lauer and Needham showed that, in principle, the two models were equivalent – H. Lauer and Needham, R. "On the Duality of Operating Systems Structures," in *Proc. Second International Symposium on Operating Systems*, INRIA, Oct. 1978, reprinted in *ACM Operating Systems Review*, **13**,2 April 1979, pp. 3-19. In the distributed systems context the asynchronous model has come to dominate. In part this is due to other factors such as network firewalls restricting the use other than TCP/IP and other "standard" protocols and the networking community having optimized TCP/IP as a universal transport for wide-area communications. Most RPC protocols intended for use across the Internet are nowadays layered above TCP/IP, as for example the World Wide Web consortium's Simple Object Access Protocol (SOAP) - http://www.w3.org/TR/soap/ and Java Remote Method Invocation (RMI) – http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

Active versus Passive Remote Objects

A different tussle over remote object invocation semantics took place outside of the SIGOPS/SOSP community, instead in the Object Management Group when defining their "Common Object Request Broker" (CORBA) specifications (http://www.omg.org). Here the debate was between those who saw objects as persistent active entities, similar to actors (C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence", *Proc. Third International Joint Conference on Artificial Intelligence*, Morgan Kaufman, 1973, pp. 235-245) versus those who saw objects as passive entities stored in a file system or database. For the former, a single name was sufficient to locate and invoke the object; for the latter a two-part name, the first part leading to a suitable "object manager" and the second part identifying the specific object was required. In both cases the interaction model was RPC. The eventual CORBA server side "Portable Object Adaptor" specification allowed for both options by allowing the adaptor to support different policies for object activation and deactivation. This achieved client side transparency but a programmer building a server side object has to know the specifics of the POA they are using.

System Models

There was a big argument in the 1980s as the PC revolution got underway about how to interface the PC with the network. The whole concept of the PC was a step back from the mainframe as a centrally managed shared facility: PC users wanted the autonomy of having their machine totally under their own control, with the freedom to choose what operating systems and applications they ran. But as PCs became more widespread and were connected to local area networks, users wanted to be able to share data (files) and expensive resources such as high quality printers. They turned to the model developed by Xerox PARC in the late 1970s as their vision of the future office: users at PARC had self-sufficient autonomous workstations (Alto, Dorado, Dandelion, Dragon etc) augmented with shared file servers, print servers, and an authentication server. A DNS-like name resolver was used to advertise servers to clients. This system model was followed by early PC networks, for example Novell Netware (https://en.wikipedia.org/wiki/NetWare), to bring "file and print" services to PC desktop clients. This in turn stimulated the development of PC-based server class machines as well as desktops, which over time stole the market from the minicomputer-based workstations that had dominated the 1980s.

Universities looking to make computing accessible to large student populations, had stayed closer to the computer utility vision that had driven the development of Multics at MIT – F. Corbató, and V. Vyssotsky, "Introduction and overview of the Multics system", *AFIPS Conference Proceedings*, **27**, pp. 185-196, 1965. MIT with the Athena system (launched 1983), CMU with the Andrew system (launched 1982) and

Cambridge with the Cambridge Distributed System (R. Needham and A. Herbert, *The Cambridge Distributed Computing System*. Addison Wesley, 1983). All explored a "thin client" model: the assumption was that users would be equipped with simple machines – dumb terminals or small PCs and significant computing would be carried out on a bank of workstation or larger class server machines in a centrally managed data centre. In the case of dumb terminal access, all computing was performed centrally; with PC clients simple tasks such as editing and email could be done on the client, but more demanding applications were run centrally. This model required that the client terminal be virtualized. In the case of dumb terminals established protocols like Telnet were sufficient but with the deployment of bit-mapped graphics displays a more advanced capability was required. Both MIT and CMU developed remote windowing protocols, with MIT's X Window system becoming the best known – R. Schiefler and J. Gettys, "The X window system", *ACM Trans. On Graphics*, **5**, 2, April 1986, pp. 79-109. UNIX was the popular choice for the server operating system and significant effort was invested in building shared file systems, directory and user authentication systems that could operate at the scale of thousands of simultaneous users. However, in distinction to Locus, these systems were willing adapt the UNIX programming environment and supporting services to fit their more loosely coupled model: there was no desire to provide the illusion of running on a single image system.

Thin client models were also created by industry, a notable example being Citrix's WinFrame (1995) that offered a thin client model for running Microsoft Windows desktops remotely from simple devices called "WinTerms" - https://en.wikipedia.org/wiki/Citrix_Systems#Products. Citrix developed their own remote windowing protocol for called ICA, optimized originally for low bandwidth (but fixed latency) dialup connections. Later the ICA protocol was extended to work on the Internet, which generally offered higher bandwidth but (very) variable latency. Microsoft in due course offered a thin client system, called Terminal Server (subsequently renamed Remote Desktop Services) and RDP protocol – https://en.wikipedia.org/wiki/Remote_Desktop_Services. X, ICA and RDP all still persist and have been joined by Real Network's RealVNC (2002) – https://www.realvnc.com.

As desktop PCs grew in size and became less expensive the economic benefits of the computer utility model were eroded and it fell into gentle decline, only to be re-awakened with the advent of mobile devices such as smartphones and tablet computers and so-called "Cloud Computing". In this incarnation however the benefits are less about the economics of devices as "thin clients" and more about "the cloud" acting as a global repository for users data and supporting long-running applications. By exporting computation to data centre servers devices can conserve energy, extending battery life. By keeping the master copies of user's data in the cloud devices users can migrate seamless from desktop to phone to tablet with updates automatically propagating between them.

Acknowledgements

Peter Denning, Franz Kaashoek and Mike Schroeder have kindly read through several previous versions of this document correcting my errors and pointing me at important work that I was either unaware of or had forgotten!