PROSE I/O

Eric Van Hensbergen
Ken Goss
{bergevan,ksgoss}@us.ibm.com

IBM Research
Austin, TX 78750
USA

Motivation

Abstractions seem to be simultaneously the principal goal and bane of modern computer science[20]. This is particularly true in modern systems in which the system software stack has become a monumental tower of babel. At its foundation we have the virtualization engine or hypervisor, multiplexing system resources between multiple logical partitions. Each of these logical partitions is running an operating system which facilitates sharing of resources between distinct processes and also provides system service abstractions for file systems, networking, and process control. On top of the operating system we have a set of facilities for distributed computing such as a message-passing-toolkit (MPI)[21] or the OpenMP[22] libraries. On top of the distributed system layer we have virtual machines[23][24][25], providing yet another layer of abstraction, system services, and resource scheduling. On server systems a middleware layer [26] runs on top of the virtual machine providing another duplicate set of system services and yet another layer of abstraction. On top of the middleware runs the application, which itself may be an interpreter or execution engine (such as one for an XML business process) [27].

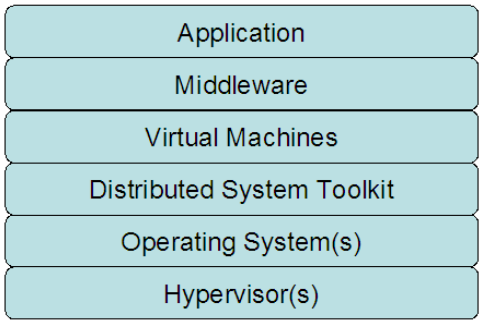


Figure 1. The modern software stack

This is a worst case view of the modern software stack, but it is a surprisingly prevalent one within service-oriented-architectures [28]. Each layer has its own resource allocation and arbitration logic. For example, there are separate time-sharing schedulers present in each layer. Each layer of abstraction hides its implementation details from the layer above it, making coordinated resource management impossible. What results is a debugging and performance tuning nightmare, with each layer adding both overhead and interference to the other layers. Additionally, the various services provided by each layer are usually immutable. Developers or users can not easily override the scheduler or memory allocation mechanisms of underlying layers. This is particularly true within the operating system layer, where privileged accounts are necessary to even interact with system services.

Microkernels [29][30] attempted to solve this problem by moving operating system services into user-space. However, provisioning and configuration of these services was still a privileged operation. Plan 9's [31] user-composable name spaces provides a different approach allowing user replacement of typical system services -- but doesn't allow user control of more complex system services such as memory allocation or scheduling. IBM's research operating system K42[32], allowed hot-swapping of these lower level kernel services, but provided a Linux API/ABI abstraction which ultimately limited its flexibility. The problem was probably best approached by MIT's Exokernel[5] research operating system which limited itself to straightforward multiplexing of hardware resources, and relied upon optional libraries (libOS) to provide familiar operating system abstractions. Developer's could choose which library instance to link their application against, or "roll their own" service or abstraction. This allows for an unprecedented degree of flexibility in the implementation of operating system abstractions, allowing application specific customizations and optimizations.

Microkernels, Plan 9, K42, and Exokernels never saw mainstream use principally because of their limited hardware platform support and lack of ability to run popular end-user applications. What seemed most desirable was a mainstream environment which could be used for the user environment and which could be "pushed out of the way" for execution of specialized applications which have their own system abstractions and services. Having the ability to leverage certain services and resources from the mainstream operating system would give developers the ability to focus effort on the handling of resources and services most critical to their application. Such a customizable environment would be particularly attractive within specialized workload domains such as high-performance computing and real-time systems.

Design

We built upon the Exokernel idea by utilizing virtualization to implement resource multiplexing, and specialized application kernels with operating system services linked in as libraries (libOS). This allowed us to take advantage of hardware partitioning features present in the PowerPC as well as more recent Intel and AMD processors. The use of preexisting virtualization solutions such as IBM's Virtualization Engine[13][15] and Xen[1] allowed us to run mainstream operating systems alongside our libOS partitions giving end-users familiar development and use environments. Additionally, hardware virtualization support on certain chip architectures[33] lets us configure logical partitioned dramatically differently -- for example, we are able to run libOS partitions in real-mode with translation disabled avoiding TLB miss penalties. Finally hardware virtualization lets us dedicate processor cores to specific applications providing increased levels of determinism by removing OS-related and I/O interference to computation.

Instead of requiring all device support be present in each libOS partition we leverage preexisting drivers running on the mainstream operating system partition or drivers running in specialized virtual I/O partitions. 9p[34], a simple, flexible communication protocol is used between partitions to provide access to disk, network, and other hardware resources. System services can be remoted in a similar fashion allowing peer partitions to provide file systems, networking stacks, and application services. In this manner a libOS application may chose to use a local custom device driver or service, or remote that service to another partition. Access to these hardware resources or system services can be provided by multiple peer partitions allowing for the possibility of redundancy and recovery during partition failures. The protocol used to communicate between partitions should be generic enough to be usable across the network allowing the same infrastructure to be deployed on scale out clusters of systems as well as large-scale SMP systems.

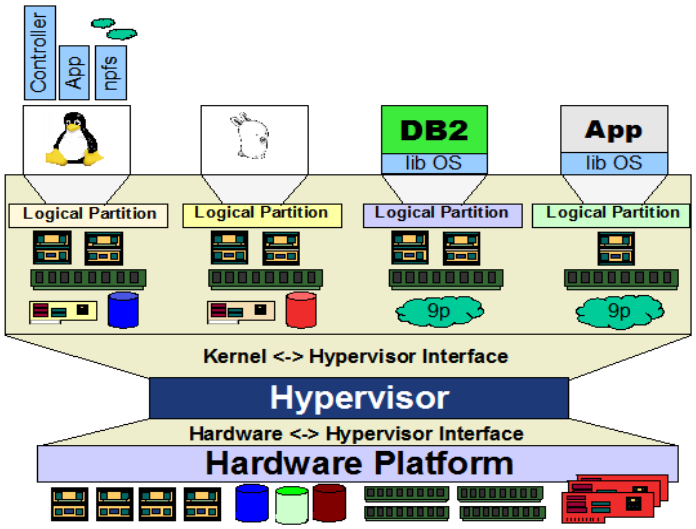


Figure 2: PROSE System Software Stack

Implementation

A hypervisor, the software agents which manage the logical partitioning of the hardware, form the foundation of our system software stack. It provides high-level memory, page table and interrupt management as well as partition scheduling policies. We based our initial prototype on the architecture employed by the IBM research hypervisor, *rHype*[13]. rHype is a small (~30k lines of code), low-latency, modular, multi-platform (supports x86 and PowerPC) para-virtualization engine. Logical partitions (LPARs) access rHype services through a "system call" like mechanism known as a *hcall*. We have also ported our libOS framework to run on the Xen hypervisor on the PowerPC platform.

As mentioned previously, a mainstream operating system, Linux, provides the primary user interface to launching, monitoring and controlling libOS partitions. This operating system partition is designated the "controller partition" or Dom0, and has special authority to allocate and control the partitioned resources of the machine. A special wrapper script (varying depending on the underlying hypervisor) is used to manage the creation of new logical partitions, starting necessary support applications, and redirecting standard I/O.

Within rHype a special device is used to issue hcalls and communicate with peer partitions. This device can be memory mapped to provide a window into the private memory of the libOS partitions. With our Xen implementation, we use the Xentools API to setup a shared memory window into the application partition's memory. Ring buffers are established within the libOS partition and can be used for communication with peer partitions including the controller. In the prototype, these ring buffers are statically compiled within the application. The wrapper scripts simply examine the application image's symbol information (using *nm(1)*) to determine the location of the ring buffer within the application before launching it. Since the prototype I/O is handled by user-space applications on the controller, a simple poll methodology is used to detect I/O activity.

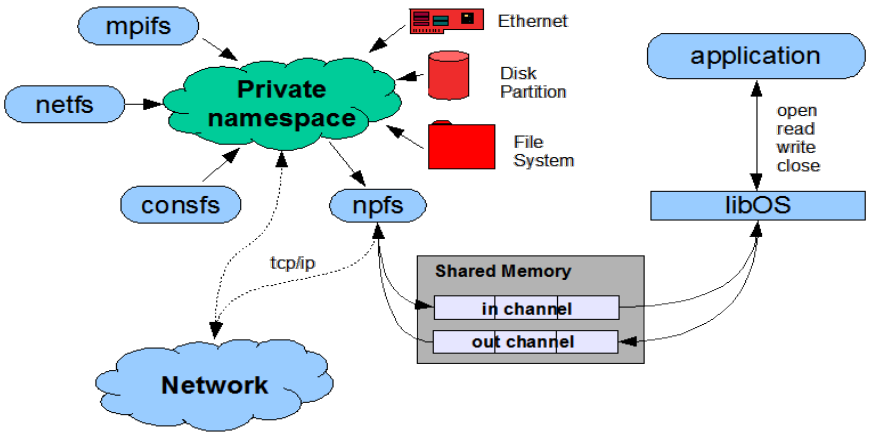


Figure 3: PROSE I/O Prototype

Resources and services are provided to libOS partitions via the 9p resource sharing protocol, originally developed as part of the Plan 9 operating system. 9p is the protocol which runs over the previously mentioned I/O channels. Some resources such as the controller's file system and access to its character and block devices may be provided by 9p directly. However, most resources are gatewayed by synthetic file server applications. These servers export hierarchical file systems representing system services and resources such as the TCP/IP stack, console I/O, or windowing system. More information on system services and resources via Plan 9 synthetic file systems can be found in the various Plan 9 technical papers and documentation[9].

We had implemented two stand-alone synthetic file system gateways as part of the file system prototype. There is a network gateway which provides socket services in a fashion similar to the Plan 9 devip[9] device. We also have a partial implementation of the cons[9] device which provides console interactions with the libOS partition as well as a libOS access to system information such as time. The synthetic file systems are organized within the global Linux name space by mounting them using the v9fs Linux kernel module[16]. This name space is then re-exported to the libOS partition by the npfs[36] application which gateways Posix file system name spaces to 9p.

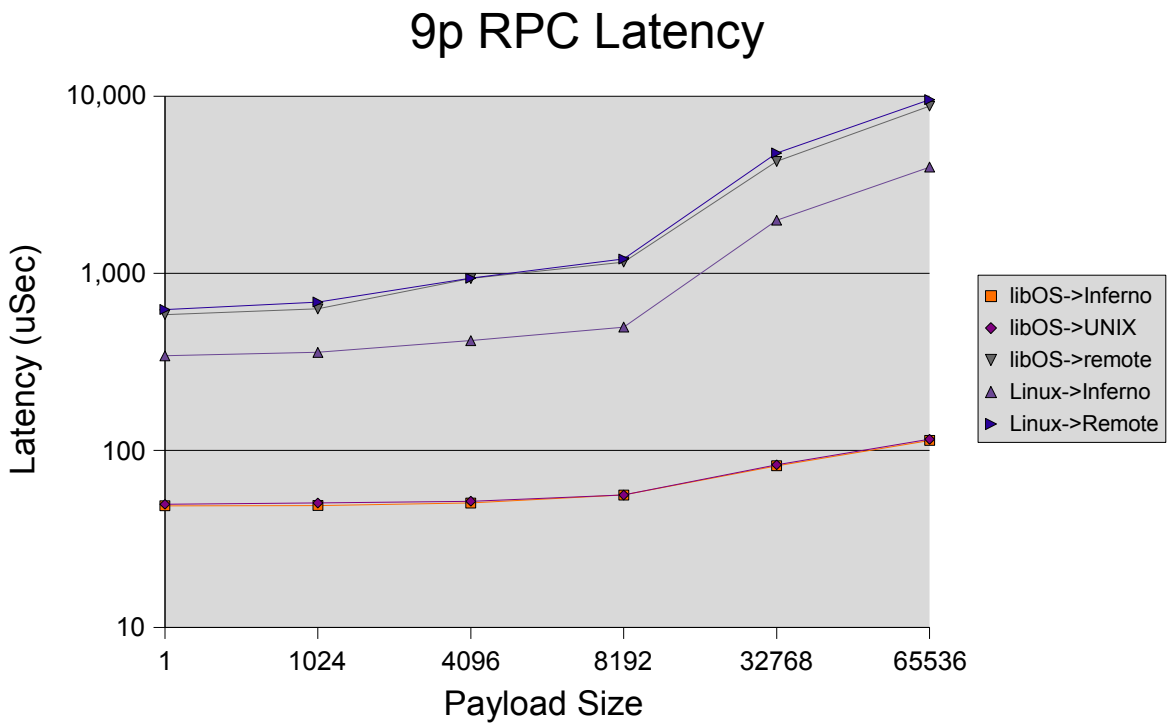
Stability and performance concerns, along with the desire to quickly support a broader range of service and resource gateways, led us to using the Inferno [35] environment as a replacement for the v9fs/npfs stack. Inferno is a distributed operating system which also runs as hosted middleware on many more traditional operating systems (Linux, Windows, etc.). It also uses the 9p protocol for resource sharing, and in its hosted environment contains a great number of gateway synthetic file systems to the underlying operating-system's resources and services. We ported the Inferno hosted environment to Linux/PPC and added a new device file system (devxenchan) which is used to manage shared memory channels to the underlying application partitions. Like the Plan 9 network statcks (such as devip), devxenchan operates as a 'clone' file system – allowing a single inferno instance to communicate with multiple child partitions.

A side benefit of using Inferno as a Dom0 infrastructure was a natural path to cluster solutions. The Dom0 partitions can export control interfaces over TCP/IP which can then be leveraged by a front-end node. By mounting resources (such as file system or networking stacks) from the front-end node, a whole cluster of machines can be managed as a single node. This provides a very attractive solution for bladed-clusters and other scale-out systems.

Evaluation

Previous papers have used micro benchmarks to explore performance benefits [37] from using libOS partitions for HPC applications [6] and the ability of hypervisors to control OS induced noise[16]. In this paper we focus on the performance of inter-partition I/O within the prototype.

We have performed preliminary evaluations using identically configured IBM Blade Center JS20 server blades running XenPPC, Dom0 Linux, and DomU's running LibOS PROSE test applications. The first thing we measured was the end-to-end latency of a 9p RPC message. In order to collect the results we generated 10,000 writes of a particular payload size to /dev/null. Each measurement was taken ten times and the average is presented. There was very little variance (< 1%) in the results collected. We took measurements of the libOS application writing to Inferno's /dev/null (shown as the libOS->Inferno data line) as well as writing to the underlying Linux /dev/null (shown as the libOS->UNIX data line). We also took measurements writing from the libOS through a transitive mount on the Dom0 Inferno to a remote Inferno instance (shown as libOS->remote) to show the relative performance of accessing resources on a front-end node. As points of comparison we include measurements taken from Linux to a local and remote Inferno server using v9fs and tcp/ip (labeled Linux->Inferno, and Linux->Remote respectively).



As can be seen from the graph, the latency over the shared memory channel is extremely low, although copy-overhead almost doubles the latency for larger payloads. The primary reason for the low-latency is the extremely greedy polling algorithm used for the shared memory channel. An unfortunate side-effect of this polling is a constant high load on the Dom0 node. Accessing resources over transitive mounts increases the latency proportionately with any access of resources over traditional tcp/ip sockets. The overhead of multiple copies of payload really start to degrade performance above 8k payloads suggesting that significant benefit can be gained by implementing a zero-copy path for payloads traversing shared memory.

Future Work

Our experiences with the initial prototype have identified multiple areas for further exploration and improvement. Our polled-communication mechanism works well when you can dedicate a processor core (or multiple processor cores) to an I/O partition -- however, in situations where you are sharing a computational core an interrupt driven mechanism would be much more preferable. The implementation of interrupt driven I/O varies from hypervisor to hypervisor and can either be built into the kernel or gatewayed to user space.

On PAPR [33] compliant architectures such as the PowerPC, inter-partition interrupt-driven I/O is managed by CR/Q channels. An example CR/Q driver is the Linux Vscsi driver which remotes SCSI operations over shared memory between partitions. Our current plan is to abstract out the transport layer from the existing Vscsi [38] driver and create a generic CR/Q service for Linux. We can then build 9p communication channels on top of the generic service, and even expose it to user-space as a new physical network using Linux packet sockets. Similar signaling mechanisms exist in other para-virtualization environments such as Xen.

A major performance problem with our existing transport solution in the number of copies it introduces with each transfer of the data. This is particularly pronounced with the use of v9fs to mount synthetic file systems before their re-export across the shared memory link. Data going to a synthetic file systems such as the TCP/IP stack makes 4 context switches and copies the data 8 times. If the TCP/IP stack service is on a remote system this number increases. Almost all of these copies and most of the context switches are unnecessary. We can avoid them by building the core of the 9p server into the kernel module. This serves the dual purpose of allowing it a tighter coupling to transports and allowing us to bypass paths through user space when talking with the v9fs driver. Further performance optimization could be gained by moving kernel service gateways (such at the TCP/IP stack) into the kernel as well.

Another source of overhead is our use of transitive mounts. In other words, we compose a libOS partition's name space on the "controller" by mounting services from both local and remote file servers, and then we re-export the name space with npfs or inferno. While this provides a very elegant configuration and management mechanism, it does so at the cost of performance. If libOS partitions were able to compose their own name spaces directly, and additional copy and context switch can be saved. One adjunct to this idea is the idea of using transitive I/O channels instead of transitive file system mounts. This would allow the "controller" to compose a directory of I/O channels to various system services (whether local or remote) and the libOS partition would attach to these directly. This could provide a compromise between elegance and performance.

Since 9p can also be used across network interconnects, resources and services can be located anywhere within the network. By using a synthetic file server to manage logical partitions these same mechanisms can be used to allow libOS

partition creation and control across the entire cluster. Such an interface could be built as an extension to LANL's xcpu service [39].

We have performed our evaluation on logical partitioned systems such as Xen and rHype, but we have yet to a study comparing our I/O methodology with the native I/O virtualization methods present with Xen and PAPR. We'd also like to evaluate our mechanisms on heterogeneous systems such as the Cell processor, or statically partitioned environments such as Blue Gene. Another area of exploration is to look at I/O devices directly providing partitioned 9p interfaces to their resource. This could be used for storage controllers, TCP/IP offload, or graphics controllers.

Acknowledgment

This work would not be possible without the contributions of Jimi Xenidis, Michal Ostrowski, Orran Krieger, and the rest of the rHype, XenPPC, and CSO teams. This work was supported in part by the Defense Advanced Research Projects Agency under contract no. NBCH30390004.

References

[1] Xen 2002, Paul R. Barham, Boris Dragovic, Keir A. Fraser, and et al., ucam-cl-tr-553, January 2003, University of Cambridge, Computer Laboratory.

[2] Cellular disco: resource management using virtual clusters on shared-memory multiprocessors, Kingshuk Govil, Dan Teodosiu, Huang Yongqiang, and Mendel Rosenblum, 2000, ACM Transactions on Computer Systems, vol 18:3, 229-262.

[3] Linux Devfs (Device File System FAQ), http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html.

[4] Disco: Running Commodity Operating Systems on Scalable Multiprocessors, Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum, 1997, ACM Transactions on Computer Systems, vol 15:4 , 412-447.

[5] Exokernel: An operating system architecture for application-level resource management, Dawson R. Engler, M. Frans Kaashoek, and James O Toole, Jr., 1995, Proceedings 15th Symposium on Operating Systems Principles, 251-267.

[6] High Performance Computing Challenge, http://icl.cs.utk.edu/hpcc/.

[7] IBM PowerPC Full System Simulator, IBM Alphaworks, http://www.alphaworks.ibm.com/tech/systemsim970

[8] Linux Kernel Development, Robert Love, 2003.

[9] Plan 9 Programmer s Manual, Volume 1, AT & T Bell Laboratories, Murray Hill, NJ, 1995.

[10] AMD Virtualization Codenamed “Pacifica” Technology, Secure Virtual Machine Architecture Reference Manual, AMD, May 2005

[11] Linux Kernel Procfs Guide, http://www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html.

[12] Partitioned Reliable Operating System Environment Home Page, http://www.research.ibm.com/prose

[13] IBM Research Hypervisor home page, http://www.research.ibm.com/hypervisor

[14] An Introduction To Virtualization, Amit Singh, 2004.

[15] Server Consolidation Using POWER5 Virtualization White Paper, H. Tsao and B. Olszewski, 2004.

[16] Grave Robbers from Outer Space: Using 9P2000 under Linux, in the Proceedings of Freenix, 2005.

[17] The Effect of Virtualization on OS Interference, Eric Van Hensbergen, in the Proceedings of the first Workshop on Operating System Interface in High Performance Applications, St. Louis, MO, 2005.

[18] Enhanced Virtualization on Intel Architecture-based Server, Intel Solutions White Paper, March 2005.

[19] Denali: Lightweight virtual machines for distributed and networked application, A. Whitaker, M. Shaw, and S. Gribble, 2002, Proceedings of the USENIX Annual Technical Conference.

[20] “I did it for you all.” Anonymous. http://www.chunder.com/text/ididit.html

[21] MPI: The Complete Reference, M Snir, et. al. MIT Press, Cambridge, MA, USA

[22] Parallel Programming in OpenMP. R. Chandra, et. al., Morgan Kaufmann, 2000.

[23] The Design of the Inferno virtual machine. P. Winterbottom, et. al. IEEE Hot Chips, 1997.

[24] Java Virtual machine Specification. T. Lindholm, et. al. Addison-Wesley, Boston, MA USA. 1999.

[25] Introducing Microsoft .NET, D.S. Platt, Microsoft Press, Redmond, WA. 2001

[26] IBM Websphere. http://www.ibm.com/websphere. 2003.

[27] Provisioning On Demand: Introducing Tivoli Intelligent Orchestrator. E. Manoel, et. al. Dec 2003.

[28] Service Oriented Architecture: A Field Guide to Integrating XML and Web Services. T. Erl. Prentice Hall PTR, Upper Saddle River, NJ. 2004.

[29] Microkernel Operating System Architecture and Mach. Black, D.L., et. al. Journal of Information Processing. March 1992.

[30] The Performance of Micro-Kernel-based Systems. H. Hartig, et. al. In the Proceedings of the 16th SOSP. 1997

[31] Plan 9 from Bell Labs. R. Pike, et. al. Computing Systems, 1995.

[32] Experiences with K42, an open-source, Linux-compatible, scalable operating-system kernel. J. Appavoo, et. al. IBM Systems Journal, 2005.

[33] Power Architecture Platform Requirements, Power.org, 2006.

[34] 9p Request for Comments, http://v9fs.sf.net/rfc

[35] The Inferno Operating System, SM Dorward, et. al., Bell Labs Technical Journal, Vol 2., No 1., Winter 1997, pp 5-18.

[36] NPFS: 9p File Server, http://www.sourceforge.net/projects/npfs, 2006.

[37] PROSE: Partitioned Reliable Operating System Environment, E. Van Hensbergen, Operating Systems Review, Volume 40, Number 2, April 2006.

[38] vSCSI. Linux Kernel Mailing List, Feb. 2006.

[39] The Xcpu Cluster Management Framework, Latchesar Ionkov, et. al. To be published in the proceedings of the 1st annual international workshop on Plan 9.