SAND2009-0245 C
SAND2009-0245C

An Extensible Operating System Design for Large-Scale Parallel Machines

Rolf Riesen Kurt Ferreira
Sandia National Laboratories*
{rolf,kbferre}@sandia.gov

Abstract

Running untrusted user-level code inside an operating system kernel has been studied in the 1990's but has not really caught on. We believe the time has come to resurrect kernel extensions for operating systems that run on highly-parallel clusters and supercomputers. The reason is that the usage model for these machines differs significantly from a desktop machine or a server. In addition, vendors are starting to add features, such as floating-point accelerators, multicore processors, and reconfigurable compute elements. An operating system for such machines must be adaptable to the requirements of specific applications and provide abstractions to access next-generation hardware features, without sacrificing performance or scalability.

1. Introduction

Large-scale, high-performance clusters and supercomputers used for scientific parallel computing require specialized operating systems (Brightwell et al. 2003). Usually, these machines run a single, parallel application that is spread across many or all the nodes of a system. Each process that is part of that application is assigned to a CPU (core) and "owns" it for the duration of the run. That means that during that run, no other processes that are not part of that application will be assigned to these CPUs. Multiple applications *space-share* a parallel machine.

OS-noise has been identified as a major culprit that inhibits scalability (Petrini et al. 2003; Ferreira et al. 2008). Parallel applications exchange messages and often need to wait for the data before they can proceed. If one of the processes is delayed because the local operating system is busy running other processes or doing housekeeping tasks, it will delay the entire application. As parallelism increases, the likelihood that any one operating system instance is not currently running the parallel application process, increases as well. That means the parallel application is slowed down as it is run on more nodes and resources are wasted as hundreds or thousands of nodes wait for the straggler.

All the resources of a node: CPUs, network interfaces, and memory, are allocated to processes of the same application. While memory protection between processes is still useful for debugging, it is no longer strictly necessary. In fact, it might be performance beneficial to let processes running on multiple cores freely share the memory of a single node (Brightwell et al. 2008). Policies for process control should also be determined by the application itself. Therefore, within a node, less protection is needed than what typical operating systems provide.

However, some protection mechanism still need to be enforced by the operating system. For example, the application should have full control over the network interface (OS bypass) so it can be managed as efficiently as possible, but trusted header information, such as the source node ID and process ID of a message, must be under operating system control. In other words, the operating system should let an application manage the nodes that have been allocated to it, while still protecting the resources of the machine that belong to other applications.

Many clusters, especially the larger ones, and most supercomputers employ a parallel file system whose storage devices are external to the machine or attached to dedicated I/O nodes. Compute nodes do not have local disks. Most other peripherals that are supported by desktop operating systems are missing as well. In fact, the main peripheral accessible to the application is a high-speed network interface sometimes directly attached to the memory bus. All I/O operations, remote memory access, and explicit message passing are handled by that one device.

This architecture limits the number of devices an operating system must support. Furthermore, many modern network interfaces are intelligent and interact with the appli-

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Submitted to HotOS XII

cation directly. Copying data through the operating system would have disastrous effects on network performance.

Because of these characteristics, parallel applications running at large scale have very specific demands of an operating system. In addition, the hardware to build clusters and supercomputers is changing and requires adaptation from the application and the operating system. The operating system is expected to match an application to the hardware it runs on as efficiently as possible.

Some of the new hardware features that require applications and operating systems to adapt are here already. One example is the use of multicore processors. Other features are not in production use yet, but are being discussed as potential performance booster for next-generation systems. Examples include attaching graphic processing units (GPUs) or other specialized processors, such as IBM's cell architecture, to general purpose CPUs to accelerate floating-point intensive calculations. More exotic devices, for example Field Programmable Gate Arrays (FPGAs) that can be reconfigured on the fly for a specific application need, or Processor in Memory (PIM) devices that could help alleviate memory bus throughput demands, are on the horizon.

An operating system must provide abstractions that unify several of these technologies and make them accessible to portable applications. Such applications cannot be re-written for every possible new feature a vendor offers. Rather, it is the operating system's role to manage the new resource, and make them available to the application.

A look at the list at `www.top500.org` reveals that the number of processors built into the five-hundred fastest systems in the world, is increasing every year. With a higher component count the likelihood of a hardware failure increases. This reduces the Meantime Between Failures (MTBF) for large scale applications.

A similar argument can be made for the software side. The likelihood that a subtle timing bug is triggered increases with the number of code instances running. A small operating system kernel is easier to debug, test, and reason about than a multi-million line, full-featured operating system. Furthermore, it is easier to restart or migrate a small operating system in case of a hardware failure or an early warning.

An earlier HotOS paper (Hunt et al. 2005) listed the challenges for next-generation systems as: dependability, security, system configuration, extension, and multi-processor programming. While that paper was written in the context of desktop operating systems, many of the points the authors make also apply to operating systems for high-end parallel computing. In particular, system configuration and multi-processor programming are areas that need to be addressed. We think that kernel extensions allow us to write small, simple, and scalable operating systems with the flexibility to adapt to the demands of future architectures and applications.

For these reasons we believe the time has come to revisit extensible operating systems and apply some of the techniques and lessons learned in the 1990's to high-end, parallel computing. We envision a very small kernel that provides base services and can be extended by the runtime system of the machine or by the application itself. Some of these extensions adapt the kernel to a given machine and are probably inserted during boot time by a trusted entity. Less trusted extensions can be inserted by the applications. These are only needed while the application is running and are meant to provide a better impedance match between the application and the underlying operating system and hardware.

We will explain our design ideas in the next section and discuss in Section 3 why we think these ideas are beneficial to high-end parallel computing platforms. We will look at related work and provide a summary at the end of the paper.

2. Nimble

Work on a kernel called Kitten that builds on our experiences with lightweight kernels (Wheat et al. 1994; Maccabe et al. 2004; Riesen et al. 2008) for massively parallel machines is currently under way. The goal of the Kitten project is to efficiently use the multicore resources that have begun to appear in modern parallel machines. In this section we describe Nimble, an extension infrastructure for a lightweight kernels such as Kitten.

2.1 Extensibility

Core operating system tasks, such as initializing devices, and simple process and memory management, will be provided by the lightweight kernel. In contrast to a full-features desktop or server operating system, lightweight kernels provide only rudimentary services. There is a single, or at most a couple, of processes running on each CPU core and there is no support for demand paging, kernel-level threads, TCP/IP, dynamic libraries and many other amenities that desktop users expect but limit performance and scalability.

Nevertheless, some applications may be willing to make performance and scalability sacrifices for a given feature. At other times in may not be practical to rewrite as service or a library on which an application depends. Additionally, providing just one more feature may not compromise performance or scalability, while making all of the available will cripple the operating system and the machine it runs on.

Therefore, we propose to use kernel extensions to customize the operating system to the specific hardware it is running on and adapt it to the currently running application. There are two types of extensions. *Trusted* kernel extensions have direct access to the hardware and are used by the system administrator and the runtime system to adapt the kernel to the hardware. This is typically done at boot time and is akin to loadable kernel modules that Linux provides.

Application-inserted kernel extensions are generally not trusted and do not have direct access to hardware or other privileged resources. However, remember that most of the node resources have been allocated to the application already. The kernel must enforce access policies to resources outside the node, but most of the node resources are managed by the application itself. This is typically done through a library that inserts a kernel extension on behalf of the application.

For example, an application-inserted kernel extension can augment the basic process scheduler present in the lightweight kernel to allow for active messages to force a context switch to a user-level handler as soon as they arrive. An application that consists of a single process per CPU core could run with an infinite time quantum.

An application could use kernel extensions to dedicate a CPU core to handle message-passing traffic from the network interface and run compute-intensive processes on the other cores. An application that is not communication intensive may use an interrupt-driven kernel extension to handle network interface requests, and use all CPU cores for computation.

Latency-sensitive operations, such as remote memory accesses or collective message-passing operations, could benefit from a kernel extension that handles some network requests in the kernel on behalf of the application instead of incurring a full context switch to run a user-level handler.

Instead of the kernel providing a slew of mechanisms, it provides only basic services and the ability to insert extensions that provide new mechanisms and set policy on behalf of the application.

2.2 Implementation

In the 1990's several methods to extend kernels were investigated. One that has not had very much attention is interpretation. An interpreter is relatively easy to write and it is easy to shield other parts of the kernel from code that is interpreted. For code from a trusted source, Nimble will disable access and privilege checks in the interpreter. This will yield a small performance advantage, but more importantly, it will allow trusted extensions to access and manipulate protected resources.

Another reason we are considering an interpreter is that we expect most extensions to be small, simple, and to only run for brief moments. For example, a process scheduler that needs to pick the next process to run from a pool of less than a handful, does not require a lot of instructions. The code to do the actual context switch is written in C, already resides in the lightweight kernel, and can be called by the interpreter. Code to initiate a data transfer through the network interface already exists as well. A kernel extension makes a single subroutine call to start the data transfer.

Techniques for fast interpretation have been studied for a long time and are still under investigation for today's byte code interpreters. One such technique, called threaded code (Bell 1973), is one we intend to pursue.

During code insertion the extension is threaded. Each instruction in the extension is converted to a subroutine call into the interpreter. This makes interpretation a two-step process. First, the instructions are examined and translated. In the second step, during the execution of an extension, execution proceeds along a thread that consists of various subroutines the interpreter provides.

When Nimble starts executing an extension, for each original statement in the extension, control flow will be redirected into the appropriate subroutine inside the interpreter. The subroutines contain the code that encapsulates the semantics of a given statement in the kernel extension.

The subroutines are written in C and compiled into Nimble. We intend to have two versions of most subroutines. One that performs access checks and another which does not.

Threaded code reduces the interpretation overhead to one or two assembly instruction per interpreted statement in the kernel extension. While many statements will be simple, such as loading a value, many are more complex and the overhead of interpretation will be negligible.

There are variations on threaded code. The method we described above is called subroutine threading. During the translation step the statements to be interpreted are translated into a series of CPU "call subroutine" instructions. It can be argued that this technique is not interpretation, since the call instructions are simply executed after the original statements have been translated. Depending on the CPU architecture, direct threaded coded may be faster. Direct threaded code is just a compact list of addresses. The interpreter reads these addresses and calls the subroutine where these addresses point to. Either of these techniques are faster than interpreting byte code (Ertl 1993; Klint 1979; Dewar 1975).

Nimble will add mechanisms to the lightweight kernel to insert and remove extensions, to call them for specific events, such as interrupts or application traps into the kernel, allow one extension to call another, and to limit the running time of an extension.

3. Discussion

Operating system kernel extensions have been extensively studied in the 1990's, but have not really caught on in mainstream desktop operating systems. We believe that one reason for that is that much of the extensibility was aimed at improving operating system performance by avoiding unnecessary kernel-to-user-level-transitions by executing user code in the kernel.

There are other techniques for that and machines have gotten fast enough for many tasks that were considered to be in need of improvement in the 1990's. We believe that kernel extensions have a place in high-end parallel computing for several reasons. One is that speed is still of primary concern here and that inefficiencies in the operating system can severely limit the scalability of a parallel machine. The other reason is that next-generation supercomputers will employ technologies – starting with multicore processors to attached floating-point engines – that will be difficult to exploit in a general purpose way.

Each application and programming model has its own specific needs. If a hardware resources and an access policy can be customized for a specific application, performance and scalability benefits will follow. In the type of machines we are considering, this is possible because they are space shared and whole sets of nodes are allocated to an application. Letting the application manage these resources is more efficient than providing general purpose mechanisms and policies. We need an operating system that can be used to manage the machine as a whole (allow for node allocation for example), but gets out of the way when an application wants to make use of the resources allocated to it.

Therefore, we want to give each application the opportunity to set its own resource allocation policy and add the specific features it needs in an operating system, while not being burdened by services it does not need and that could limit performance or scalability. Allowing applications to insert user-level code into the kernel seems an ideal way to achieve this flexibility.

Allowing user-level code to execute inside a lightweight kernel makes sense because the usage model and the requirements for high-performance parallel computers are quite different from the needs of a desktop user or even a server farm. Some aspects of embedded computing apply as well, but these systems, once deployed, are more static in nature than the application mix run on a parallel computer.

4. Related work

The idea of executing user code, an extension, inside the kernel has seen several incarnations. Sand-boxing, also called software-based fault isolation (Wahbe et al. 1993), is the idea of limiting data accesses to a certain segment of main memory. This is done by inserting code before potentially unsafe instructions that sets (or checks) the upper n bits of the address to a value that corresponds to a segment which the code is allowed to access. For our approach this may be too limiting, since we do want to make some memory mapped devices accessible to extensions, while preventing access to memory-mapped registers that must be protected.

The SPIN project (Bershad et al. 1994) used a trusted compiler to generate spindles that get inserted into the kernel. The spindles are digitally signed to ensure that they were generated by the trusted compiler. The runtime system for the chosen language and the cryptographic tools to verify the signatures would need to be available on each node. Depending on the source language chosen, this may be a significant amount of code that is statically linked with the parallel application.

Interpreters have been studied extensively and some of them have been embedded in kernels before. The BSD packet filter is one example (Mogul et al. 1987). Another is our work of adding a FORTH interpreter to the firmware of a Myrinet network interface (Wagner et al. 2004). We used that to improve the performance of collective MPI operations such as broadcast.

Interpreters are often considered to be too slow for system services. However, our extensions are small and perform simple tasks. It should be possible to gather the most often used constructs and sequences into a virtual machine which can be optimized to execute efficiently (Pittman 1987; Proebsting 1995). Then, using indirect threaded code or direct threaded code techniques, build an extremely fast interpreter (Bell 1973; Dewar 1975; Kogge 1982; Ertl 1993).

Several operating systems have made use of extensions. We already mentioned SPIN. Global Layer Unix (GLUnix) (Vahdat et al. 1994) used software-based fault isolation to move OS functionality into user level libraries. We want to move user code functionality into the kernel. The VINO kernel was designed to let applications specify the policies the kernel uses to manage resources. That is what we are interested in, but specifically for high-performance parallel environments, instead of database management systems for which VINO was designed.

In the μChoices operating system (Campbell and Tan 1995; Tan et al. 1995) agents can be inserted into the kernel. These agents are written in a simple, flexible scripting language similar to TCL, and are interpreted. Agents batch a series of system calls into a single procedure that requires only one trap into the kernel to be executed. Agents use existing kernel services and do not extend the functionality of the kernel or provide services that are not available at user level. Agents are a simple optimizations to eliminate the overhead of several system calls. We also need to mention the MIT Exo kernel (Engler et al. 1995; Engler and Kaashoek 1995). It attempts to lower the OS interface to the hardware level, eliminating all abstractions that traditional operating systems provide, and concentrates on multiplexing the available physical resources. This is similar to our lightweight kernels which provide only very basic services and rely on user-level libraries to implement other services. Nimble is meant to extend this concept and push some of that functionality back into the kernel when it is needed at run time.

Methods to safely execute untrusted code in a privileged environment are compared in (Small and Seltzer 1996).

5. Conclusions and future work

Lightweight kernels have proven successful in the past on Intel's Paragon and ASCI Red at Sandia National Laboratories, on Cray's XT-3 Red Storm, and on IBM's Blugene/L series of machines. These lightweight kernels are small and scalable, allow applications to get most of the available memory (without demand-paging) and run on tens of thousands of nodes in parallel.

As more applications are being ported to these kinds of machines, the demand for additional features and services increases. A modern operating system must provide some of these features without compromising scalability or efficiency.

We are working on a new lightweight kernel called Kitten that carries our experiences with large-scale parallel machines forward to machines with potentially hundreds of cores per node. We have started the design of Nimble which will be integrated into Kitten. Nimble will provide the infrastructure to let applications extend Kitten's functionality. These extensions are meant to provide additional services and provide access to next-generation hardware features.

References

James R. Bell. Threaded code. *Communications of the ACM*, 16 (6):370–372, June 1973.

Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report CSE-94-03-03, University of Washington, February 1994.

Ron Brightwell, Arthur B. Maccabe, and Rolf Riesen. On the appropriateness of commodity operating systems for large-scale, balanced computing systems. In *International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

Ron Brightwell, Kevin Pedretti, and Trammell Hudson. Smartmap: operating system support for efficient data sharing among processes on a multi-core processor. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

Roy H. Campbell and See-Mong Tan. μChoices: An object-oriented multimedia operating system. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*, pages 90–94, May 1995.

Robert B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of HotOS V*, May 1995.

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, 1993.

Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.

Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. Broad new OS research: Challenges and opportunities. In *Proceedings of the 10th conference on Hot Topics in Operating Systems (HotOS IV)*, 2005.

Paul Klint. Interpretation techniques. *Software Practice and Experience*, 11:963–973, 1979.

Peter M. Kogge. An architectural trail to threaded-code systems. *IEEE Computer*, pages 22–32, March 1982.

Arthur B. Maccabe, Patrick G. Bridges, Ron Brightwell, Rolf Riesen, and Trammell Hudson. Highly configurable operating systems for ultrascale systems. In *First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, pages 33–40, June 2004.

Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.

Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 150–152, June 1987.

Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, January 1995.

Rolf Riesen, Ron Brightwell, Patrick G. Bridges, Trammell Hudson, Arthur B. Maccabe, Patrick M. Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, to appear, September 2008.

Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *1996 USENIX Annual Technical Conference*, January 1996.

See-Mong Tan, David K. Raila, and Roy H. Campbell. An object-oriented nano-kernel for operating system hardware support. In *Proceedings of the Fourth International Workshop on Object Orientations in Operating Systems*, pages 220–223, August 1995.

Amin Vahdat, Douglas Ghormley, and Thomas Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report UCB CS-94-842, Computer Science Division, UC Berkeley, December 1994.

Adam Wagner, Hyun-Wook Jin, Dhabaleswar K. Panda, and Rolf Riesen. NIC-based offload of dynamic user-defined modules for Myrinet clusters. In *IEEE Cluster Computing*, pages 205–214, September 2004.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. *Scientific Programming*, 3:275–288, 1994.