Multicore OSes: Looking Forward from 1991, er, 2011

Multicore OSes: Looking Forward from 1991, er, 2011

David A. Holland and Margo I. Seltzer
Harvard University

Abstract

Upcoming multicore processors, with hundreds of cores or more in a single chip, require a degree of parallel scalability that is not currently available in today's system software. Based on prior experience in the supercomputing sector, the likely trend for multicore processors is away from shared memory and toward shared-nothing architectures based on message passing. In light of this, the lightweight messages and channels programming model, found among other places in Erlang, is likely the best way forward. This paper discusses what adopting this model entails, describes the architecture of an OS based on this model, and outlines a few likely implementation challenges.

1 Introduction

The recent advent of multicore processors has changed the rules for software development. Instead of a guaranteed speed boost, new hardware only provides increased parallelism. Software only gets faster if it can take advantage, but parallel code is difficult and expensive to write and may not produce satisfactory results. By great effort Solaris has been made to scale to perhaps 128 cores; however, chips with hundreds of cores or more seem likely in the moderately near future [6, 7, 17].

Some think this a crisis [10, 23, 18]. However, there is one widely overlooked but fundamental detail: we have been here before. From a hardware perspective, there are assorted differences between multiple cores on a single chip and multiple chips on a single motherboard. To software, though, a multicore machine is nearly indistinguishable from a multiprocessor machine. Some cache pressure issues arise because of different cache sharing models [11] but otherwise, in both cases, system and application software sees some number of apparently independent execution units. The Solaris scalability work was largely done on multiprocessor Sparc machines prior to the multicore era.

In particular, the parallel supercomputers of the 1980s and 1990s exhibited all the same scaling limitations and ensuing software issues that we expect to see in multicore systems. Rather than waste time repeating that history, we should look at where that work led. Parallel supercomputers began with shared memory multiprocessor designs much like today's four- and six-core boxes. They have developed into massive shared-nothing clusters that communicate by message passing, like BlueGene [1]. The experience in that sector says that conventional thread programming using locks and shared memory does not scale to hundreds of cores. Moving to the cloud, we also find that Map/Reduce is based on a shared-nothing model. Drawing too detailed an analogy between domains with different characteristics is dangerous; however, it is reasonable to assume that when ordinary server machines come with cores in large numbers that they too will be shared-nothing.

There is, consequently, no point pursuing the conventional thread programming model and no point trying to rework languages and tools to make that model more friendly to non-experts. Instead, we should be looking to programming models, concurrency paradigms, and languages that natively support, or are based entirely on, messages rather than shared memory. Note that the Barrelfish group [5] reached essentially the same conclusion from a somewhat different set of premises.

Given the lead times and expense associated with developing system software, we should already be doing this. There is nonetheless no time like the present to get started. The alternative is to give up and run a thousand VMs in one box; that seems undesirable.

The obvious candidate for a new paradigm is the lightweight message channels of Hoare's communicating sequential processes [13] and Milner's pi calculus [16]. These lightweight channels are different from most commonly seen uses of messages in that they are fine grained and low level: in this model sending a message is an action comparable in scope to making a procedure call.

(This is lighter weight than the messages typically used on supercomputers; however, communicating between cores on the same die is also lighter weight than communicating between cluster nodes in a rack.)

This paradigm has not (yet) seen wide adoption, but has been used successfully in demanding contexts: it is the model behind Erlang [3], which has been used to build real-world systems with impressive scalability and reliability properties. The Ericsson AXD301 ATM switch, whose software is written entirely in Erlang, boasts nine nines of reliability [2], meaning it is down for no more than 32 ms per year. A weaker version of the model powered occam [14], the language used on the Inmos Transputer. Recently Google's Go language [12] has made the model available to a broader audience.

The lightweight channels model is highly suitable for upcoming multicore hardware, because it avoids the need for shared memory. Fortunately it is also suitable as a basis for a new generation of operating systems. Because traditional procedure or function calls are a special case of messages, the conventional Unix system call API can easily extend to messages. Existing single-threaded code that is not performance critical can run unchanged. Old shared-memory threaded code can be run on a small to intermediate number of cores, as long as hardware supports that. New code can be written to take full advantage of hundreds of cores at once.

The lightweight channels model can support either a microkernel or macrokernel design. However, additional architectures are possible that share the advantages of both traditional designs, as both are based on synchronous procedure calls while messages can be asynchronous. In particular a thread or core need not switch protection domains to send system call messages. One such architecture is discussed herein; others are possible.

In the next section we discuss related work, then in Section 3, we discuss the lightweight message model in more detail. Section 4 presents our proposed OS architecture. Section 5 discusses some challenges that arise when implementing this design, and Section 6 summarizes and concludes.

2 Related Work

Messages as a concept are found in many places, including most distributed OSes and such highly obscure software constructs as TCP/IP. Attempting to list even the highlights would be futile. Lightweight messages and lightweight message channels as discussed in this paper are less common. Most messages seen in systems are middleweight, comparable to a system call or network packet: most microkernel messages and distributed OS messages fall into this category. Mach [20] is the canonical example. (Note however that in some systems, such as L4 [15], messages are synchronous; the caller is suspended until a response arrives. These are really procedure calls, not messages in the general sense.) The message systems commonly used in cluster supercomputers are roughly this weight also.

Lightweight messages, comparable to a local procedure call, mostly appear only when provided by a programming language. The Amiga OS [8] was a notable exception; it was structured entirely upon messages in a C-based shared memory environment. Barnes et al. have a message-oriented operating system under development [4]. It is written in an extended variant of occam; though intended for embedded systems the broad design outline is (unsurprisingly) similar to that proposed here.

As noted earlier, Erlang [3] provides lightweight messages. Many real-world systems have been written in Erlang, but it remains outside the mainstream of systems development. Rob Pike developed Newsqueak [19] more than twenty years ago, specifically to bring the structural advantages of the message model to GUI programming; this technology has been carried forward to Go [12].

The work of Hoare [13] and Milner [16] forms the theoretical foundation of the model, with other important work being done by Reppy [21].

Conventional macrokernel OSes (BSD, Linux, Solaris, UNIX, CP/M, whatever) are not message-based at all and do not enter into this discussion, although FlexSC [22] proposes message-based system calls for Linux. The relationship of the proposed architecture to the conventional microkernel and macrokernel architectures is discussed at the end of Section 4.

3 A Model of Messages and Channels

The lightweight channel and message model is a different programming paradigm from traditional procedure-based or even object-based code. This section provides a brief outline of the model, explaining the new constructs and operations and how they relate to more familiar code.

The lightweight channel and message model adds a new type (the channel) and two new statements (send and receive). Channels are the objects through which messages are sent; they identify endpoints for communication. Messages can typically be any language value. The statement `c <- i;` sends the value `i` to the channel `c`. The statement `i <- c;` reads a value out of the channel `c` and stores it in the variable `i`. Depending on the implementation, send can be blocking or non-blocking. A blocking send waits until a receiver is available; a non-blocking send queues values for later. Blocking send is easier to implement in a low-level environment (no buffering) and is more powerful [21]; however, non-blocking send tends to be easier to use and, being less synchronous, is probably faster.

A function call `r = f(a, b);` is equivalent, given a listener thread on channel `c` that evaluates `f`, to writing `c <- (a, b, c1); r <- c1;`, where `c1` is a fresh channel used to send the return value back. This is the basis of all network RPC systems, of course, but it remains true at this level as well, and illustrates how the lightweight channels model is a generalization of conventional procedure calls. (Of course, nothing prevents supporting ordinary procedure calls and implementing them in the usual way.) Note that channels can be sent through channels. This makes it possible to, for example, plumb a connection by passing around a channel to be used to carry data, and then afterwards move the data directly to its destination by a single send operation.

The model also adds a new control structure, *choice*. For example, the syntax

```
choose {
 option r1 <- c1: action1(r1); break;
 option r2 <- c2: action2(r2); break;
 option r3 <- c3: action3(r3); break;
};
```

executes exactly one of the option lines, choosing to receive from whichever channel becomes ready first and then executing the associated `action` code. In environments with blocking send, choice typically allows options that send as well as options that receive. Choice provides functionality akin to the Unix `select` system call (but on program objects rather than open files) and is one of the things that makes the model powerful.

In this model threads are also lightweight, so typically starting one is easy. The syntax `start { foo(); }` launches a new thread to call the procedure `foo`.

Channels provide synchronization as well as communication. Strict implementations, such as Erlang, use no shared memory; threads send messages through channels by copying. This buys scalability at the cost of some memory bandwidth overhead.

3.1 Peer vs. Hierarchical Structure

One of the benefits of the lightweight channels model is that it allows moving away from procedure calls as the dominant mechanism for structuring programs. Procedure calls are asymmetric: the caller and callee are fundamentally different. This produces a hierarchical structure where the caller runs on top of the callee. Bidirectional calling among peer subsystems tends to become messy in languages that require a caller/callee relationship. Typically one subsystem must artificially sit atop the other, with reverse-direction calls set up as callbacks.

In the lightweight channels model, the relationship between sender and receiver is symmetric, because messages go from one running thread to another with no control transfer. Peer subsystems can be structured to send messages back and forth on a peer basis, instead of requiring a false hierarchical relationship. This is particularly desirable for GUI programming, where the application and display send messages back and forth. As mentioned earlier, Newsqueak [19] offered this model.

There are a number of other cases in operating system design where events or data flow naturally in the opposite order to the normal subsystem layering. For example, thermal, power, and hot-plug events necessarily originate in the kernel and flow upward to user space. Handling these in a traditional nested kernel design is always somewhat problematic. Similarly, in current Unix systems, asynchronous I/O completion notices are handled via signals. If the process or thread receiving a signal is working in the kernel, it must abandon and unwind everything that was in progress in the kernel to deliver the signal. Then, typically, the process must restart the system call and redo all the work it just unwound. This is unnecessarily wasteful; most systems today have some special purpose channel-like system to allow receiving these and other notices more effectively. In an environment designed around message channels this is not needed.

Note that hierarchical nesting of abstractions is still a perfectly fine design principle, message channels or not, when it makes sense [9]. However, it is neither necessary nor desirable to shoehorn all problems into such a design.

4 Architecture

The foremost design choice for adopting the lightweight channels model is the implementation language. The system should be structured around the implementation language the same way Unix is structured around C. Because this system is needed in the mainstream, and therefore must run old code easily, the language should be a concurrent dialect of C and not Erlang. This language must retain as many of the advantages of Erlang as possible (e.g. small thread stacks) without adopting Erlang's less desirable features (e.g. dynamic typing, sharing forbidden) and also not compromising the value of C as a systems implementation language. However, detailed language design is beyond the scope of this paper.

The key point from an operating system design perspective is that it is no longer necessary to transition to kernel mode to make system calls, and no longer necessary to use the cumbersome signal delivery mechanism to send messages asynchronously from the kernel to an application. Instead, we assume that some number of kernel components are running on some cores and some number of application components are running on other cores. Making a system call involves sending a message from an application thread running on an application core to a kernel thread running on a kernel core. This can be

done without any mode transitions even without any particular hardware support [22]. We can, however, reasonably suppose that future hardware will have native support for sending and receiving messages.

In an aggressive design one might well run applications directly on a bare core with no system services at all underneath. If an application wants e.g. virtual memory services or handlers for hardware exceptions it can provide them itself or link with system-provided code in libOS fashion. This is perfectly reasonable in a shared-nothing environment where applications cannot scribble on each other. A more conservative design, or one that works given shared physical memory, would still have to provide virtual memory and similar low-level services underneath the application, but the size of the "kernel" layer underneath the application would be minimal. In either case it does not matter if the kernel components that receive system call messages are running the processor in supervisor or user mode. Also note that in either case legacy code can be linked against a compatibility library and used unchanged.

The kernel components that receive system call messages are only the outer interface, of course. Changing the system call interface around in a similar manner, just to run the kernel and applications on different cores, has been proposed before but will not itself make the kernel scalable. Indeed, the whole kernel can and should be structured to use lightweight channels in place of procedure calls. Any abstraction found in a standard kernel design can and probably should be converted to an autonomous thread sending and receiving messages. These can then be placed across a large number of cores as desired. For example, the file system could be structured so that every vnode is its own thread, which communicates with other threads that administer cylinder groups and free-maps and so forth. The premise of the model is that messages (and threads) are made lightweight enough for this kind of structure to be possible with only small overhead that is made up by scaling.

Dispatch to different objects via a common interface, which is conventionally done with tables of function pointers, is done in this environment by sending to a channel (or family of related channels) using a common message protocol. The code receiving the messages can be different for different channel instances.

It is also almost certainly desirable to give each device driver its own, single, thread. Drivers would receive and queue requests from elsewhere in the kernel; the code to process the requests can then be written as simple active procedural code, with no need for further synchronization except to wait for interrupts. This eliminates a fertile source of driver bugs. Note that as most hardware has limited if any ability to do more than one thing at once, there is little drawback to making any individual device driver single-threaded. This is not a novel idea but the lightweight channels model makes it easy to implement.

It is also worth noting that while verification of concurrent programs is difficult and verification of highly concurrent programs is even more difficult, the use of messages, channels, and defined protocols offers some potential for static verification using techniques developed for networking software.

This design is neither a macrokernel nor a microkernel in the conventional sense, but exhibits properties of both. For example, it is like a microkernel in that the kernel is divided into components that communicate among themselves but do not share memory or explicit state. It is like a macrokernel in that the kernel is one body of code and anything in the kernel can "call" (send messages to) anything else if the language-level interfaces (channels rather than external functions) are suitably exposed. It is unlike a microkernel because the central function of a microkernel, conveying IPCs from one process to another, is relegated to hardware. And it is unlike a macrokernel because any individual thread operating any individual internal object can be set up to run with the processor in user mode rather than supervisor mode. The VM services of the conservative design could be considered to be a vestigial microkernel; however, in the aggressive design this code disappears completely.

This point may be somewhat controversial; it is worth noting but probably not worth belaboring. The design proposed is structurally more similar to a client/server network application or to a cluster environment than to either traditional kernel design.

5 Implementation Issues

There are doubtless a large number of challenges and difficulties that will appear while implementing this design. Most probably can only be discovered in the course of development; this is characteristic of building research systems. However, some can be foreseen ahead of time.

Implementing choice effectively is always somewhat difficult; implementing it effectively for hardware-based channels can be expected to be a challenge.

Waiting for channels to become ready will likely be a source of hassles. This has been seen already with Infiniband and similar technologies, where waiting for operations to complete causes significant complications.

It is not entirely clear how virtual memory should operate in this environment. In the aggressive design described in the previous section virtual memory for protection is abandoned, and support for memory-mapped files and other uses of the MMU can be handled by a libOS. (And it is not entirely clear whether memory-mapped files are desirable or even workable in a shared-nothing environment.) In the conservative design, the

virtual memory system is retained, but its internal design will be necessarily much different from today's centralized model, and it is not yet clear exactly how.

The search for parallelism to enable scalability can yield too much. With lightweight and fine-grained channels and threads it is easy to write code that uses vast numbers of threads. For example, one might build a virtual memory system with a thread for every page of physical memory in the system; that would produce too many threads no matter how many cores are available. The risk is that there may be no clean intermediate design points between too many and too few threads and design compromises may be needed.

Partial failure, in which some components of a system fail but others keep operating, leaving the system inoperable and not easily recovered, becomes a problem whenever there are multiple nontrivial autonomous entities. Making a kernel built with lightweight channels fully fail-stop is likely to be a challenge. On the other hand, given some of the experience with Erlang [2] it may be feasible to aim for not failing as an alternative.

Scheduling in general, and the specific problem of deciding which threads to place on which cores, and which groups of threads to place together on the same core, is likely to present a new range of difficulties as the number of threads and cores goes up.

And, of course, scaling in this world is only easier, not free or automatic; it will take work, and will be difficult early in the development cycle before the hardware is available.

6 Conclusions

This paper argues for the operating system level adoption of a lightweight messages and channels programming model. It outlines a novel OS architecture based on this model and identifies some areas in which the implementation will likely be challenging.

Adopting the model and this or a similar architecture will allow building operating systems that run effectively on upcoming processors offering hundreds of cores or more on a single chip. The likely alternative is the thoroughly unsatisfying and inefficient approach of turning such a chip into a cluster of hundreds of apparently separate virtual machines, with a few cores each, running unmodified existing OSes. If the world becomes forced to adopt such an approach, we, the systems research community, will have failed.

We would prefer not to have to say that in fifteen years.

References

[1] ADIGA, N., ET AL. An overview of the BlueGene/L supercomputer. In Proc. of Supercomputing 2002 (November 2002).

[2] ARMSTRONG, J. What's all this fuss about Erlang? http://pragprog.com/articles/erlang.

[3] ARMSTRONG, J., DACKER, B., VIRDING, S., AND WILLIAMS, M. Implementing a functional language for highly parallel real time applications. In Proc. 8th Intl. Conf. on Software Engineering for Telecommunication Systems and Services (1992), pp. 157–163.

[4] BARNES, F. R., ET AL. A scalable, compositional operating-system for commodity platforms. http://rmox.net/.

[5] BAUMANN, A., ET AL. The Multikernel: A new OS architecture for scalable multicore systems. In Proc. of the 22nd SOSP (October 2009).

[6] BECHTOLSHEIM, A. Memory technologies for data intensive computing. In Proc. of HPTS 2009 (October 2009).

[7] CLARK, J. Intel: Why a 1,000-core chip is feasible. http://www.zdnet.co.uk/news/emerging-tech/2010/12/25/intel-why-a-1000-core-chip-is-feasible-40090968/.

[8] COMMODORE-AMIGA INC. Amiga ROM Kernel Reference Manual: Libraries, 3rd ed. Addison-Wesley, 1992.

[9] DIJKSTRA, E. W. The structure of the "THE"-multiprogramming system. Commun. ACM 11 (May 1968), 341–346.

[10] FARNHAM, K. Interesting multicore crisis graph and analysis. http://software.intel.com/en-us/blogs/2008/01/17/interesting-multicore-crisis-graph-and-analysis/.

[11] FEDOROVA, A. Operating System Scheduling for Chip Multithreaded Processors. PhD thesis, Computer Science, Harvard University, 2006.

[12] GOOGLE. The Go programming language. http://golang.org/.

[13] HOARE, C. A. R. Communicating sequential processes. Commun. ACM 21 (August 1978), 666–677.

[14] INMOS LIMITED. occam2 Reference Manual. Prentice Hall, 1988. ISBN: 0-13-629312-3.

[15] LIEDTKE, J. On micro-kernel construction. In Proc. of the 15th SOSP (1995), pp. 237–250.

[16] MILNER, R. The polyadic pi-calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, 1991.

[17] PATTERSON, D. Hardware trends. In Proc. of HPTS 2007 (October 2009).

[18] PATTERSON, D. The trouble with multicore. IEEE Spectrum (July 2010). http://spectrum.ieee.org/computing/software/the-trouble-with-multicore.

[19] PIKE, R. Newsqueak: A language for communicating with mice. Tech. Rep. 143, AT&T Bell Laboratories, January 1989.

[20] RASHID, R. F. From RIG to Accent to Mach: the evolution of a network operating system. In Proceedings of 1986 ACM Fall joint computer conference (Los Alamitos, CA, USA, 1986), ACM '86, IEEE Computer Society Press, pp. 1128–1137.

[21] REPPY, J. Concurrent ML: Design, application and semantics. In Functional Programming, Concurrency, Simulation and Automated Reasoning, P. Lauer, Ed., vol. 693 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1993, pp. 165–198.

[22] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with exception-less system calls. In Proc. OSDI 2010 (2010).

[23] WOLFE, A. Intel blog warns of multicore crisis. http://www.informationweek.com/blog/main/archives/2008/01/intel_blog_warn.html.