A Proposal for a Public Domain Systems Library

Brent Welch
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
welch@parc.xerox.com

Position Statement for WWOS III

Introduction

This position paper reflects my desire to capitalize on the excellent work done by others in the area of operating systems research. The best way to do this is by sharing source code so that others can directly evaluate work that is reported in the literature. This approach would address the problem that technical papers about software rarely explain all the details required for others to reimplement the work. Overall descriptions are given, performance results are presented, and an occasional interface is defined. This is all done in an ad-hoc, non-machine-readable way, so the work slips by and others reinvent out of necessity. In reality, software is defined by the code itself, and this is rarely distributed or studied in its own right. The wide success of UNIX, for example, was largely due to the distribution of the source code so that others could modify and enhance the original work. The drawback of the UNIX sources is that they are encumbered by license agreements that prevent access by small projects, not to mention that they are quite old by now and have evolved in clean and not-so-clean ways over the last 20 years.

Consider, for example, the recent comparison of Sprite and Amoeba [Douglis91]. The comparison is between two very different systems. Amoeba is a microkernel-based system oriented towards distributed applications. Sprite is based on a monolithic kernel and is oriented towards networks of UNIX workstations. Performance for many basic operations in Amoeba is better, but this can be traced to simplifications with respect to UNIX semantics. There is no virtual memory and a different process creation model, for example. In contrast, the Sprite file system uses aggressive caching techniques that make some big jobs, such as compilations, run very fast. However, [Douglis91] does not make a comparison at this level because UNIX emulation in Amoeba is “too slow”. Are these pragmatic differences, or fundamental differences? How hard would it be to take the best features of the two systems and make a new system that is better? Each system is designed in a modular way, so in theory it should be possible. In practice it would be difficult because, in spite of modular design, there are complex dependencies among modules that make it difficult to extract modules. Again, are these fundamental dependencies, or pragmatic? Could the systems be structured in a way that eases sharing of key components?

The ideal situation for me as a researcher in operating systems would be to have, sort of as a public trust, a library of operating system modules that could be composed in various ways to perform experiments and build new systems, or at least studied to see exactly how systems are being implemented. There are a number of difficulties in actually achieving this, and some of these are discussed below. However, there are also lots of interesting problems to solve along the way to achieving it that are quite interesting in themselves. I think most people would agree that a public domain operating systems library would be quite useful.

“A fantasy,” you say. “Impossible to manage, coordinate, comprehend and utilize.” Perhaps. First I will outline an approach to building the library, and then discuss some of the related problems.

The Ideal Library

The primary function of the library would be as a repository for interfaces and implementations for core operating system abstractions. Putting together a working system is not a goal of the library itself. This is important to note because it allows for conflicting interfaces and competing implementations, both of which are inevitable and necessary if the library is to improve over time. My ideal is to have a library of building blocks that are simple so they can be easily understood and maintained, that have well defined interfaces so they can be easily integrated into bigger systems, and are “good” so that useful, high-performance, robust systems can be easily built from them.

I am inspired by Massilin’s Synthesis kernel work in which each abstraction has been carefully distilled to be completely orthogonal of others [Massilin89][Pu91]. Basically, all interactions with other modules are via procedure variables that are defined at “system composition” time. Most of the Synthesis examples I have seen are for simple abstractions like queues that have a handful of procedures in their interface. Some procedures are exported, such as queue() and dequeue(), while others are imported such as qfull(), qempty(), and qstart(). The instantiation of the imported procedures defines the way the queue synchronizes and schedules itself with respect to the rest of the system, while the exported procedures define the queue abstraction itself.

There are two categories of modules, glue code and building blocks. The ideal building block is something like a Synthesis module that just has a set of procedure variable slots that need to be filled in. Glue code composes building blocks by filling in the slots of various building blocks and by making calls on their exported interfaces. Glue code, in turn, can export interfaces and import them, thus becoming a higher-level building block. There is a direct analogy with board-level hardware design. The building blocks are standard chips, the glue code is the connecting wires on the PC board, and the external interfaces are standard bus and cable connectors.

As another software example, consider a device driver. The building blocks are chip-specific modules that control a particular piece of hardware such as a DMA controller, SCSI controller, or ethernet interface, and chip-independent modules for things like buffers and queues. The main part of the driver is glue code that manipulates the hardware indirectly through the chip-specific modules to achieve some desired behavior. At the top-level, the driver exports one or more interfaces to the rest of the system to handle user I/O, bootstrap and autoconfiguration. The driver also imports interfaces that manage interrupts, scheduling, and synchronization. This example is derived from Allesandro Forin’s work on the Mach 3 device drivers[Forin91].

On the surface, this approach seems just like ordinary software engineering. What’s different? First, the goal of providing a useful toolkit of systems modules is different than the goal of most operating systems projects, which is to provide a working operating system. Of course, building a useful system is an indirect goal. The primary goal, however, is to evolve a set of interfaces and implementations that cleanly define the abstractions of threads, address spaces, protocols, scheduling, file systems, etc., that are by now well known, as well as the host of underrated internal interfaces that are really necessary in real systems. Achievement of clean interfaces and good implementations can take considerable time. Good designs often result from experiences with prototypes that highlight the problem areas. Eventually, the proper interfaces and good implementations arise.The goal of a systems library is to capture these good interfaces and implementations in a machine-usable form, in source code.We need to canonize these important abstractions, just as the mathematicians have canonized the differential equations that describe the fundamentals of physics.

Problems

There are two classes of problems facing a project like the operating systems library, cultural and technical. First, the cultural problem. We've come so far without one, why should we spend any energy on such a large project. Won't the effort outweigh the gain? Why would anyone be willing to contribute their carefully crafted interfaces and implementations? What is the payoff to the individual contributors and the computer science community at large?

The payoff to the community is a software base useful for building new systems and for comparing competing ideas. This payoff is difficult to measure, at best, but I think most would agree that such a software base would be worthwhile, if it already existed. In the best case, the payoff would be an operating system, or family of operating systems, that incorporate the best work in the field. The payoff to individuals is also somewhat intangible, unless there is some way to acknowledge "good" contributions just as good technical papers are acknowledged. Corporations have to be content with an indirect payoff. They may wish to contribute software that makes it easy to use their hardware, for example. Many vendors already give out source code to their drivers, if requested. Unfortunately, many system vendors consider their operating system proprietary and guard it to make it difficult to "clone" their systems. It might be possible to convince them to give out binary-only modules for crucial low-level functions.

There are a number of technical problems: Version management, system integration, language conflicts, runtime support conflicts, and overall comprehension.

Version management and systems integration are closely related. System integration is a problem because there could be many competing modules for a particular function, each with its own set of dependencies on other supporting modules. In order to make good decisions, the system designer must be aware of such interdependencies. Then, once any decision has been made as to the system's composition, changes in modules by other researchers must be tracked, incorporated, rejected, or further improved. Solutions to this class of problems can be arbitrarily sophisticated. However, I think that a registry of interfaces, implementations, and their individual dependencies forms the basis for a variety of tools that can be built to help manage the library. First on my list would be a browser that can help a designer visualize the wide range of interfaces, implementations, and their dependencies. There has to be a better approach than looking through directories and reading interface definitions.

Language and runtime support conflicts are also related. It may or may not be feasible to intercall between modules in different languages. Even if the calling sequence can be worked out, runtime conflicts arise. The main areas for runtime conflict are memory management (e.g., garbage collection), synchronization model (e.g., single-threaded or multi-threaded), and exception handling. My approach to these problems would be to annotate components of the library with their assumptions about these runtime features. This merely shoves the problem off to the system integrator, but at least the assumptions are documented. In fact, I can imagine closely related families of modules that vary only on these assumptions. This could help us measure the impact, both positive and negative, of these features on a system.

Related Work

Currently the main depository of source code are the anonymous ftp directories scattered throughout the internet, and systems like Archie [Emtage92] make it possible to search this resource. There is also OSF, whose main product is “software technology”, or source code, that vendors incorporate into their own products with some value added features. I laud the source code sharing spirit of these systems. However, while this works well for stand-alone applications found in the ftp directories, it has worked less well for enabling software. The problem with OSF DCE, for example, is that each subsystem is large and complex in itself, and the relationships and interdependencies among the components is not well defined. These subsystems have been designed in the context of various systems, and it is quite likely that subtle dependencies on their environment have affected their implementation. It might be very difficult to take only a subset of the DCE functionality. In fact, it may also prove to be difficult to make all the pieces work well together.

There are a number of research operating systems that could contribute to an operating systems library. Some, such as the V system [Cheriton84], have been distributed for some time. Others, like Sprite [Ousterhout88], could be distributed except for the perceived effort of supporting the system. The X-kernel [Peterson89] is available via FTP, but its protocol stacks are best utilized in the context of an existing operating system, as a replacement for the UNIX socket implementation, for example. Mach [Accetta86] is widely used because it was aggressively marketed to the UNIX and research communities, and to companies that wanted out of the operating systems business. However, the original code base for Mach was 4.1 BSD UNIX, and even the microkernel bears many resemblances to the original UNIX code. This brings up licensing issues, and it also makes some defacto design decisions that ought to be reconsidered.

The general problem with these research systems is that they have focused on a particular problem domain and have put less effort into other areas. This is an obviously good research strategy, but it now leaves us in the state where the best overall system doesn’t exist. Instead, the X-kernel has the best protocol stacks, Sprite has the best file system, Mach has the best VM system, and so on. Furthermore, I am sure there are other projects that could challenge these claims, if only there was a level playing field upon with to make comparisons.

Conclusions

I am proposing a library of operating systems modules that is freely available and is a object of research and investigation in its own right. I am motivated by the observation that, in spite of the many advances in the state of the art in operating systems research, the path of transferring this technology into existing systems is slow and difficult. Operating systems are the foundation of our computer systems, and they present a paradox. On one hand, it is important to “get them right” so the system built atop them works as best it can. On the other hand, it is risky to change them for fear that the whole system will break or that massive changes to the rest of the system will be needed. However, in today’s environment of inexpensive workstations, it is much easier to experiment with low-level software than in the days of timesharing systems. Fear of interface changes stems from lack of understanding of the whole system. This is a self-perpetuating problem, however, because each minor hack to a system makes it less maintainable. Instead, it is time to extract out the salient features of various operating system abstractions and publish “reference” implementations. A public code base will foster increased understanding and faster transfer of research results to industry.

References

Accetta86. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, “Mach: A New Kernel Foundation for UNIX Development.” *Proc. of the Summer 1986 USENIX Conference*, July 1986.

Cheriton84 D. Cheriton, “The V Kernel: A software base for distributed systems”, *IEEE Software*, 1, 2 (Apr. 1984), 19-42

Douglis91 F. Douglis, J. Ousterhout, M. F. Kaasheok, A. Tanenbaum, “A Comparison of Two Distributed Systems: Amoeba and Sprite”, *Computing Systems* 4, 4, (Fall 1991), 353-384

Emtage92 A. Emtage and P. Deutsch, “archie - An Electronic Directory Service for the Internet”, *Proc. of the Winter 1991 USENIX Conference*, Jan. 1992, 93-110.

Forin91 A. Forin, D. Golub, B. Bershad, “An I/O System for Mach 3.0”, *Proc. of the Mach Workshop*, Nov 1991, 163-176

Massilin89 H. Massalin, C. Pu, “Threads and Input/Output in the Synthesis Kernel”, *Proc. of the 12th Symp. on Operating System Prin.*, Operating Systems Review, Dec. 1989, 191-201

Ousterhout88 J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, B. Welch, “The Sprite Network Operating System”, *IEEE Computer* 21, 2, (Feb 1988), 23-36

Peterson89 L. Peterson, N. Hutchinson, S. O’Malley, M. Abbott, “RPC in the x-Kernel: Evaluating New Design Techniques”, *Proc. of the 12th Symp. on Operating System Prin.*, Operating Systems Review, Dec. 1989, 91-101

Pu91 C. Pu and H. Massilin, “Quaject Composition in the Synthesis Kernel”, *Proc. of the 2nd International Workshop on Object Orientation in Operating Systems*, Oct. 1991, 29-35