RC24436 (W0711-179) November 26, 2007
Computer Science

IBM Research Report

K42: Lessons for the OS Community

Robert Wisniewski, Dilma Da Silva, Marc Auslander, Orran Krieger*, Michal Ostrowski*, Bryan Rosenburg
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Currently at VMWare

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

K42: Lessons for the OS Community

Robert W. Wisniewski [‡] Dilma da Silva [‡] Marc Auslander [‡§]
Orran Krieger [‡¶] Michal Ostrowski [‡¶] Bryan Rosenburg [‡]

Foreword

In the late 1990s, IBM Research started a research initiative called the Adventurous Systems and Software Research program. The goal of this program was to allocate resources to support longer-term research, removing the shorter-term horizon implied by the standard funding model, which involves partnering with a development organization. We started the K42 operating system project under this auspice. We had the opportunity for the first three years to pursue an ambitious research agenda because we were not bound by product impact requirements often seen in industrial research settings, nor immediate publication pressures often seen in academic settings.

Other factors that impacted the environment in which K42 was conceived are technical. In the mid 1990s vendors were scaling machines by either shared-memory multiprocessors (SMMPs) or non-shared-memory clusters. The K42 group focused on researching how to design systems software for machines projected by the SMMP scaling model. While this model did not become dominant, large machines still exist. Further, the advent of multicore chips makes this research relevant, and there will likely be a resurgence of parallel operating systems research.

The initial focus of the K42 group was on scalability, but as the project progressed other interesting directions were pursued, motivated primarily by the observation that fewer projects in complete operating system design were being undertaken in the 2000s than in the previous two decades. We spend some time in this paper exploring why this trend may be happening, but an implicit statement by the operating systems community one could take away may be that either Linux or Windows is the correct solution. We felt that while the inertia of the mass of software designed for these two models is great, innovation in the system software stack is also important. This observation motivated the on-the-fly customizability aspects of K42 as well as subsequent explorations of alternative ways of achieving legacy support while allowing innovation.

Abstract

We started the K42 project more than ten years ago with the ambitious goal of developing an operating system for next-generation hardware that would be widely valued and thus widely used. Based on the premise that current operating systems were not designed to be scalable, customizable, or maintainable, we set forth to rectify that by applying proven techniques from other disciplines to operating systems and by developing additional innovative mechanisms.

Now, ten year later, K42 is used by ten or so universities and national labs for research purposes, not ten million information technology departments desiring better everyday computing platforms. As a presentation to the primary operating systems community we provide an examination from two different perspectives as to what went right and what went wrong. First, we concentrate on what technology worked well and why, and what technology failed or caused undue difficulties, and why. Second, based on that experience, we provide our thoughts on the state and direction of the OS community at large.

To be clear, this paper is neither a results paper nor an overview paper; we refer to other papers for background material. Rather, it is an exploration by researchers with experience with at least six different previous operating systems of the merit of technologies investigated in K42, and an extrapolation of the implications of that experience to the wider operating system community.

1. INTRODUCTION

Research in complete operating systems, which involves developing new fully-functional OSes capable of running significant applications, is hard and resource consuming. First, operating systems require a tremendous amount of infrastructure to present a complete solution to the user, while the research community and the marketplace are not kind to those who take time to get that infrastructure right. Second, there are rarely clearly right or clearly wrong solutions. One benchmark will favor mechanism A, while another favors mechanism B. Third, there are many disciplines, e.g., networking, memory management, file system, etc., involved in putting together an operating system. Fourth, to run many different applications they need to be ported to the new OS interfaces or the OS must support a standard set of interfaces; see the first observation. This list is far from complete.

A fair question then, is why start a new complete-operating-system initiative. Our hypothesis was that the operating systems available did not meet the community's needs with regards to scalability, customizability, and maintainability, or facilitate significant innovation. Other researchers and developers in the community have also written about the

[‡]IBM T. J. Watson Research Center. This work was partially supported by DARPA contract NBCH30390004 and DOE's FAST OS program
[§]IBM Fellow Emeritus
[¶]currently at VMWare

serious shortcomings of existing solutions [22, 21, 16, 20, 17].

There has been a marked decline in the number of complete-operating-system initiatives. The environment for such initiatives is no longer conducive for several reasons. Pressures from academic publication volume or industrial research deliverables have increased. More importantly, the number of legacy interfaces and applications that must be supported for a system to be relevant has increased with the growing user base. In the ten years before we began K42, a large number of complete OS projects were undertaken, including Chorus, MACH, Sprite, Synthesis, Peace, Amoeba, Clouds, Spring, Apertos, Choices, Opal, VINO, Plan9, Exokernel, SPIN, Rialto, Paramecium, Nemesis, Scout, Tornado, Eros. In the last ten years, fewer complete OS projects have been started. Only K42 [18] and Flux [15], and more recently Singularity [17] and Asbestos [13] are examples. We do not imply that OS research does not occur, rather that research into whole or complete operating systems has declined.

Despite the challenges, we were motivated to undertake K42 because we believed there was a fundamental need based on future software requirements to examine the core design of kernels. Based on the premise that existing operating systems were not designed to be scalable, customizable, and maintainable, we set forth to apply proven techniques from other disciplines and develop additional innovative mechanisms to address these needs.

Today, there are a growing number of research organizations, including universities and national labs, that are using K42 for research. Because, as noted above, such undertakings are rare in the operating systems community, we present to the primary research operating systems community an open evaluation from two different perspectives as to what went right and what went wrong.

A fundamental decision was to apply object-oriented design using C++ throughout the system. Ten years ago object orientation in the operating system was viewed as highly controversial, at least as witnessed by reviewer comments and conversations with other OS researchers. More recently, operating systems, including commercial ones, are becoming more modular, with more tightly defined interfaces between components. As modularity was both a pervasive design decision and viewed with skepticism, this decision receives considerable attention throughout the paper. In short, the researchers involved with K42 unanimously viewed this decision as a success.

Another important design decision was avoiding the use of centralized code paths and data structures. We accomplished this by having separate sets of object instances manage each operating system resource. This design was motivated by the desire to scale, but also positively impacts customizability and maintainability. It too is viewed as a success.

Customization was another technology explored extensively in K42. The goal was to combine on-the-fly customization of K42 with Linux API support to allow innovation by researchers and developers while supporting legacy interfaces and running legacy applications. Code, we reasoned, could use the standard interfaces and directly call native K42 services when performance was needed or to take advantage of new innovations. The customization was successful in its own right, but because of Linux API support issues did not lead to allowing innovation while providing legacy support.

Of all our decisions, the decision to make the system complete and not just a research prototype was the one with the biggest impact. Although at the outset, we all agreed this was a good thing, with positive and important consequences, it proved to be a mammoth time sink and is viewed with very mixed opinions among the K42 team. The reasons we chose this path and ensuing complications have significant implications for the operating systems research community and the systems community as a whole. In particular, we describe the impact of the perceived or real need to support legacy software and interfaces. Unless the OS community believes that the two popular systems, Linux and Windows, are the right answer for the future, serious soul searching into fundamental technology that allows modification of underlying structure and techniques to break away from supporting legacy interfaces, needs to occur. In later sections we describe how some of our techniques, combined with virtualization, has potential to address these challenges.

Other principles we discuss in the paper include leveraging the performance advantages of 64-bit processors, moving traditional system functionality out of the kernel, the use of an integrated performance monitoring infrastructure, and our open-source software model.

We have demonstrated the efficacy of the techniques we employed that allow K42 to scale, and of the mechanisms we implemented for customizability. The jury is still out as to whether the techniques we applied for maintainability will prove effective.

This paper is neither a results paper [25, 5, 6, 8, 26, 27, 4, 24, 3] paper nor an overview paper; these are available on our web page [18]. Rather, it is an exploration by researchers with experience with several operating systems, including AIX, Hurricane, IRIX, Linux, RP3 Mach, Solaris, and Tornado, of the merit of technologies investigated in K42 and an extrapolation of the implications of that experience to the wider operating system community. The best single reference providing an overview of K42 is a 2006 Eurosys paper [19].

The rest of the paper is structured as follows. Section 2 contains our evaluation of, and experience regarding, the decisions we made. Section 3 describes why we chose to implement a complete OS, why this was a bad idea, and the implications for the OS community at large. Section 4 concludes.

2. EVALUATION AND EXPERIENCE

This section of the paper describes our experience and lessons learned from K42. For each area we indicate what we did right and what we did wrong. We generalize these experiences and describe their implications on other complete operating systems projects.

2.1 An OO OS

Object-Oriented (OO) programming — everyone says what a wonderful thing. Well maybe not everyone — OS researchers, at least those that reviewed early papers and those we met at conferences, were skeptical. For those espousing OO benefits, high on the list is code re-usability; few mention scalability. That, however, was our primary motivation. Succinctly stated, implement OS resource management by grouping the needed code and data together in local objects, then place the object on the processor requesting the resource. Most OS resource requests, especially in a multiprocessor, are independent. Thus, the model should scale well. Continuing with the OO motivation list one will find flexibility and encapsulation. We used those in K42 to provide customizability and maintainability.

Throughout the paper when we refer to the OO design or model we mean K42's use of an object-oriented paradigm, where each OS resource is implemented by an independent set of one or more object instances, and where the code, locks, and data structures for managing that resource are encapsulated in those objects.

Customizability: We used our object-oriented design to address four broad areas in K42, one of which was customizability. Each resource K42 manages is implemented by a set of one or more objects. For example, within a process each open file has a set of objects that represent it. Thus, if a file is known to be small or read-only, an object optimized for that behavior can be used thereby achieving the significant performance improvements demonstrated by previous work. A wider-scope example is the process object. In K42 we have uniprocessor and multiprocessor versions of the process object optimized to match the behavior exhibited by a given process.

In addition to customization at process creation, K42 provides the ability to hot-swap objects and dynamically upgrade the system. This significantly increases the usefulness of customization, because for example, when a file is opened, the future usage pattern is not known by the OS. We discuss dynamic customization in more detail in Section 2.2.

In addition to our work, many other groups have looked at OS customization. In practice, little of this work is being utilized in mainstream operating systems. Although customization sounds attractive, either the technology is not yet mature enough for the mainstream audience, or there is not sufficient motivation. Even in K42, which makes customization easier, there is not widespread usage. In K42, this may be because the core team are kernel developers rather than users. We, like other researchers, believe in the long term usefulness of this customizability. An OO model is mechanistically the right approach, but policy-wise the fruits have not yet been borne out.

Scalability: A second area in which OO design was useful was in scaling the OS. The goal is to localize in individual objects the data and code needed to service a request for each instance of an OS-managed resource. By so doing, we avoid using global locks and global data structures. Using local objects was successful, with the caveat that we introduced interesting challenges when trying to implement global policies. For example, because memory is tracked in separate objects, there was no single list from which to perform a working set computation or on which to run a clock algorithm. Similarly, when trying to find the thread with the next highest priority, there was no one place to look. In both instances, the solutions we produced were efficient with similar policy characteristics to the central solution, and because they were implemented in the K42 model they scaled well.

An OO design is not the only way to avoid global locks and global code paths. In fact, operating systems, over the last ten years, have been removing them. The difference is the effort it takes to remove them in a given model. As an example, when we were tuning K42 to scale, there were times we found a lock had become global due to a change in how the object was used from how it was originally designed. However, because an OO model enforced a limited scope the lock could affect, removing it was not difficult. A very well modularized OS would have allowed the same, and one does not technically need OO to accomplish that; however, one does not write GUIs in assembly, i.e., use the right tool for the job. Using an OO model for reasons we stated provides a much smoother path to scalability. This we got right; an OO model for designing a scalable OS works.

Quick Bringup and Porting: A third area in which we utilized OO design was in facilitating quick system bringup and simplifying porting. We utilized the notion that with an OO model we could design the needed interfaces then quickly implement simple versions of non-critical code. Improvements and optimizations to the code could be made later as needed. K42 can be ported quickly to a new architecture using generic code and over time more optimized versions that are architecture specific can be implemented. The model has been a effective.

A caution though needs to be raised for new functionality. In a large space such as OS infrastructure, it is rare the motivation is found to go back and write the better version of code that was done quickly for the initial implementation. Thus, while the OO model does allow quick bringup, it can lead to later performance issues. We often did not find ourselves implementing the better version until performance debugging uncovered a problem. Whether this is because our group is small for an OS development effort, or whether the problem is endemic to large infrastructure undertakings such as an OS, is not clear, but the lure and trap exist.

Maintainability: A fourth area in which we utilized OO design was for improving system maintainability. The theory is that an OO design increases the likelihood that required changes to the system are contained to local objects, and therefore would be simpler to change and test. This property is especially useful in operating systems where traditionally code manipulates other components' data, for example, the file system and memory management code. The model makes it easier to isolate bugs because objects touch only data local to an object. While the original K42 team feels this was a success, it is viewed less enthusiastically by developers not as intimately familiar with K42 due to the increased complexity.

General OO Issues: There are two issues we encountered when using an OO model to implement K42. This first is that the use of implementation inheritance increased the complexity of the system, especially to users not familiar with OO programming. Good OO tools can help ameliorate the difficulty. The second is a greater need for achieving the good breakdown between functionality and object decomposition.

With the OO model we found it relatively easy to specialize objects. Our memory and I/O hierarchies are examples of this. In addition, the OO model allowed implementation inheritance. Both specialization and especially inheritance make the system appear complex to a new K42 developer. One idea that did help in the few places we used it, is to encapsulate functionality into common classes. This still preserves the OO model but reduces the complexity. It is more difficult to program without implementation inheritance, but it was a mistake to use it as much as we did because it leads to a more-complex-than-needed system, especially visible to those that have not programmed the system since inception.

A second related issue is determining the correct breakdown of functionality between objects. The original design of K42 was targeted at remaining personality independent, i.e., not coupling the underlying mechanisms with the supported API. As we embraced the Linux API and faced the reality of a timely implementation, we gave up on personality independence, and some coding assumptions and models from Linux were foisted onto the base infrastructure of K42. This was especially noticeable in the I/O space where K42 code was less developed and where we used more Linux code. However, the most pronounced disturbance was in implementing `fork()`. In addition to the well-known performance problems, the number of objects `fork` impacted was substantial. It caused us to implement additional special objects, to coalesce objects we intended to remain independent, and caused additional complexity due to unanticipated increased inter-object interaction.

The previous two experiences are more about OO design then OS design. But because we have described the usefulness of OO design throughout this section, we felt it was important to temper it with the pitfalls we fell into when using that model.

For maintainability, the jury is still out. Because required updates are self-contained and individual objects implement particular software or hardware requirements, updating code for specific functionality is easier. However, until the system is used by more application developers, and has gone through several versions, a definitive answer can not be provided.

2.2 Dynamic Customizability

Hardware and software are becoming more complex. Our approach to addressing the increasingly disparate needs of different applications was to provide first-class customization of operating system resources. In K42, each resource is managed by a set of one or more objects that can be chosen to match the specific needs of a given application or hardware platform. There has been considerable work that has examined customizing operating systems [9, 23, 14] and a much longer list of work showing how one-off specialized implementations of a particular service would be beneficial [25]. Recent work has argued that operating systems are not flexible enough [21]. In Section 2.1 we described static customization, here we focus on dynamic customization.

A crucial aspect of providing customizability is defining the boundaries at which customization may occur. In a traditional operating system it would be difficult to, for example, modify on a region-by-region basis how in-core page faults are handled, or to have different behaviors for different files. In K42, objects provide a natural boundary at which to customize. They also allow us to relatively easily provide the more challenging aspect of dynamic customization, namely changing data structures, versus modifying only code. The tight integration of dynamic customization with K42's OO model proved to be a large win.

As we noted in previous examples, an implementation with local-only objects as encouraged by the OO model can lead to challenges. The desire to perform a dynamic upgrade of the system is a good example. We distinguish hot swapping [25], which we define as switching one object on-the-fly, from dynamic upgrade [6, 7], which we define as switching all objects of a given type. Dynamic upgrade would be useful, for example, in applying a patch to fix a security hole, or when upgrading the version of a given object. Because everything was local, we had no easy way of performing a dynamic upgrade in K42. This is representative of issues that occur in highly modularized systems. We solved it by introducing factory objects that, when an object is created, track that object, thereby building up a notion of global state, but remained scalable by using clustered objects [4].

We have programmed some objects motivated by examples from the literature, and a few other objects allowed for by K42's clustered objects to transparently trade-off distributed versus shared implementations. However, we do not have a large base of multiple object implementations for users of the system to switch among.

Mechanistically our dynamic customizability has been a success. The implementation was relatively easy and clean. Because there has not been widespread use of the facility in K42, it is difficult to evaluate whether as a generic programming paradigm it is easy to integrate with application-level programming. To aid in this direction we are exploring automatic ways to utilize the technology [27]. However, the best method to validate the usefulness of this technology is for the wider research community using K42 to implement many different alternatives to allow the user community choices, and evaluate whether the users make effective use of the different implementations.

A reason that dynamic customization is not widely available is that operating systems like K42 that have implemented it do not have a large user base, so many alternative object implementations have not been written. In operating systems such as Windows or Linux, the underlying capability does not exist, and the marketplace is not demanding its implementation because the benefits are not proven by a large user base. More recently however, the ability to ap-

ply patches without having to schedule system downtime is receiving considerable interest from customers and the marketplace. This requirement could be satisfied by techniques like dynamic upgrade and is a good path for OS researchers interested in seeing this technology take hold.

2.3 Scalability

K42 was designed from the beginning for scalability. There is a design tradeoff in determining how to achieve scalability. Existing operating systems had been designed for uniprocessors, with multiprocessor support retrofitted, sometimes at significant cost. SMPs with large numbers of processors have not become as prevalent as was predicted in the mid 1990s. However, some of the technology developed for scalability in K42 will be applicable to the machines built with a few chips per box and many cores per chip as per current trends.

As we mentioned in the previous section, an object-oriented design was a key towards achieving scalability. Clustered objects [3] were built following this model. They paid off. Clustered objects provide a good model for how to scale OS resources as the demand for the underlying managed resource grows. They provide an interface behind which, transparently to the client, an appropriate level of distribution can be implemented. For example, the process object was implemented as shared version and later as a distributed version. The researchers who had been involved with scaling previous operating systems agree that the clustered object model makes scaling easier.

We invested in a couple other pieces of infrastructure that helped us achieve scalability in K42. Our IPC model helped achieve scalability by automatically localizing requests from client to servers. We implemented processor-conscious memory allocators. These allocate memory on a specific processor or node, and track that memory so when it is freed it returns to the correct pool. We employ fine-grain locking. We also avoided locking hierarchies. Instead, in K42's programming model, locks are not held when calling other objects. This simplifies avoiding deadlock, but increases difficulties with timing windows of multiple in-flight calls. Also in some situations it increased programming complexity because it imposed additional constraints on cross-object calls.

While we have been successful in scaling K42, the result is tempered by the fact that we have run only on a 24-way system. While it scaled well that far, better than other operating systems, that does not mean it would scale to 1000s of processors. Likely, some issues would arise. The question is could they be addressed reasonably easily. Demonstrating the infrastructure by showing scaling to a larger number of processors would be more convincing. This was not a lack of interest and time but of available hardware.

2.4 Integrated Performance Monitoring

From K42's beginning, and throughout the project, we integrated an infrastructure that allowed for correctness debugging, performance debugging, and performance monitoring of the system [26]. We place such a high priority on this capability that at boot time we dedicate a piece of memory for this purpose. This has proven extremely valuable throughout the life of K42, not only for helping understand the performance of K42 itself, but also for understanding the performance of applications on K42. In fact, we have had Linux users run their applications on K42 so they could utilize the performance monitoring infrastructure.

In retrospect, it is obvious that such an infrastructure would prove useful. However, both on this project, and in a previous one, there was initial resistance to putting in such an infrastructure, and such resistance continues in Linux, for example. In both cases though, the infrastructure solved problems that would otherwise have been difficult or unsolvable without such integrated performance monitoring support. Contrary to some espoused wisdom, simple examination of the code can not uncover some types of performance issues. The infrastructure provides both a horizontally (different components within a given layer) and vertically (different layers in the execution stack) integrated framework. Horizontal integration allows, for example, anomalies between the virtual memory system and the scheduler to be detected because each use the same interface. This integration is not uniformly agreed as the way to go. For example, an approach advocated in Linux is for each subsystem to have its own patched version. Integration applies to more than just horizontally across the OS, but vertically to all the layers in the execution stack including hardware. Our recent work on PEM [27, 12] demonstrates the effectiveness of vertically integrated performance data.

The model we chose to use in K42 allows efficient logging of data from kernel and user space. In addition we have defined aggregation interfaces similar to Sun's DTrace or Linux's System Tap. The advantage we have found from the additional functionality, is that while aggregation is useful for many problems, having an efficient mechanism for getting out all the data is invaluable for detailed visualization or for post-process analysis [12]. Unlike some other systems, we have been willing to "clutter" up our code with static trace points. This provides the advantage of allowing more efficient data gathering, versus dynamic tracing, and thus allows greater amounts of data to be available for post-processing.

Because this effort was well-integrated from the beginning, the developers of K42 have used the tracing infrastructure throughout the project. It has been used to correctness debug the filesystem, to performance debug odd behavior in our polling mechanisms, and to performance tune our scalability through lock analysis. Perhaps though, the most important testimony is that is has been used by researchers outside the K42 group in preference to other operating systems' performance monitoring infrastructure to understand and characterize application behavior.

2.5 Linux Compatibility and Sandboxing

Near the beginning of the project we decided that it would be important to support an existing interface that had a large base of users and applications, and minimize our investment in supporting hardware. Thus, we chose to support a Linux API and ABI. In order to be fully functional and to minimize our investment in supporting specific hardware, we chose to also use Linux device-driver code. While the decision to implement a complete OS is the focus of Section 3, here we describe our specific experiences with the Linux API, and sandboxing low-level Linux code for use in the kernel.

We provide a common API by incorporating non-performance-sensitive portions of code from Linux and implementing ourselves the parts needed to fulfill K42's goals. In areas where a glibc interface impedes performance, applications can invoke a native K42 object directly. We have implemented significant Linux functionality, enough so that we could run the IBM JVM J9, the IBM database DB2, Supercomputing ASCII benchmarks, and various SPEC benchmarks, . The challenge of fully supporting Linux was not the system call interface, but features like /proc and SysV shmem and semaphores.

To minimize our investment in supporting hardware we used Linux device-driver code. Because early device code contained certain firmware expectations, we were also strongly encouraged to use much of the Linux boot code. What started off as a minimal amount of Linux functionality, piece by piece snowballed into a large intrusion into K42. It is difficult to take just a small portion of some large infrastructure because those pieces make use of other parts of the large infrastructure and have assumptions about the environment. For example, Linux device drivers make specific assumptions about the concurrency and preemption model. These assumptions did not match K42's original model and additional sandboxing was needed to maintain our model while still providing the environment the Linux code needed. As more Linux code was used, there was pressure to use even more, and our sandboxing environment became more complicated.

Another issue is that Linux is a rapidly moving target and trying to stay current with our support is challenging. Even code that we believed relatively stable gets outdated quickly as other portions of the OS progresses. In short attempting to support a wide range of hardware while still providing a common API requires a lot of investment in "uninteresting" infrastructure that detracts from a project's main goals. We discuss the consequences of this and the implications for the OS community more in Section 3.

2.6 Cats and Dogs

There are several more experiences that do not fit neatly into any of the other sections.

Early in the project we invested in optimizing the exception-level paths. Days were spent optimizing away a couple of register uses saving handfuls of cycles. While early on this appeared useful and improved micro-benchmark results, once we incorporated other code that was less careful about such optimizations, all the early efforts were swamped.

Nevertheless, some portions of the K42 low-level paths such as IPCs between servers in different address spaces continue to pay off. In K42, we can perform an IPC without context switching to the kernel. We do not need to use any kernel buffer space and only a handful of registers need to be saved, most are passed through intact. The low-level exception code authenticates the originator of the IPC. Servers can then validate that the calling client has access to make a given request. This was a clean idea and has served us well.

Another area of K42 that worked well were the decisions we made regarding scheduling. In conjunction with our IPC mechanism, we use hand-off scheduling [10] to efficiently switch between different address spaces. The only downside is that because the kernel is not involved, we could not do accounting on this transition should we desire it. We also provide a fully user-level thread scheduling capability. All threads within a given address space are multiplexed on a single kernel entity. This binds few kernel resources and provides efficient thread scheduling. We do not block the user-level scheduler when one of its threads blocks for I/O, instead we return control and it blocks only the thread. This model also worked well. Finally, frustrated with the Unix model of priority scheduling and the lack of ability to reason about what it was actually doing, we implemented a proportional-share scheduler tempered with five fixed-priority bands. This allows full control of I/O versus CPU threads, to really `nice` a process, and to provide for both soft real-time and gang-scheduled jobs. While we have not fully explored real-time and gang-scheduling, the other aspects have worked out well.

Early on we predicted cheap 64-bit architectures would be mainstream and we designed our code to take advantage of that, for example by using sparse virtual addressing. Given this expectation, the delays in wide availability of 64-bit hardware hindered our ability to run K42 on widely available machines. Thus, it was much later before collaborators had easy access to running K42. This is now remedied and K42 is being ported to more readily available platforms allowing wider use.

Similar to the Exokernel, we moved functionality traditionally in the kernel to user-space servers, and to the address space of the process. In many cases this allowed short-circuiting what would typically be system calls. For example, for small files we track the cursor position and map in the file data to the process's address space. This type of work can occur in other operating systems and with a correct authentication model is a win.

K42 is open source. We believe this, combined with the OO design, allows developers a novel opportunity to contribute code [2]. Many Linux developers write code that is useful to a subset of Linux users, but is never taken back into the main kernel, e.g., scalability patches. K42's model of customizability, in which objects only need affect users that invoke them, provides a unique opportunity for users to contribute code back to the K42 base without the tension of determining how widely applicable it is.

3. IMPLEMENTING A COMPLETE OS

Without a doubt, the biggest frustration of the team was the immense amount of time and effort we spent implementing and supporting the Linux API, which was motivated by the goal of implementing a complete operating system. We use "a complete OS" to mean a fully-functional system that could be used by a large community, and that would run a large set of existing significant applications with no porting. The intent to build a complete OS was motivated by several factors.

First, demonstrating the value of the project to potential users can better be accomplished by running real applications and providing a familiar environment. Second, stated

in the positive direction, the ability to produce results on meaningful applications beyond microbenchmarks is valued by the academic community. This was supported by early feedback on K42 before it was as complete as it is now. Third, the intent of the project was to provide more than research results, it was to offer a real alternative solution.

While this may seem ambitious, the alternative is to accept that the fundamental structure of Linux or Windows is the right solution for ten years from now; if that is plausible, then what about twenty or thirty years out. The operating systems research community has undertaken increasingly fewer new complete OS efforts. It is not clear if this is because researchers feel that Linux or Windows is the right answer, or if new complete OS research has become too costly, or even potentially if the increasing maturity and competitiveness of the OS research field has created an environment that discourages such attempted innovation because of publication requirements. Whatever the reason, we believe it is important for the OS research community to drive whole-OS innovation in addition to incremental improvements. In addition to being a complete OS, K42 is a good vehicle on which to try new research approaches.

While we acknowledged we needed to support legacy interfaces, we did not recognize early in the project the amount of effort and the impact that fully supporting them would involve. Having to support the increasing number of legacy interfaces will weigh heavily on future systems. What is needed therefore, is technology that will allow the OS community to phase out old interfaces and introduce new ones, while still providing a usable computing platform.

With K42 we believed we could address these issues. The new flexible customizable structure we definitely had. The thought that some objects could be written to support the needed legacy interfaces and then switch to new ones seemed plausible. It failed. In part, it failed because we did not recognize ahead of time the significant ramifications of needing to support legacy interfaces with reasonable performance. Perhaps if we had pursued the personality independent research angle we had started with or if instead we had used hot swapping to instantiate new objects when an application `forked`, we may have been able to avoid affecting the internal structure as dramatically. Time and resources however dictated that we take a more expedient route that involved a deeper intertwining of the Linux and native K42 code. Fully supporting the Linux API significantly changed the structure of K42.

In retrospect, we chose an inopportune place to separate out what we utilized from Linux and what we implemented to achieve our goals. In a successor "library OS" project called Libra [1], we chose to package performance-critical kernel functionality with the application and use virtualization to call out to a standard Linux to handle non-performance-sensitive requests. Innovative designs for managing kernel-level resources are linked in with the application from a library of such functionality allowing the customization that proved effective in K42, but without the burden of supporting legacy interfaces. Instead these calls, such as miscellaneous requests to /proc can be shipped to the companion Linux OS. This approach offers potential to allow system stack innovation while supporting applications written against legacy interfaces. In a paper in this issue [11] we provide early insight into the effectiveness of this type of approach and vision into its potential.

4. CONCLUSIONS

Nearly ten years ago, the K42 team embarked on the ambitious journey of designing and implementing a kernel that would scale to the increasing number of shared cores, that would be customizable allowing the increasingly disparate application set to have resources managed as needed by the individual applications, and that would be more maintainable in the face of an increasing amount of functionality and platforms that operating systems need to support. In this paper we provided experiences and insight into what we did right and what we did not, and how those experiences apply to other OS efforts.

We were successful with the techniques we employed to allow an operating system to scale and be customized. In both areas, we demonstrated good results and published our work. Whether the techniques we applied for maintainability will be effective remains to be decided.

Another goal of the project was to implement a complete operating system supporting legacy interfaces usable by a large community. We are currently short of this mark. Whether these difficulties are endemic is unclear. We believe that some of the more recent virtualization work provides the potential to achieve legacy support while allowing innovation. We hope other research is undertaken, and that research on K42 continues, to better answer this.

Status and Availability

K42 now runs significant applications, including web servers such as Apache, scientific libraries such as MPI, JVMs such as J9, and databases such as DB2. Recent work has significantly eased the effort needed to get K42 built and running. K42 is available open source under an LGPL license and may be downloaded via instructions from our web page at `www.research.ibm.com/K42`. Also there, are papers on K42, pointers to the K42 discussion list, and suggestions on how to participate in this research project.

Acknowledgments

A kernel development project is a massive undertaking, and without the efforts of many people, K42 would not be in the state it is today. In addition to the authors, other current K42 core team members include: Jonathan Appavoo, Maria Butrico, Amos Waterland, and Jimi Xenidis. The following people have contributed much appreciated work to K42: Reza Azimi, Andrew Baumann, Michael Britvan, Patrick Bridges, Chris Colohan, Phillipe DeBacker, Khaled Elmeleegy, David Edelsohn, Raymond Fingas, Hubertus Franke, Ben Gamsa, Garth Goodson, Darcie Gurley, Kevin Hui, Jeremy Kerr, Edgar Leon, Craig MacDonald, Mark Mergen, Iulian Neamtiu, Michael Peter, Jim Peterson, Eduardo Pinheiro, Bala Seshasayee, Rick Simpson, Livio Soares, Craig Soules, Michael Stumm, David Tam, Manu Thambi, Nathan Thomas, Gerard Tse, Volkmar Uhlig, Timothy Vail, Manjunath Gorentla Venkata, and Chris Yeoh.

5. REFERENCES

[1] G. Ammons, J. Appavoo, M. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a jvm in a virtualized execution environment. In *VEE (Virtual Execution Environments)*, San Diego CA, June 13-15 2007.

[2] J. Appavoo, M. Auslander, M. Butrico, D. da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[3] J. Appavoo, D. da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, and J. Xenidis. Experience distributing objects in an smmp os. *TOCS*, 0(0):0, 2008.

[4] J. Appavoo, K. Hui, M. Stumm, R. Wisniewski, D. da Silva, O. Krieger, and C. Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, pages 3–8, 2002.

[5] A. Baumann, J. Appavoo, D. da Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Workshop of Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*, pages 21–27, Boston Massachusetts, October 9 2004.

[6] A. Baumann, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *USENIX Technical Conference*, pages 279–291, Anaheim, CA, April 2005.

[7] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX Technical Conference*, Santa Clara, CA, June 2007.

[8] A. Baumann, J. Kerr, J. Appavoo, D. D. Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proc. of 6th Linux.conf.au (LCA)*, Canberra, April 2005.

[9] B. N. Bershad, S. Savage, P. Pardyn, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating System Principles*, 3–6 December 1995.

[10] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[11] M. Butrico, D. D. Silva, O. Krieger, , M. Ostrowski, B. Rosenburg, D. Tsafrir, E. V. Hensbergen, R. W. Wisniewski, and J. Xenidis.

[12] C. Caşcaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT'05))*. IEEE Computer Society Press, September 2005.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, pages 251–266, 3–6 December 1995.

[15] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM Press.

[16] J. Hennessy. The future of systems research. *Computer*, 32(8):27–33, August 1999.

[17] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: Challenges and opportunities. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005. USENIX.

[18] The K42 operating system, http://www.research.ibm.com/k42/.

[19] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a real operating system. In *Proceedings of EuroSys'2006*, pages 133–145. ACM SIGOPS, April 2006.

[20] D. Milojicic. Operating systems — now and in the future. *IEEE Concurrency*, pages 12–21, Jan-March 1999.

[21] J. Mogul. Operating systems should support business change. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, June 2005. USENIX.

[22] R. Pike. System software research is irrelevant. http://herpolhode.com/rob/utah2000.pdf, February 2001.

[23] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. An introduction to the architecture of the VINO kernel. Technical report, Harvard University, 1994.

[24] L. Soares, O. Krieger, and D. D. Silva. Meta-data snapshotting: A simple mechanism for file system consistency. In *SNAPI'03 (International Workshop on Storage Network Architecture and Parallel I/O)*, pages 41–52, 2003.

[25] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Technical Conference*, pages 141–154, San Antonio, TX, June 9-14 2003.

[26] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, Phoenix Arizona, November 17-21 2003.

[27] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Caşcaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *PAC2 - Power Performance equals Architecture x Circuits x Compilers*, pages 15–24, Yorktown Heights, NY, October 6-8 2004.