K42 Overview

Jonathan Appavoo
Marc Auslander
Dilma DaSilva
David Edelsohn
Orran Krieger
Michal Ostrowski
Bryan Rosenburg
Robert W. Wisniewski
Jimi Xenidis

K42 is an open-source research kernel for cache-coherent 64-bit multiprocessor systems. K42 focuses on achieving good performance and scalability, providing a customizable and maintainable system, and being accessible to a large community through an open source development model. To that end, K42 fully supports the Linux API and ABI and uses Linux libraries, device drivers, file systems, and other code. In this paper we present a brief overview of K42, describe the goals of K42 and the core technologies we used to achieve those goals. More detailed descriptions of specific technologies and OS services are available in separate white papers.

1. Introduction and Goals

The K42 project is developing a new operating system kernel incorporating innovative mechanisms, policies, and modern programming technologies. Most existing operating systems were designed using technology that was current when those systems were new but that is now outdated. Their designs use centralized critical code paths, global data structures, and global policies. We believe that major structural changes, not just incremental modifications to existing systems, are needed to achieve excellent performance in a maintainable and extensible system.

We don't intend to introduce a new personality that would require the porting of applications, but instead intend K42 to be Linux API- and ABI-compatible. Also, we hope to exploit the rich set of device drivers, file systems, and other code available with Linux, and to be a part of the community that is developing core kernel technology. Therefore we are focused only on designing and implementing a new operating system kernel, allowing us to maintain external and internal Linux compatibility. Our support for the Linux API and ABI makes our system available to a wide base of application programmers, and our modular structure makes the system accessible to the community of developers who wish to experiment with kernel innovations.

In designing K42 we've tried to 1) structure the system using modular, object-oriented code, 2) avoid centralized code paths, global data structures, and global locks, and 3) move system functionality from the kernel to server processes and into application libraries. While we believe in these design principles, we're willing to make sensible compromises for the sake of performance. We're not a research project that carries its design philosophies to extremes in order to fully explore their ramifications.

Key goals of the K42 project include:

- **Innovation.** Develop and explore innovative technologies and provide an environment that allows easy prototyping of new operating system technologies.
- **Performance.** A) Scale up to run well on large multiprocessors and support large-scale applications efficiently. B) Scale down to run as well on small multiprocessors as kernels that do not scale up. C) Support small-scale applications as efficiently on large multiprocessors as on small multiprocessors.
- **Wide availability.** A) Be available to a large open-source and research community. B) Support Linux external and internal interfaces. C) Make it easy to add specialized components for experimenting with policies and implementation strategies. D) Open up for experimentation parts of the system that are traditionally accessible only to experts.
- **Customizability.** A) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. B) Let the system adapt to changing workload characteristics.
- **Applicability.** A) Effectively support a wide variety of systems and problem domains. B) Make it easy to modify the operating system to support new processor and system architectures. C) Support systems ranging from embedded processors all the way up to high-end enterprise servers.

The rest of this paper is organized as follows. In Section 2 we briefly describe the structure of K42. Section 3 describes some of the different technologies we employ in K42 and the approaches we have taken in its design and implementation. In subsequent sections we examine two of these key technologies in more detail: our user-level implementation of system services in Section 4 and our object-oriented model in Section 5. Section 6 concludes. For more in-depth information on particular subjects see our other white papers: Clustered Objects in K42, K42's Memory Management, K42's Filesystems, K42's Performance Monitoring and Tracing Infrastructure, The K42 Linux Environment, On-The-Fly Object Switching in K42, Utilizing Linux Kernel Components in K42, Real-Time in K42, Scheduling in K42

2. Overview of K42's Structure

K42 is structured around a client-server model (see Figure 1). The kernel is one of the core servers. It currently provides memory management, process management, IPC infrastructure, base scheduling, networking, device support, etc. [1] Above the kernel are applications and system servers, including the NFS file server, name server, socket server, pty server, and pipe server. For flexibility, and to avoid IPC overhead, we implement as much functionality as possible in application-level libraries. For example, all thread scheduling is done by a user-level scheduler linked into each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented technology. All inter-process communication (IPC) is between objects in the

1. In the future we plan to move networking and device support into user-mode servers.

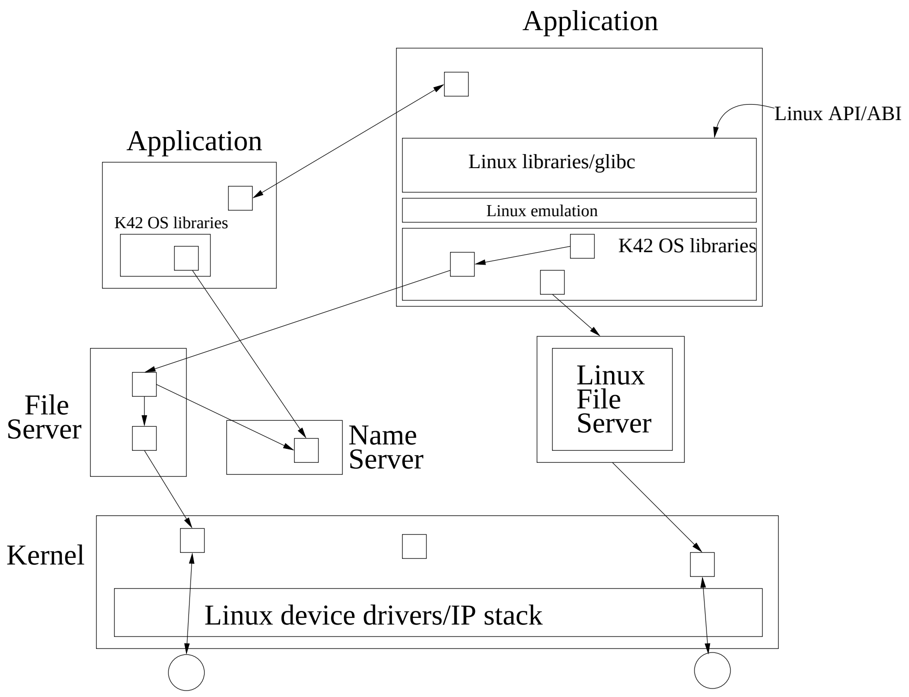

Figure 1. Structural Overview of K42

client and server address spaces. We use a *stub compiler* with decorations on the C++ class declarations to automatically generate IPC calls from a client to a server, and have optimized these IPC paths to have good performance. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls.

From an application's perspective, K42 supports the Linux API and will support the Linux ABI. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When writing an application to run on K42, it is possible to program to the Linux API or directly to the native K42 interfaces. All applications, including servers, are free to reach past the Linux interfaces and call the K42 interfaces directly. Programming against the native interfaces allows the application to take advantage of K42 optimizations. The translation of standard Linux system calls is done by intercepting glibc system calls and implementing them with K42 code. While Linux is the first and currently only personality we support, the base facilities of K42 are designed to be personality-independent.

We also support a Linux-kernel "internal personality". We are developing a set of libraries that will allow Linux-kernel components such as device drivers, file systems, and network protocols to run inside the kernel or in user mode. These libraries will provide the run-time environment that Linux-kernel components expect. This infrastructure will allow K42 to use the large code base of hardware drivers available for Linux.

The rest of the paper contains more details on the motivation behind the structure we have outlined here, as well as other technologies and mechanisms utilized in K42.

3. K42 Technology

To engineer an operating system to perform and otherwise behave well, a number of technologies are needed. These often act in concert, supporting each other. We list some of the notable examples used in K42. Many of them are explored in greater detail in other white papers.

- In K42 much of the functionality traditionally implemented in the kernel or servers is implemented in libraries in the application's own address space. This provides for a large degree of customizability because applications can implement system functionality using libraries customized to their needs. Overhead is reduced in many cases by avoiding crossing address spaces to invoke system services. Also, space and time overhead is consumed in the application and not in the kernel or servers. This is described in more detail in Section 4.
- We have applied an object-oriented technology to the entire system, where each virtual (e.g. virtual memory region, network connection, file, process) and physical (e.g. memory bank, network card, processor, disk) resource is managed by a different set of object instances[Auslander97,Gamsa99,Kriege Each object encapsulates all the meta-data necessary to manage the resource as well as all the locks necessary to manipulate the meta data. We avoid global locks, data structures, and policies. Key aspects of this technology (described in more detail in Section 5) include:
  - To achieve good performance, the objects used to implement a service can be customized (by the application or OS) to the demands on that service. Moreover (and for high-availability), objects can be swapped to new implementations without taking the system down and while they are in use.
  - Good multiprocessor performance is achieved because: 1) independent requests to different resources proceed independently; no shared data structures are traversed and no shared locks are accessed, and 2) good locality is achieved for resources accessed by a small number of processors, and 3) our *clustered-object* technology [Gamsa99] lets widely accessed objects be implemented in a distributed fashion.
  - The modular nature of the system makes it maintainable. Also, programmers can contribute code back to the K42 base affecting only applications that choose to use those contributions.
  - All interaction between applications and servers are directed to objects. Stub-compiler technology we have developed allows new interfaces to be easily added (for clients that are aware of them) and allows servers full freedom in the implementation of an object interface (i.e., there is no need to derive two implementations of an interface from a common C++ class). A capability like authentication service is provided by the stub-compiled code.
- To support Linux applications and kernel components, we support the external and internal (device driver/file system/IP stack) interfaces and execution models. We expect to run Linux application binaries without re-compilation. We have also developed technology that allows Linux components and libraries to be used without modification; this is critical to be able to maintain compatibility. Also, we do this without for the most part compromising K42 goals. For example, we support the Linux device driver model which assumes a non-preemptable kernel, while still being fully preemptable. [2]

2. For an explanation of support of Linux drivers in the K42, see the Utilizing Linux Kernel Components in K42 white paper..

- K42 is designed to be easily ported to new hardware, and then subsequently tuned to exploit architecture-specific features of the target hardware platform. For easy portability, default implementations of services are provided in a machine-independent fashion. The default implementations depend on a small number of primitives. Machine specific implementations of these services can be provided to tune the system to an architecture. Because there are no machine-independent data structures that machine-specific code must support (e.g., Linux's two-level page table), few constraints are imposed on the machine-specific implementation. For example, we are able to exploit such machine features such as the PowerPC inverted page table, and the MIPS software-controlled TLB, without compromising the overall design or the other machine implementations.
- Much of the system functionality is implemented in user-level servers[Liedtke96]. Along with systems such as L4 [Liedtke95], K42 maintains good performance via an efficient IPC mechanism that has performance comparable to system calls. This facility requires no kernel storage for messages or authentication. In addition to fast IPC, K42 uses shared memory communications between clients, servers, and the kernel to further reduce communications cost.The strategy for moving function from the kernel to servers is pragmatic and respects performance.
- We have focused the design of K42 on the needs of multiprocessors, and more specifically NUMA multiprocessors. NUMA multiprocessors can scale up to thousands of processors requiring that system software take into account memory locality to achieve good performance. Our scalability goal is not only to support large systems, and applications that may span the entire system, but also to run sequential and small-scale parallel applications as efficiently as they run on a small scale multiprocessor. A major investment has been made in IPC, locking, and memory allocation infrastructure to enable the system to exploit locality in application requests. In many cases our mechanisms have fundamentally different designs from facilities designed for uniprocessor operating systems, but perform just as well on a single processor as these other facilities, and achieve better scalability. To accomplish this, we make extensive use of *processor specific* memory, which is memory where the same virtual range maps to different physical addresses on different processors. This allows us to efficiently address local resources.
- K42 was designed to run on 64-bit processors. The dependence on 64 bits enables pervasive implementation optimizations. Examples include the use of large virtual arrays rather than hash functions, the allocation of memory bits for distinguishing classes of allocated memory, and exploiting the fact that we can atomically manipulate 64-bit quantities efficiently.
- K42 is fully preemptable and most of the kernel data is pageable. Except for low-level interrupt handling and code for dispatching real-time applications, K42's threading model allows the kernel to be preempted at any point. This provides for low-latency interrupt handling. Only the kernel code and the data of low-level objects is pinned. This significantly reduces the kernel's footprint and provides more physical memory for applications.
- We are developing a scheduling infrastructure that can provide quality-of-service guarantees for processors, memory, and I/O, and that simultaneously supports real-time, gang-scheduled, regular (time-shared), and background work. K42 uses synchronized clocks (hardware or software) on different processors to allow work to be scheduled simultaneously

for short periods of time on multiple processors. This ability to support fine-grained gang-scheduled applications simplifies the task of scientific programmers and game developers. This is because they can write their applications to use a fixed set of resources and still deploy those applications in a general purpose system running interactive and real-time applications.

- Traditionally, the error of using a "stale" pointer to deleted storage is avoided by using "existence locks" or use counts to protect pointers. Full-scale garbage collection can also solve this problem, but is not appropriate for low-level OS code. K42 uses a new mechanism, in which deletion of K42 objects is deferred until all currently running threads have finished[McKenney01,Gamsa99] . This allows a programming style whereby an object releases its own lock before making a call on another object, thus improving base system performance, increasing scalability, and eliminating the need for complex lock hierarchies and the resulting complex deadlock avoidance algorithms. This last point is critical, because it means that a high level of sophistication is not needed to develop core OS code. The technique used is related to *type safe* memory [Greenwald96], but minimizes the amount of time during which the memory is guaranteed to be type safe.

4. User-Level Implementation of System Services

In K42 much of the functionality traditionally implemented in the kernel or servers is moved to libraries in the application's own address space. This work has a similar flavor to the Exokernel [Engler95], Psyche [Marsh91], and Scheduler Activations [Anderson91] work. This change allows for a large degree of customization because applications can implement traditional system functionality using libraries customized to their needs. For example, applications with specialized needs (e.g., games, subsystems, scientific applications) can provide their own libraries, replacing much of the OS functionaity that would traditionally be implemented in the kernel or system servers, without sacrificing security and without impacting the performance of other applications. Overhead is reduced in many cases because crossing address space boundaries to invoke system services can be avoided. Also, space and time overhead is consumed in the application and not in the kernel or servers. For example, an application can have a large number of threads without consuming any additional kernel pinned memory. In many cases, we are able to handle common-case critical paths (e.g., for a non-shared file) efficiently at user-level, while handling more complex situations (e.g., multiple applications accessing the same file) in the kernel or a system server.

Implementation in the application's address space impacts the design of many operating system services. The implementation is not necessarily more difficult, but it is different. So far, we have been able to develop implementations for these services that are as efficient as those of other operating systems. Moreover, in some cases we have found implementations that are more efficient. In this section we review some of the services that we have implemented largely at user-level, and discuss our experience.

4.1. Thread Scheduling

All thread scheduling has been moved to user level. The kernel is aware only of user processes and maintains one kernel-level entity, which we call a *dispatcher*, representing that process. [3] All threads are multiplexed at user level on this dispatcher. Events that would ordinarily block the process are instead reflected back to the scheduler library code running in the application. This scheduler code can then take the appropriate action, for example blocking the current thread and running another thread. In this way, any number of threads can be multiplexed on a dispatcher without negative consequences for the user application. This ties up fewer kernel resources, makes the scheduling more efficient, and most importantly, allows flexibility for optimizations at user level. This user-level scheduling facility provides a framework that has allowed other services to be moved to user level.

4.2. Timer interrupts

If an application has thousands of threads, many of those threads may be waiting for timer events (e.g., timeouts on socket operations). With K42, the dispatcher has a single timer request outstanding, for its *next* timeout, and all subsequent timeouts are maintained in the application's address space. This in fact results in better performance because most timeouts are to handle exceptional events, and are canceled without ever occurring. By keeping the state in the application address space, we avoid interaction with the kernel when a timeout is canceled, providing an inexpensive mechanism for the common-path timer operation.

When a timer event for a dispatcher actually occurs, it is passed up to the dispatcher as an asynchronous notification. The dispatcher code can unblock any thread or threads that were waiting for event and can then make an informed decision as to whether to resume whatever it was doing or to instead switch to one of the newly-enabled threads. User-level code, rather than the kernel, makes that decision.

4.3. Page Fault Handling

On a page fault, we maintain the state of the faulting thread in the kernel only long enough to determine if the fault was in-core (i.e., the page is already resident in the page cache or just needs to be zero filled, etc.). If it is in-core, the kernel handles the fault directly. Otherwise, the fault is reflected back to the dispatcher, the dispatcher schedules another runnable thread or yields. This contrasts with existing systems that use an M on N thread model (having M user level threads multiplexed on N kernel threads), where another kernel-level thread is scheduled when a page fault occurs, and not another user-level thread within the same kernel thread.

As with timer events, page-fault completions are passed up to the dispatcher as asynchronous notifications, and the dispatcher code can decide whether or not to switch threads when faulting threads become runnable.

The cost of saving the state in user level is only incrementally greater than the cost of a kernel state save. This is because we carefully avoid saving or copying registers unnecessarily.

The user-level functionality provided for page-faults enables customizations.[Hand99]. For example, applications that don't use floating point registers can avoid saving that state. More im-

3. In reality, there is a dispatcher for each processor on which the application runs, and for each level of service required by the application. See white-paper on scheduling in K42.

portantly, specialized applications, e.g., work-queue-based scientific applications, can be written efficiently because kernel threads are not blocked without notification.

4.4. IPC Services

The IPC services implemented in the K42 kernel are very basic; the kernel hands the processor from the sender to the receiver address space, keeping most registers intact, and giving the receiver an unforgeable identifier for the sender [4] . Most of the work of IPC is done in user-level libraries that are responsible for marshaling and de-marshaling arguments into registers, setting up shared regions for transferring bulk data, and authenticating requests.

The K42 IPC facility is as efficient as the best kernel IPC facilities in the literature [Liedtke95,Haeberlen00]. However, because the implementation is in user-level, it can be customized to, for example, use problem domain specific transports for efficiency, minimize authentication overhead, and/or minimize state saving when communicating between trusted parties.

4.5. I/O Servers

In most client/server operating systems, servers maintain state for every outstanding request from a client thread (often, this state is maintained by blocking a server thread). In K42, as a general policy, if the resources necessary to handle a request are unavailable, the server returns an error, the application blocks the thread in its own address space, and the server notifies the client when a request can be re-issued. For example, servers that provide services such as sockets, ptys, and pipes maintain information about all the applications attached to a communication port, and notify the applications when new data becomes available. While we first introduced this scheme in order to avoid using up server resources, it turns out that it has two other benefits. First, complete state about the file descriptors an application is accessing is available in its own address space space. This means that operations like the Posix **select()** call can be efficiently implemented without any communication with the kernel or servers. Second, and more importantly, it allows us to use an event rather than a polling (e.g., **select()**) model for handling I/O requests. This allows more efficient implementations of, for example, web servers [Banga99], because there is no need to block threads for long periods of time. Also, we plan on using event interfaces to better enable real-time tasks by avoiding implicit blocking semantics on I/O requests.

5. Object Model

K42 uses a modular structure with independent object instances managing each physical and virtual resource in the system. This *building-block* technology is our solution to providing customizability and to avoiding global code paths, data structure, and policies. Related work by others includes [Kohler00.Mosberger96Reid00]. In this section we discuss this aspect of our system, providing motivation for using an object-oriented design and describing its advantages from the application writer’s perspective.

4. Note, we only discuss here the standard synchronous IPC services that are the fast critical operations, other services are discussed in the scheduling white paper.

When most existing commercial operating systems were designed, object-oriented programming was less mature than it is today. For these and many other reasons, many operating systems are implemented with global code paths that traverse global data structures and implement global policies. For example, the memory management of many systems have a critical page fault path that traverses a global page cache and implements the global clock algorithm for controlling paging to disk. Problems with global policies, code paths, and data structures include:

- The policies implemented by the operating system apply to all applications including those with specialized needs such as subsystems like databases or web servers.
- Every resource instance is treated the same regardless of how it is used or its structure. For example, all files are implemented in the same way irrespective of their size or if they are being accessed in a read-only fashion.
- Applications with special needs are more complex because they must work around the limitations of the OS policies, e.g., databases often perform their own memory/disk management.
- Applications with special needs get worse performance than they would if the OS directly implemented policies that matched their requirements.
- The addition of special-purpose code or policies for specific critical programs (e.g., scientific applications, databases) typically adds extra conditional branches to critical code paths, negatively impacting the performance of other applications, and requireing full system test.
- Global data structures make it difficult to retain the state needed for special-purpose policies.
- A single implementation of any critical code path implies that a small team must control it.
- Global code paths encourage developers to violate the modularity between different parts of the system for short term performance gains resulting in long term maintainability problems. For example, many systems entangle the memory management and file system code to achieve better performance for paging. While in the short term this may improve performance, it makes code more difficult to maintain, and in the longer term the performance degrades because of the difficulty of modifying the code to meet new constraints.
- It is difficult to scale global data structures, e.g., a global hash table results in poor multiprocessor performance. Global data structures many times also imply global locks.
- Poor locality causes sequential and small-scale parallel applications to perform poorly on a large-scale multiprocessor because any overhead needed to make the OS implementation scale impacts all applications.

An alternative to global code paths, data structures, and policies is to structure an operating system in an object-oriented fashion. The Unix Vnode interface [Kleiman86], streams facility [Ritchie84], and device driver interface are all good examples of this. In each of these cases, a well-defined interface behind which different implementations can be provided has enabled flexibility and innovation, for example there are many Linux file systems that have explored various possible designs.

In K42 we have applied an object-oriented design to the entire system, where each virtual (e.g. virtual memory region, network connection, file, process) and physical (e.g. memory bank, network card, processor, disk) resource is implemented by a different set of object instances. For example, instead of having a global page cache, K42 maintains an independent page cache for every file being accessed. For each resource unit, an object instance of the desired type is instantiated to manage that unit.

5.1. Customizability

Per-resource object instances allow multiple policies and implementations to be supported by the system. Because each resource instance is implemented by an independent object instance, resource management policies and implementations can be controlled on a per virtual resource basis. This allows, for example, every open file to have a different pre-fetching policy, every memory region to have a different page size, and every process to have a different exception handling policy.

In specifying building-block compositions, applications choose from a set of building blocks provided by the operating system and (trusted) third party building-block providers, and specify how they are to be connected. Application writers need not be system programming experts; they do not need to write code, but only need to compose a set of predefined modules. Safety is not an issue for the same reason and because the building blocks verify type constraints when they are connected. The modularity of building-block compositions makes the system easy to maintain and extend because building-block compositions are expected to cover the vast majority of customization needs, it is only infrequently necessary to add new building blocks, and such extensions can then be restricted to trusted agents. From a security perspective, this can be viewed as similar to, for example, the installation of new dynamically-loadable device drivers in conventional systems [Rubini01].

Building blocks can be hot-swapped to new implementations without taking the system down and while they are in use. This allows K42's implementation of resources to be tuned to the dynamic characteristics of the applications using them without bringing the system down. This could be used, for example, to change the implementation of a file when it grows, from one optimized for small files to one optimized for large files.

5.2. Multiprocessor performance

K42's per-resource object instances result in good scalability on multiprocessors. An object encapsulates all the data necessary to manage a resource, as well as all the locks necessary to access that data; there are no global locks or global data structures in the system. As long as application requests are to different resources, they are handled by the system entirely in parallel. Moreover, objects accessed on just a single processor have good temporal and spatial locality.

Although an object-oriented design can help scalability, some objects, such as the file cache for a widely shared file, may be widely accessed on a large multiprocessor. We have developed a technology called *clustered objects* [Gamsa99] that allows the implementation of such objects to be partitioned or replicated across the multiprocessor. Clustered objects are a partitioned object model similar to [Homburg00,Makpangou94]. The distribution of an object incurs no overhead on common-case requests to the object, and is entirely transparent to clients of the object. More details of this approach can be found in the Clustered Objects in K42 white paper.

The customizability and hot-swapping described above is critical for our goals of scalability. For sequential and small-scale parallel applications, implementations of resources can be used that have low overhead but do not scale. As an application creates more threads, the system can swap in implementations that can handle the new demands. Hence, the system can on a large scale multiprocessor support both large-scale and sequential applications efficiently.

5.3. Client/Server interaction

All interaction between applications and servers are directed to objects. We use stub-compiler technology, where interfaces are declared via decorations to our actual C++ class definitions, thus binding our IPC mechanisms to the C++ interfaces. The interfaces are polymorphic in that the user need not be aware of the exact object instance/implementation when a call is made on a given object. For example, a FileObject could be instantiated as a SmallFileObject instance or a BigFileObject instance. This polymorphism is provided by the C++ method invocation rather than by C function tables. The stub compiler generated code marshals and de-marshals arguments for the method calls. For efficiency, it can take advantage of registers on architectures that have enough registers. New services can be easily added with interfaces specific to the service. This mechanism provides the ability to download new objects that extend existing interfaces.

The stub compiler in essence creates two objects for each interface. The first of these objects, the "Stub" object, presents the targeted interface to the client application and marshals calls to the interface over the IPC mechanism. The second object, the "X" object, receives IPC messages and de-marshals them into calls on the actual object that is exporting the given interface. The binding between the X object and the target is resolved at compile time and is based on method signatures, that is, the X object expects the target to have a set of methods conforming to a specified interface. Thus, there is no need for the target object to be related by inheritance to the X object and consequently any object can export any stub-compiled interface that it conforms to. This provides a level of polymorphism beyond that offered by C++; an object can export multiple un-related interfaces without being derived from any of them.

The stub-compiled interface technology includes a model for performing authentication. Calls across our IPC interface are tagged with a caller identification by the kernel. Associated with each call is a set of access rights that are verified by the stub-compiler generated code on the callee side of the IPC call prior to the IPC mechanism invoking the method of the class that was called. A capability-like mechanism is implemented above this service, allowing a client with the correct access rights to provide other clients with access to that same object independent of the type of the specific object.

5.4. Other aspects of object-oriented technology

As with other object-oriented operating systems [Campbell93,Hamilton93,Yokote93,Shapiro89], object-oriented technology provides K42 with a high degree of modularity. Programmers with expertise in an area of operating systems can more easily contribute code without needing to intimately know all areas of the system. For example, a programmer with file system expertise can contribute code to the file system module without having to be familiar with all the memory management code paths (it still will be valuable to understand the workings of the memory management system at a white paper level). The modular structure enforced by the underlying infrastructure reduces the need to have only a small team that controls the code. As with kernel development, the barrier for programmers writing major sub-systems or problem-domain specific code such as data bases, web servers, and scientific programs, is reduced because they don’t require as much OS expertise to make contributions.

Groups with specialized needs can develop specialized objects, contribute them back to the system, and have them used by a larger community that has the same needs. With K42, an object with a different implementation or policy can be introduced without any penalty for applications that don't use it. With Linux, many interesting policies that have been prototyped have not made it into distributions because they entail extra overhead (e.g., conditional branches) in an operating system structured with global code paths traversing global data structures.

The modular structure will facilitate researchers because competing policies can be applied to solve specific problems (rather than impacting all applications), and because multiple policies can be investigated at the same time in the same OS. This will enable a more quantitative approach to OS research. We believe K42 can be attractive as a teaching vehicle, allowing students to rapidly gain skills in OS development.

The hot-swapping technology mentioned above is not only useful for customization, it also enables high availability. Bug patches, security patches, and performance enhancements can be installed without bringing the system down.

5.5. Challenges

Object-oriented technology, because of the additional costs associated with the extra level of indirection, typically results in poor performance. While it may seem counter-intuitive, in K42 we have used object-oriented technology to achieve better performance. We buy back the extra costs of an object-oriented design by the ability the extra level of indirection gives us to customize the objects to the application and, on a multiprocessor, by the locality individual object instances provide. However, we have also found that a pragmatic approach is needed when using C++, for example, using large-grained objects to amortize the overhead, and examining the assembly code generated to discover performance bugs in the compiler.

Another challenge is that it can be difficult to achieve a global state if all the data for achieving that understanding is scattered throughout many object instances. For example, we have many objects each managing the pages for many small files. However, to effectively use a working set algorithm, a certain minimum number of pages are needed, therefore we can't run such an algorithm on what might be a natural granularity (i.e., that of each object). As another example, it is difficult to run the globally next highest priority thread when the priorities of threads are distributed throughout a series of user-level schedulers. Moving to instances of objects for each resource in the system has had tremendous advantages but has also opened interesting research issues that we have had to address.

6. Concluding Remarks

We have outlined some of the core K42 technology. This technology involves fundamental structural changes, and as a result we were not able to incrementally modify the vanilla Linux kernel. On the other hand, K42 is not a new operating system. It provides a Linux personality and supports Linux applications and kernel modules without modification. That is, it provides an alternative implementation of some of the core Linux-kernel technology, and depends on the much larger remaining Linux code for everything else.

We have made K42 available under a LGPL license to enable experimentation and to aid in technology exchange between K42 and vanilla Linux. The modular structure of the system makes it a great teaching, research, and prototyping vehicle, and we expect that policies and implementations studied in this framework will be transferred into vanilla Linux. Also, in the long term, we expect that the kind of technologies we are exploring with K42 will be important to Linux.

We have developed and validated much of the core infrastructure of K42 and are approaching full functionality. K42 currently runs on 64-bit Mips (NUMA) and PowerPC (SMP) platforms and is being ported to x86-64 and IA64 systems. We expect in the near future to achieve self-hosting, demonstrate better base performance for real applications than existing kernels, demonstrate better scalability, and demonstrate that specialized subsystems can customize the OS to achieve better performance at reduced complexity. However, many of the edge conditions are not yet addressed, with many simplistic object implementations. Many of the features described above, such as the resource management infrastructure, the gang-scheduling feature, and hot swapping are available only in prototype form.

References

[Anderson91] *Scheduler Activations: effective kernel support for the user-level management of parallelism*, Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, October, 1991, Proceedings of 13th ACM Symposium on Operating Systems Principles, 95-109.

[Auslander97] *Customization Lite*, M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm, Proceedings of the 6th Workshop of Hot Topics in Operating Systems, 1997, 43-48.

[Banga99] *A Scalable and Explicit Event Delivery Mechanism for UNIX*, Gauruv Banag, Jeffrey C. Mogul, and Peter Druschel, 1999, Usenix Annual Technical Conference, 253-265.

[Campbell93] *Designing and Implementing Choices: An Object-Oriented System in C++*, Roy Campbell, Nayeem Islam, Peter Madany, and David Raila, 36(9), 117-126, September, 1993, Communications of the ACM.

[Engler95] *Exokernel: An operating system architecture for application-level resource management*, Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr., 1995, Proceedings 15th Symposium on Operating Systems Principles, 251-267.

[Gamsa99] *Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System*, Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Micheal Stumm, 1999, ACM Symposium on Operating Systems Design and Implementation.

[Greenwald96] *The synergy between non-blocking synchronization and operating system structure*, Michael Greenwald and David Cheriton, October, 1999, In 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), 123-136.

[Haeberlen00] *Stub-Code Performance is Becoming Important*, Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig, October 2000 , First Workshop on Industrial Experiences with Systems Software (WIESS-00), 31-38.

[Hamilton93] *The Spring Nucleus: A Microkernel for Objects*, Graham Hamilton and Panos Kougiouris, June, 1993, 1993 Summer USENIX Conference, 147-160.

[Hand99] *Self-Paging in the Nemesis Operating System*, Steven Hand, 1999, ACM Symposium on Operating Systems Design and Implementation.

[Homburg00] *An Object Model for Flexible Distributed Systems*, P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge, May, 1995 , First Annual ASCI Conference, 69-78.

[K42 LKIntern] *Utilizing Linux Kernel Components in K42*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[K42 Linux Environment] *K42 Linux Environment*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[K42 Overview] *K42 Overview*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[Kleiman86] *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, S. R. Kleiman, October, 1986, Proceedings of the Summer 1986 USENIX Technical Conference, 207-218.

[Kohler00] *The click modular router*, Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, 2000, ACM Transactions on Computer Systems, vol 18:3 , 263-297.

[Krieger97] *HFS: A performance-oriented flexible file system based on building block composition*, O. Krieger and M. Stumm, 15(3), 286-321, August, 1997, ACM Trans. on Computer Systems.

[Liedtke95] *On micro-kernel construction*, J. Liedtke, 1995, Proceedings 15th ACM Symposium on Operating System Principles, 237-250.

[Liedtke96] *Toward Real Microkernels*, Jochen Liedtke, 1996, Communications of the ACM, vol 39:9 , 70-77.

[Makpangou94] *Fragmented objects for distributed abstractions*, Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro, Readings in distributed computing systems, 170-186, July, 1994, IEEE Computer Society Press.

[Marsh91] *First-Class User-Level Threads*, Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos, October, 1991, Proceedings of 13th ACM Symposium on Operating Systems Principles, 110-121.

[McKenney01] *Read Copy Update*, Paul McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni, July 2001, Ottawa Linux Symposium.

[Mosberger96] *Making Paths Explicit in the Scout Operating System*, David Mosberger and Larry L. Peterson, October, 1996, ACM Symposium on Operating Systems Design and Implementation.

[Pike93] *The Use of Name spaces in Plan 9*, Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, 27(2), 72-76, April 1993, Operating Systems Review.

[Reid00] *Knit: Component Composition for Systems Software*, Alastair Reid, Matthew Flatt, Leigh Stroller, Jay Lepreau, and Eric Eide, October, 2000, 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), 347-360.

[Ritchie84] *A Stream Input-Output System*, D. M. Ritchie, October, 1984, AT&T Bell Laboratories Technical Journal, 63(8), 1897-1910.

[Rubini01] *Linux Device Drivers*, Alessandro Rubini and Jonathan Corbet, 2001, O'Reilly.

[Shapiro89] *SOS: An Object-Oriented Operating system — Assessment and Perspectives*, M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot, 2(4), 287-337, 1989, Computing Systems.

[Yokote93] *Kernel Structuring for Object-Oriented Systems: The Apertos Approach*, Yasuhiko Yokote, 1993, Proceedings of the 1st International Symmposium on Object Technologies for Advanced Software, 145-162.