# Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel

J. Appavoo
M. Auslander
M. Butrico
D. M. da Silva
O. Krieger
M. F. Mergen
M. Ostrowski
B. Rosenburg
R. W. Wisniewski
J. Xenidis

**K42 is an open-source, Linux-compatible, scalable operating-system kernel that can be used for rapid prototyping of operating-system policies and mechanisms. This paper reviews the structure and design philosophy of K42 and discusses our experiences in developing and using K42 in the open-source environment.**

K42 is an open-source, Linux**-compatible, scalable operating-system kernel whose implementation is based on advanced programming techniques and incorporates innovative mechanisms and policies.[1–5] K42 was developed by several groups with cooperation from a number of universities, in particular University of Toronto, Carnegie Mellon University, and more recently the University of New South Wales. A community of users that includes a number of universities and national laboratories is active through a Web site and a mailing list.[6]

The main goals of the K42 project are:

***Scalability/performance***—K42 should run efficiently on a range of multiprocessors, from large to small, and should support efficiently both large- and small-scale applications.

***Adaptability***—K42 should manage system resources in a way that matches the changing needs of the running applications and that contributes to the self-regulating (autonomic) behavior of the system.

***Extensibility/maintainability***—K42 support should be extensible to additional hardware platforms or applications; upgrading the system with new components should be possible without interruption in system services.

***Open-source compatibility/customizability***—K42 should facilitate open-source collaboration and the incorporation of new features in support of the needs of specific user groups.

The following principles were adopted in the design of K42:

1. Use modular object-oriented code.
2. Avoid centralized code paths, global data structures, and global locks.

3. Move system functionality from the kernel to server processes and into application libraries.

When applying these design principles, however, we made compromises when their use conflicted with our performance and scalability goal. Our intent was not to carry the design principles to an extreme in order to fully explore their ramifications, but rather to use them to achieve a better performing, more customizable system.

K42 was designed to scale up to machines containing hundreds of processors, and down to machines with four processors. To achieve such scalability, K42 has been structured in an object-oriented manner so that each resource of the system (e.g., virtual memory region, network connection, file, process) is managed by a "per-instance" object (or set of objects). For example, when a user opens a file, a new file-object instance is created to manage that file. The object-oriented design also provides a high degree of customizability because concurrent applications can choose the way resources are managed to best serve their needs. Even within one application, different uses of a given resource may be managed differently. We have also provided self-managing (autonomic) capabilities by implementing mechanisms that allow these per-instance resources to be swapped "on the fly," that is, while the system is running.

In K42 we use the rich set of device drivers, file systems, and other code available with Linux, and we are active in the community that is developing core kernel technology. We are developing an alternative to the Linux kernel for research prototyping, not a new operating system.

By nature, the open-source system software carries with it certain requirements. Because open-source software is to be used by a wide spectrum of users, the software should be configurable to meet the needs of various groups of users and should be flexible enough so that the code base does not "fragment." The open-source development process should allow the contribution of one group to be used by other groups, and should allow developers to take advantage of the specific features of the platform they are targeting for their software. The software should run on a wide range of multiprocessors, from large (with hundreds of processors) to small (2-way multiprocessors), and should provide mechanisms and tools for developers to monitor the performance of the system.

We examine in this paper K42's capabilities as a vehicle for open-source development and explore the advantages of the open-source collaboration environment. We describe the K42 features that address the requirements listed earlier and demonstrate its successful development model for an open-source community. K42 allows developers to experiment with operating system policies, such as a new memory management policy, without requiring detailed knowledge of the entire system. Furthermore, K42's per-instance resource management allows the incorporation of function in the base K42 code that affects only specific applications. Thus, open-source developers with non-mainstream needs can have their code integrated into K42 without affecting other users. Because the overall structure of K42 is significantly different from previous UNIX** kernel implementations, developers interested in working with K42 first have to become familiar with the new design.

The rest of the paper is structured as follows. In the next section we present an overview of K42 and the technologies used in its development. In the section that follows we describe the experience we and others have had with K42. We highlight those aspects of K42's design and features that facilitate smooth open-source collaboration, and we present some of the challenges that we encountered. Then, we present selected results and describe our current research directions. We conclude with a summary of the main ideas contained in this paper.

## THE STRUCTURE AND TECHNOLOGIES OF K42

In this section we provide an overview of K42 and the technologies we used in its development. Following a description of the overall structure of K42, we describe in some detail an aspect of the system that illustrates the design principles at work. We selected for this purpose memory management, and we describe the objects involved and their role in the system. We then discuss the object-oriented design, the user-level implementation of system services, and a number of other technologies used in K42.

### Structural overview

The structure of K42 is based on the client-server model as illustrated in ***Figure 1***.[1] The kernel, shown

at the bottom of the figure, provides memory management, process management, interprocess communication (IPC) infrastructure, base scheduling, networking, and device support. Above the kernel there are two additional layers: the system servers layer and the applications layer. The system servers may include an NFS (Network File System) file server, a KFS (K42 File System) scalable file server, a name server, a socket server, and a pipe server. For flexibility, and to avoid context-switch and 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.

The implementation of all layers of K42 (the kernel, system servers, and user-mode libraries) is based on object-oriented technology. We use a stub compiler on a C++ class augmented with decorations (annotations that extend the language for additional functionality) to automatically generate IPC calls from a client to a server, and have optimized these IPC paths for performance. The kernel provides the basic IPC transport, and attaches sufficient information to the message for the receiver to authenticate each call.

Compatibility with the Linux API and the Linux ABI is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When an application is written to run on K42, it is possible to program either to the Linux API or directly to the native K42 interfaces. Programming against the native interfaces allows the application to take advantage of K42-specific optimizations. Standard Linux system calls are trapped and reflected to the K42 library loaded into the client address space. Emulation code in the library may implement a reflected system call locally, or it may make IPC calls on the kernel or other servers. (For performance reasons, we also provide a version of the dynamically loaded GNU C Library that branches directly to the K42 library for system calls, avoiding the hardware traps and subsequent reflections.) While Linux is the first and currently only "personality" we support, the base facilities of K42 are designed to be personality independent.

The Linux-kernel internal personality is provided by a set of libraries that allow Linux-kernel components

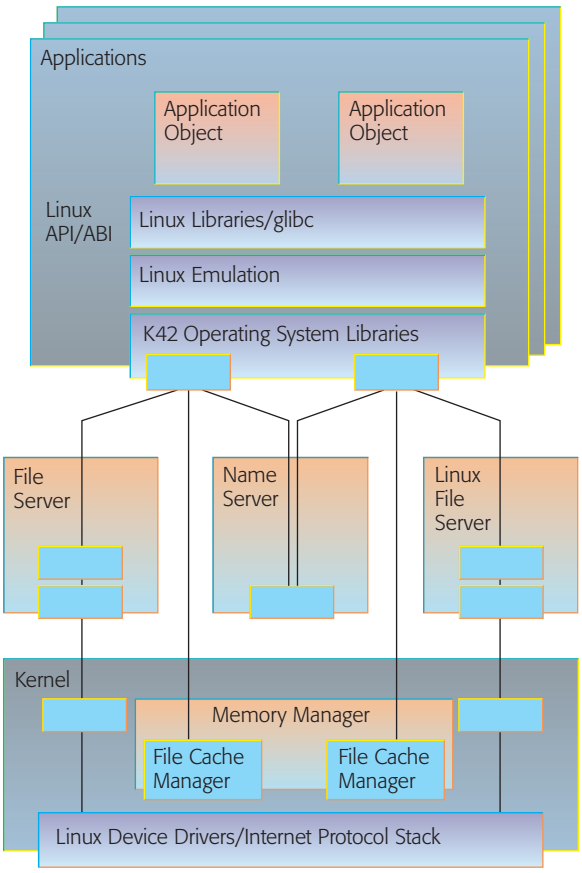


**Figure 1**
Structural overview of K42

such as device drivers, file systems, and network protocols to run inside the kernel or in user mode. These libraries provide the runtime environment that Linux-kernel components expect.

To make the structural overview above more concrete, we describe as an example the group of K42 objects responsible for handling the memory management function in K42. All the objects in ***Figure 2*** are contained in the Memory Manager component of the kernel shown in Figure 1. Next, we briefly describe the objects shown in Figure 2 and their handling of a page fault. Additional details can be found in Reference 5.

***Process,*** the root of the object tree in the figure, is the kernel object that represents a process. It maintains a list of the regions in the address space of the process and, when needed, accesses the hardware-specific information (HAT).

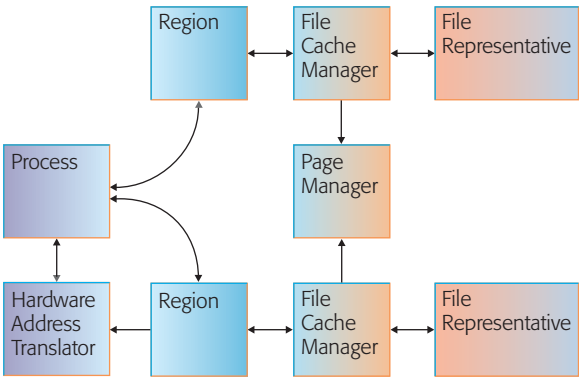


**Figure 2**
K42 memory management

***Region*** represents an area of memory and manages the mapping from a contiguous page-aligned range of virtual addresses to a contiguous page-aligned range of file offsets.

***FR*** (File Representative) is the object the kernel uses to access a file. It communicates with the file system in order to perform I/O.

***FCM*** (File Cache Manager) controls the page frames currently assigned to hold file content in memory. It implements the local paging policy for the file and supports Region requests to make file offsets addressable in virtual memory.

***PM*** (Page Manager) controls the allocation of page frames to FCMs, and thus implements the more global aspects of the paging policy.

***HAT*** (Hardware Address Translator) is the object that manages the hardware representation of an address space.

A page fault is resolved as follows:

1. The Process component identifies the Region component responsible for the faulting address and triggers its processing of the fault.
2. The Region component checks for permission violations (e.g., a write access to a read-only Region) and then sends the FCM a request to map, with the correct permissions, the corresponding file offset to the virtual address.
3. The FCM determines the page frame in which the requested file offset falls and asks the HAT to update the address-space mappings to map the faulting virtual address to that page frame. In this case, a page-fault-complete indication is passed to the low-level code that called the Process object, and the faulting instruction is retried.
4. If the FCM cannot provide a page frame immediately, it allocates a page frame and descriptor and calls its FR to schedule an I/O operation to fill the page frame. In the normal case, the fault may be processed asynchronously. The FCM queues a notification request object to the page-frame descriptor and returns an indication that the page fault has not been resolved.
5. If the FCM discovers that the requested page has already been scheduled for I/O, it proceeds as described in the previous step, but without scheduling any new I/O.
6. Sometimes a page fault cannot be handled asynchronously. This is the case if the faulting dispatcher cannot accept a page-fault notification because it is disabled or because it has exceeded a fixed limit on outstanding faults. In this case, the fault is handled synchronously.
7. The FR that starts the I/O operation for its FCM usually communicates asynchronously with its file server. As soon as the call is initiated, the kernal thread processing the fault usually terminates. When the I/O operation is complete, the file system makes an IPC call to the kernel, which then completes the page fault processing.

This example demonstrates the characteristic way in which K42 provides a common service that is found in any operating system. The set of objects used and their interactions are intended to provide an insight into the object-oriented nature of K42 and into its overall structure.

## Object-oriented design

Each resource in the system (e.g., virtual memory region, network connection, file, process) is managed by a different set of object instances. Each object encapsulates the meta-data necessary to manage the resource as well as the locks necessary to manipulate the meta-data. We avoid global locks, global data structures, and global policies.

The vast majority of requests to an operating system are independent and may be processed asynchronously, provided the underlying design and data structures permit it. Some requests, however, involve dependencies, that is, the sharing of resources. K42 provides an enhanced object-ori-

ented model by means of clustered objects[7,8] that makes the distinction between independent and dependent requests transparent (the client is not aware of it). Clients use clustered objects, and the underlying implementation determines transparently the appropriate distribution for achieving good multiprocessor performance.

We have used object-oriented methods in K42 for multiple reasons, one of which was to achieve good performance on multiprocessors. The use of object-oriented methods facilitates this because (1) when no shared data structures are traversed and no shared locks are accessed, unrelated requests to different resources can be processed independently (and thus concurrently), (2) good locality of reference is achieved for resources accessed by a small number of processors, and (3) the clustered-object technology lets widely accessed objects be implemented in a distributed fashion.

In addition, per-instance resource management allows multiple policies and mechanisms to be supported simultaneously, and this makes K42 customizable. Because each resource (or allocation unit of a resource) is accessed through a service that is implemented by an independent set of objects, resource management policies and implementations can be controlled on a per-resource basis; thus, different applications can use different resource management policies. Even within a given application, different policies may be supplied for different instances of a given resource. For example, every open file may have a different prefetching policy, and different page caches may have different replacement policies. K42 extends these customizability advantages by providing hot swapping. Hot swapping allows object instances to be replaced dynamically as an application's behavior changes, as new functionality becomes available, or as bug fixes are implemented.[8,1,4]

Finally, object-oriented technology helps to achieve other goals as well. The modular nature of the system makes it more maintainable by providing a clean model for supporting new applications and new hardware. Although for each new platform or application to be supported, additional objects may be created, these objects remain simple and easy to program. Further, because the impact of modifying a given object is limited to a small number of components, experimentation with new resource management policies and algorithms is facilitated. Although these motivations for using object-oriented technology look good in theory, their application does not always produce the expected results. In the section "Experiences," we discuss the challenges we have encountered in practice.

## User-mode implementation of system services

In K42, much of the functionality traditionally implemented in the kernel or system servers is moved to libraries that execute in the client process. This approach is similar to the one in Exokernel[9] and Psyche[10] and in Scheduler Activations.[11] The approach allows for a large degree of customization because libraries customized to the needs of the applications can be used. For example, subsystems, games, and scientific applications can provide their own libraries, replacing much of the operating system functionality that would traditionally be implemented in the kernel, without sacrificing security and without impacting the performance of other applications. Security is not affected because only information that would have been accessible to an application is stored in the library. Overhead is reduced in many cases because crossing address space boundaries to invoke system services can be avoided. Also, space and time is consumed in the application and not in the kernel or system servers. As an example, an application can have a large number of threads without consuming any additional kernel memory. In many cases, we can handle common-case critical paths (such as nonshared files) efficiently in user mode, while handling more complex functions in the kernel or in a system server. We describe now some of the services that have user-mode implementations in K42.

***Thread scheduling***—All thread scheduling has been moved to user mode. The kernel is aware only of user processes and maintains an entity called a dispatcher that handles the scheduling of all threads in user mode. Events that would ordinarily block a 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 application. This ties up fewer kernel resources, makes the scheduling more

efficient, and most importantly, allows flexibility for optimizations in user mode. This user-mode scheduling facility provides a framework that has allowed other services to be moved to user mode.[3]

***Timer interrupts***—If an application has thousands of threads, many of those threads may be waiting for timer events, such as timeouts on socket operations. In K42, the dispatcher has a single timer request outstanding, for its next timeout, and all subsequent timeouts are maintained by a user-mode function. This results in better performance because most timeouts handle exceptional events and are canceled before occurring. By keeping the state in the application address space, interaction with the kernel whenever a timeout is canceled is avoided, thus 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, and the user-mode function determines how to handle the event.

***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 is "in-core." If it is, then the kernel handles the fault directly. Otherwise, the fault is reflected back to the dispatcher, and the user-mode scheduler either schedules another runnable thread or yields. With our model, kernel resources are saved because the kernel only needs to reserve resources for one entity per address space. As with timer events, page-fault completions are passed up to the scheduler as asynchronous notifications. The user-mode functionality provided for page faults enables customizations.

***IPC services***—The IPC services implemented in the K42 kernel are simple. The kernel hands the processor from the sender to the receiver address space, keeping most registers intact, and giving the receiver a non-forgeable identifier for the sender. Most of the work of IPC is done in user-mode libraries that are responsible for marshaling and demarshaling 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.[12] The kernel provides the basic IPC transport and attaches sufficient information for the server to perform authentication on those calls. Because the implementation is in user mode, it can be customized to, for example, use problem-domain-specific transports for efficiency, minimize authentication overhead, and minimize state saving when communicating between trusted parties.

***I/O servers***—Unlike other operating systems, in K42, blocked threads waiting for I/O are blocked in the thread's own address space rather then in a server or the kernel. I/O servers notify applications about changes in the state of the I/O objects they are using (e.g., sockets becoming readable, disk requests being completed) with asynchronous messages. When changes in the state of I/O objects are communicated to a client, it unblocks the appropriate threads or launches new threads. Although this scheme was introduced in order to avoid using up server resources, it also has two other benefits. First, complete state about the file descriptors that an application is accessing is available in the application's own address space. This means that operations like Posix select( ) can be implemented efficiently without communication with the kernel or servers. Second, and more importantly, it allows us to use an event-driven rather than a polling model for handling I/O requests, making implementations of services such as select( ) more efficient. This allows more efficient implementations of, for example, web servers, because there is no need to block threads for long periods of time.

## Additional K42 technologies

We describe additional technologies that we have used in K42 in this subsection as follows: integrated performance monitoring, hot swapping, a customizable and scalable operating system, comprehensive scheduling, and lock avoidance.

### *Integrated performance monitoring*

As part of the original design, K42 includes an integrated tracing and performance-monitoring infrastructure. More recently, we have extended the model to encompass all aspects of the software stack. K42's event-tracing infrastructure provides for correctness debugging, performance debugging, and performance monitoring of the system. The infrastructure allows for cheap and parallel logging of events by all levels of the system including applications, libraries, servers, and the kernel. This event log may be examined while the system is running, written out to disk, or streamed over the network. Post-processing tools allow the event log

to be converted to a human-readable form or to be displayed graphically.

The tracing infrastructure in K42 achieves several goals:[13]

1. Provide a unified set of events for correctness debugging, performance debugging, and performance monitoring
2. Allow data gathering to be enabled dynamically by having the infrastructure always compiled into the system
3. Separate the collection of events from their analysis
4. Allow events to be efficiently gathered on a multiprocessor
5. Have minimal impact on the system when tracing is not in use and allow for zero impact by providing the ability to compile out (remove the function at compilation time) events if desired
6. Provide cheap but flexible logging for either small or large amounts of data per event.

This performance-monitoring infrastructure has proven invaluable not only in helping us achieve good performance in K42, but in aiding developers of Linux programs trying to understand the behavior of their applications.

### Hot swapping

Hot swapping allows the individual object instances used to implement a service to be tuned to the varying demands on that service. For example, in K42, when a file is accessed exclusively by one application, an object in the application's address space handles the file control structures, allowing it to take advantage of mapped file I/O, thereby achieving performance benefits of 40 percent or more. With hot swapping, when the file becomes shared, a new object can dynamically replace the old object. This new object communicates with the file system to maintain the control information. We have also used hot swapping to switch between shared and distributed implementations of the object representing a region in the kernel when we discovered the region object being used throughout the multiprocessor. Based on the hot-swapping infrastructure, a dynamic update[14] capability has been prototyped. Dynamic update is a mechanism that allows software updates, bug fixes, and patches to be applied without loss of service or downtime, thus providing greater system availability. The hot-swapping infrastructure can be used to perform dynamic-granularity system monitoring, allowing low-intensive overall monitoring to occur until a problem is detected, then high-intensive monitoring to be introduced at the trouble locations. The hot-swapping infrastructure can also be used to allow for more extensive system testing by introducing faults or delays.

### Customizable scalable file system

In addition to supporting standard Linux file systems such as ext2, K42 includes KFS,[15] a fine-grained adaptable file system that is customizable at the granularity of files and directories, allowing K42 to meet the requirements and usage access patterns of various workloads. In KFS, each file or directory may have its own tailored service implementation, and these implementations may be replaced on the fly. By doing so, KFS addresses the difficulties found in traditional file systems designed to handle a specific set of requirements and assumptions about file characteristics, expected workload, and usage and failure patterns. KFS also includes a meta-data "snapshot" capability, allowing it to have the properties of a journaled file system (JFS) with much lower performance overhead. KFS induces lower overhead than a write-ahead JFS and scales better as the number of clients and file system operations grows. KFS has also been implemented successfully in Linux.

### Comprehensive scheduling

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 (time-shared on parallel computers), 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, without the need for global synchronization. The ability to support fine-grained gang-scheduled applications can simplify parallel programming tasks, and the ability to run mixes of all classes of jobs allows developers to develop and test in an environment similar to the final production environment.

### Lock avoidance

Traditionally, the error of using a stale pointer to deleted storage is avoided by the use of existence locks or reference counts to protect pointers. Full-scale garbage collection can also solve this problem

but is not appropriate for low-level operating-system code. K42 uses an independently developed mechanism similar to the RCU[16] (Read Copy Update) mechanism, in which deletion of K42 objects is deferred until all currently running threads have finished. This mechanism allows 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. The technique used is related to type-safe memory.[17]

### *Other K42 features*

K42 was designed to run on 64-bit processors. Designing for 64-bit architecture enables pervasive implementation optimizations. Examples include the use of large virtual arrays rather than hash tables, 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 structures are 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 and requires pinning of only kernel code and data of low-level objects. Reducing the required pinned memory potentially reduces the footprint of the kernel and provides more physical memory for applications.

## EXPERIENCES

In this section we describe our experience with K42 as well as that of our collaborators. We cover the overall object-oriented design; modularity for maintainability, new applications, and new hardware; user-mode implementation; and version compatibility issues.

### Overall object-oriented design

K42's object-oriented model provides a development environment in which it is comparatively easy to bring up the operating system on a new platform. By developing simple or stubbed-out implementations behind a full and correct interface, we were able to bring up K42 quickly. For example, behind the interface to the objects that provide a file system and IP (Internet Protocol) service, we implemented a simple serial protocol to an existing machine that could provide the IP service. Component by component, we have replaced services as needed. Although there have been a few unexpected difficulties, this model has worked well. For example, when we implemented a new variant of the page cache object, its interface made only the physical address accessible, although it would have been more convenient to have had a virtual rather than a physical address. Overall however, the model has been a solid success.

One of our concerns involved the use of per-instance resource management. We feared that it might be difficult to achieve a desired global state when the data affecting that state were scattered throughout many object instances. For example, we have many objects managing the allocation of pages for many small files. To effectively use a working set algorithm, a certain minimum number of pages is needed, and thus we cannot run such an algorithm on what might be a natural granularity. As another example, it is difficult to globally select the next highest priority thread when the priorities of threads are distributed throughout a number of user-mode schedulers. Although per-instance resource management has produced advantages, it has also raised interesting research issues that we have had to address. In our experience, the difficulty of implementing local policies that produced a desired global state has not prevented us from achieving good performance. As expected, we have found that changes to specific objects were more easily made because their effects were local. This should imply that open-source developers will not need to gain an understanding of the overall system when attempting to tune the performance of a particular component.

The largest drawback of the object-oriented nature of the system is the complexity it introduces, a complexity that becomes apparent when one is trying to understand the interactions between the different components of the system. Although originally we planned to avoid the use of implementation inheritance, we later decided that in some places the benefits outweighted the difficulties introduced. These decisions, and the nature of object-oriented software, imply that understanding cross-module interactions requires wading through significant layers of interface. For open-source developers familiar with standard UNIX operating-system structure, gaining this understanding poses a barrier to K42 overall system development. Fortu-

nately, however, work on individual components requires less expertise.

We were also concerned about the assembly code generated by the compiler for the C++ code of K42. We have found that a pragmatic approach is needed. Sometimes we use larger-grained objects than we would like in order to limit the overhead, and sometimes we have had to examine the generated assembly code in order to discover the performance bugs created during the compilation. For the most part, we have found the performance of C++ code to be adequate and the necessary compromises acceptable. In a few cases we have worked with g++ (the GNU C++ compiler) developers to fix or understand particular behavior. Because we have avoided compiler-supported multiple inheritance, we have observed an increased proliferation of objects.

Although our object-oriented model supports the use of multiple objects in managing resources, the use of this ability is limited at present. As the number of applications increases, we may need to add infrastructure that records the objects used by various applications and the objects that perform best for each application. We have begun work on a continuous program optimization infrastructure that would enable this.[18]

## Modularity for maintainability, new applications, and new hardware

Our experience shows that the object-oriented model facilitates the implementation of new services as well as the porting of K42 to new architectures. New services, such as large-page or segment support, can be added with minimal impact. For example, POWER4* GPUL (GigaProcessor Ultralite) supports large pages. K42's design allowed us to write code to take advantage of these large pages with changes localized to that module. There is no duplication of structures forced by writing to a "hardware abstraction layer"—the machine-specific hardware interface can be at different levels for different architectures.

As previously stated, stubbed interfaces enable the quick bring-up of the system. We have provided unoptimized architecture-independent code for much of the machine-dependent part of K42. This allows the porting to new architectures to proceed quickly, with tuning taking place after the system was booted and running. Although our experience with new hardware and new architectures is limited, we can report favorably on two such cases: first, a porting of K42 to the AMD** x86 64-bit platform (through kernel bring-up), and second, incorporating of test code for a new memory model into K42. These tasks were performed relatively quickly by a group of developers who worked closely with the core K42 team but were not familiar with K42 beforehand.

## User-mode implementation

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 the user-mode services described in the last section that are as efficient as those of other operating systems. Moreover, in some cases we have found implementations that are more efficient. For example, for small files that are used by a single application, a system library object is invoked that stores the data in the file and the current position in the user address space. Thus references to update the current position or read data proceed without any system calls or context switches. Once again, this model is different from what an open-source Linux developer is used to. Our experience indicates though, that once the model is understood, it is advantageous from an open-source development perspective because it allows finer control over the module that is being modified.

## Version compatibility issues

Because K42 supports the Linux API and ABI, if a program compiles and runs on PowerPC* Linux, it should compile and run on K42. This, however, assumes that the same versions of glibc (GNU C Library), Linux, toolchain (set of development tools), and so on, are used in both cases. Dealing with multiple versions of the software that are themselves incompatible makes it difficult for K42 to generically support Linux. This problem is exacerbated when we require that collaborators build a development environment that includes the toolchain. We will, however, be moving soon to a model where the K42 kernel will be placed onto an existing Linux machine. This significantly reduces the complexity of running K42, but does not affect the impact of the wide variety of Linux versions that exist. To run on K42, the application first will have

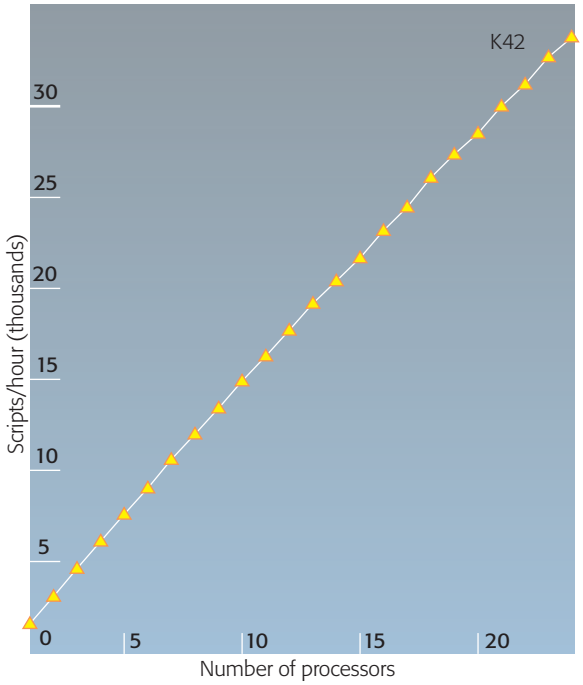


**Figure 3**
Results of running SPEC SDET on K42

to run on a particular variant or small set of variants. Though the effort just described appears prohibitive, most applications run across all versions of Linux, glibc, and so on, and thus will not pose a problem when running on K42.

In addition to causing difficulties in building and running K42, the incompatibilities between glibc and Linux-kernel data structures also add complexity and hurt performance in the emulation layer. For example, the stat(2) system call in Linux, invoked from glibc, involves conversions between Linux's struct stat passed in from the glibc layer and Linux's struct kstat used in the kernel implementation. Because K42 uses the glibc definition struct stat, the stat(2) operation involves conversions between glibc's struct stat, Linux's struct stat, and Linux's struct kstat. As discussed earlier, because most of the incompatibility issues have been dealt with, new developers will be able to concentrate on designing modules targeted to their application needs. However, even in other environments developers need to be aware of version and data-structure incompatibility issues. As with other open-source projects, incompatibilities may cause problems for developers, especially problems that cannot be reproduced when attempts are made to diagnose them.

## STATUS AND CURRENT DIRECTIONS

We have made considerable progress towards making K42 a real system capable of supporting large applications. K42 runs on PowerPC 64-bit platforms including POWER3*, POWER4, POWER4+*, Power Mac** G5, and Apple G5 Xserve hardware, and several different simulated PowerPC hardware platforms. We have run a next generation JVM** (Java Virtual Machine), the SPEC (System Performance Evaluation Corporation) SDET (Software Development Environment Throughput) suite, Apache** HTTP (Hypertext Transfer Protocol) server (from the Apache Software Foundation), mySQL database server (from mySQL Inc.), various scientific applications such as the UMT2K benchmark (from Lawrence Livermore National Laboratory), and an ASCI (Accelerated Strategic Computing Initiative) nuclear transport simulation application. We are working to be fully self-hosting and soon hope to be running our development environment on K42.

K42 is being used by several university collaborators and national laboratories and by multiple groups within IBM. We have had collaborations with the University of Toronto, Carnegie Mellon University, and more recently, the University of New South Wales that have produced significant results.[7,8,1,4,5,19,14] Additional universities and national labs have recently started to use K42, and we are continuing to encourage collaboration through the Web site and the mailing list.

Because performance continues to be one of our goals, we illustrate the results obtained related to hot swapping and scalability. Details of these experiments can be found in References 4 and 5, respectively.

*Figure 3* shows the results from an experiment based on SPEC SDET that models the behavior of a number of simultaneous UNIX users (same as the number of processors). K42 scales well through 24 processors, where its peak of 33808.1 scripts/hour is achieved, yielding an efficiency of 89.4 percent. The results demonstrate the effectiveness of K42's scaling.

*Figure 4* shows performance results when running a 1-way SDET workload with competing streaming applications running in the background. The higher

throughput line (hot swapping enabled) corresponds to an experiment in which the system monitors the pattern of the streaming-application file access and triggers the dynamic modification of the page-management policy for that file. The streaming application achieves the same performance; whereas, the interactive users in the system, as represented by SDET scripts, achieve better throughput.[4]

## Current directions

A large part of the early effort on K42 was to design for and show good scalable performance on multiprocessors. This continues to be an important direction, and we make sure that new implementations of system services are consistent with the K42 scalability goals.

In order to gather feedback from the community, we are engaging more development groups to use K42 as a prototyping platform. K42 offers developers a vehicle for testing new ideas quickly and ascertaining their value before incorporating them in Linux or other operating systems. Some of the technology originally developed for K42 has been transferred to Linux,[16,15,20] and because of the ability to easily prototype ideas, we believe this will continue. In addition, as more advanced policies are implemented in K42, we expect that its user base will grow. K42 is currently the prototyping environment for the IBM PERCS (Productive, Easy-to-Use, Reliable Computing System) project[21] and has been useful in that role. The IBM PERCS project is being done for the DARPA (Defense Advanced Research Projects Agency) HPCS (High Productivity Computing Systems) initiative.

Hot swapping, which began as an initiative to support autonomic computing, allows the operating system to swap in new objects as needed for performance, or to be upgraded while remaining available. Early work in that area demonstrated the usefulness of swapping individual objects for performance reasons. Based on that positive experience, current work is underway to extend K42's hot-swapping capability to allow the system to switch all instances of a given class, thus allowing for dynamic upgrade.[14]

To decide what instances of which objects to switch, K42 must have performance-monitoring data. However, just monitoring the kernel may not provide the best data on which to make a hot-swapping decision. We are participating in a broad continuous optimization effort with the goal of monitoring the entire system, from the hardware counters and low-level firmware and software (hypervisor), up through the operating system, compiler, runtime environment, and middleware, to the application. Code running throughout the system will examine this data and make recommendations back to the various layers, including the operating system. This feedback, for example, might be an instruction to use large pages for a given region. This work is being prototyped on K42, and if successful, will be transferred to other operating systems.

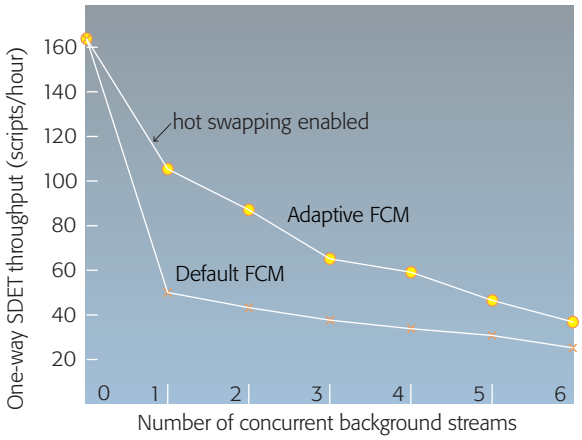


**Figure 4**
Comparative performance of an adaptive page replacement algorithm

The file system is another area of active research. KFS, which has been shown to be highly adaptable, is based on an object-oriented design, with each element in the file system being represented by a different set of objects. Developers can add new implementations to address their specific needs without affecting the performance and functional behavior of other clients. KFS also has been implemented on Linux with performance similar to ext2, while providing the journaling capabilities of ext3. Work is ongoing in KFS to explore its flexible design beyond performance gains. Our current goal is to evaluate how much the flexibility of KFS simplifies the support of object-based storage.

We are examining and optimizing significant subsystems, such as databases and JVMs. Work has begun to get these applications running on K42 and

to understand the areas in which the customizability of K42 could most help improve their performance. Developing an understanding of JVM or database optimization accomplishes two things. First, it validates the K42 model on applications considered important by the systems community, and second, any useful technology that we develop in the process can be transferred to Linux. If, for example, we prototype a novel paging policy for databases in K42 and show it has a positive impact on performance, then that experiment provides concrete motivation for including such a policy in other operating systems.

The K42 modular development model allows developers with specific needs to contribute code to K42, because such customized code does not impact other users. As a concrete example, developers at NASA (National Aeronautics and Space Administration) Ames created development patches to Linux for enhanced scalability, but this work was not incorporated in the main kernel because of possible negative impact on other users. The need for specialized changes that are not usually of interest to users of small workstations is characteristic of the high performance computing community.

One of the motivations behind our object-oriented model was to enable contributions to K42 that benefit groups of users but may not be of interest to others. Our approach allows incorporating a function that is available to those who need it without negatively impacting the other users. Although we believe this can be done, there are significant hurdles that must be overcome.

We are just beginning to develop a significant collaborator community working with our core team. Within the K42 community we need to decide who will be given the authority to commit code (add newly developed code to the existing base) directly to the K42 base. These persons will be required to maintain the code style, and more importantly, to enforce the K42 design principles. In addition, allowing anyone to commit source to K42 represents a security exposure. The current plan is to design and code-review the first several contributions and then allow collaborators who have teamed up with a member of the K42 development team to directly commit code. While this addresses to some extent the first two issues (authority to commit, maintain code style), it does not address security concerns, an issue quite familiar to the open-source community.

A more fundamental issue involves the testing and verification of the objects in the system. Unlike projects such as SPIN[22] or Vino,[23] K42 does not place restrictions on code replacement or on the programming model for that code. The policy has been to allow trusted developers to dynamically load new kernel objects written in C++. This is no longer sufficient. There is still a need to verify that the new objects match the specifications of the objects they are to replace. While formal verification models do not handle the needed generality, a technique that combines verification and an empirical harness based on specifications with iterative testing of the code shows promise.

## CONCLUSIONS

We have outlined the structure and core technology of K42 and have described our community's experience with it, particularly its ability to serve as an open-source development platform. Its modular structure makes it a valuable teaching, research, and prototyping vehicle, and we expect that the policies and implementation mechanisms studied in this framework will continue to be transferred to Linux and to other operating systems. In the longer term we believe the design principles we are studying will become important to the industry.

K42 can be used as a vehicle for rapid prototyping of ideas. For example, one could use it to determine whether a new memory management policy shows promise. If so, then there is a strong incentive for taking the time to implement it in Linux or another operating system. Because K42 is well modularized, a prototype of such a policy is reasonably easy to program and potentially to be implemented and tested by developers without requiring them to have significant kernel expertise.

Our system is available as open source software under an LGPL license. Please see our home page[6] for additional papers on K42 or to participate in this research project.

## ACKNOWLEDGMENTS

A kernel development project is a huge undertaking, and the efforts of many people made K42 possible. The following people have contributed to K42:

Andrew Baumann, Michael Britvan, Chris Colohan, Phillipe DeBacker, Khaled Elmeleegy, David Edelsohn, Hubertus Franke, Ben Gamsa, Garth Goodson, Kevin Hui, Jeremy Kerr, Craig MacDonald, Michael Peter, Jim Peterson, Eduardo Pinheiro, Rick Simpson, Livio Soares, Craig Soules, David Tam, Manu Thambi, Nathan Thomas, Gerard Tse, Timothy Vail, Amos Waterland, and Chris Yeoh. The work described here was supported by Defense Advanced Research Project Agency Contract NBCH30390004.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds, Object Management Group, Inc., Advanced Micro Devices, Inc., Apple Computer, Inc., Sun Microsystems, Inc., or Apache Software Foundation.

## CITED REFERENCES

1. J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. da Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling Autonomic System Software with Hot-Swapping," *IBM Systems Journal* **42**, No. 1, 60–76 (2003).
2. M. Auslander, D. Edelsohn, D. da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, *Memory Management in K42*, IBM Corporation (August 2002), http://www.research.ibm.com/K42/.
3. M. Auslander, D. Edelsohn, D. da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, *Scheduling in K42*, IBM Corporation (August 2002), http://www.research.ibm.com/K42/.
4. 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," *USENIX Technical Conference*, San Antonio, TX (June 9–14 2003), pp. 141–154.
5. J. Appavoo, M. Auslander, D. Edelsohn, D. da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, "Providing a Linux API on the Scalable K42 Kernel," *Proceedings of the FREENIX track: USENIX Technical Conference*, San Antonio, TX (June 9–14 2003), pp. 323–336.
6. The K42 Project, IBM Corporation, http://www.research.ibm.com/K42/.
7. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," *Symposium on Operating Systems Design and Implementation*, New Orleans, LA (February 22–25, 1999), pp. 87–100.
8. J. Appavoo, K. Hui, M. Stumm, R. Wisniewski, D. da Silva, O. Krieger, and C. Soules, "An Infrastructure for Multiprocessor Run-Time Adaptation," *Workshop on Self-Healing Systems (WOSS'02)*, Charleston, SC (November 18–19, 2002), pp. 3–8.
9. D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Operating Systems Review*, **29**, No. 5 (1995), pp. 21–27.
10. B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-Class User-Level Threads," *Operating Systems Review* **25**, No. 5, 110–121 (1991).
11. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Operating Systems Review* **25**, No. 5, 95–109 (1991).
12. J. Liedtke, "On Micro-Kernel Construction," *Operating Systems Review* **29**, No. 5, 237–250 (1995).
13. R. W. Wisniewski and B. Rosenburg, "Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems," *Proceedings of Supercomputing 2003*, (available on CD-ROM only) Phoenix, AZ (November 15–21, 2003).
14. A. Baumann, J. Appavoo, D. da Silva, O. Krieger, and R. W. Wisniewski, "Improving Operating System Availability with Dynamic Update," *Proceedings of the Workshop of Operating System and Architectural Support for the On Demand IT Infrastructure (OASIS)*, Boston, MA (October 9, 2004) pp. 21–27.
15. L. Soares, O. Krieger, and D. D. Silva, "Meta-data Snapshotting: A Simple Mechanism for File System Consistency," *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'03)*, New Orleans, LA (October 2003), pp. 41–52.
16. P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell, "Read Copy Update," *Proceedings of the Ottawa Linux Symposium (OLS)*, Ottawa, Canada (26–29 June 2002), pp. 338–367.
17. M. Greenwald and D. Cheriton, "The Synergy Between Nonblocking Synchronization and Operating System Structure," *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM, New York (1996), pp. 123–136.
18. R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi, "Performance and Environment Monitoring for Whole-System Characterization and Optimization," *Proceedings of the First IBM P=ac2 Conference*, Yorktown Heights, NY, Thomas J. Watson Research Center, IBM Corporation (October 6–8, 2004), pp. 15–24.
19. K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm, "Position Summary: Supporting Hot-Swappable Components for System Software," *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany (May 20–23, 2001), p. 170.
20. A. Baumann, J. Appavoo, D. da Silva, O. Krieger, and R. W. Wisniewski, "Improving Operating System Availability with Dynamic Update," *Proceedings of the Workshop of Operating System and Architectural Support for the On Demand IT Infrastructure (OASIS)*, Boston, MA (October 9, 2004) pp. 21–27.
21. R. Graybill, *High Productivity Computing Systems*, Information Technology Processing Office, DARPA, http://www.darpa.mil/ipto/programs/hpcs/.
22. 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," *Operating Systems Review* **29**, No. 5, 267–284 (1995).

23. M. Seltzer, Y. Endo, C. Small, and K. A. Smith, *An Introduction to the Architecture of the VINO Kernel,* TR–34–94, Harvard University (1994).

*Accepted for publication October 25, 2004.*
*Published online April 7, 2005.*

***Jonathan Appavoo***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (jappavoo@us.ibm.com).* Jonathan has a Master of Computer Science from the University of Toronto, where he is enrolled in the Ph.D. program. He is an employee at IBM Thomas J. Watson Research Center where he is working with the K42 team. His research focuses on multiprocessor operating system performance with a particular interest in scalable data structures.

***Marc Auslander***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (Marc_Auslander@us.ibm.com).* Mr. Auslander is an IBM fellow, member of the National Academy of Engineering, an ACM Fellow, and an IEEE Fellow. He received the A.B. in mathematics from Princeton University in 1963. He joined IBM in 1963 and the IBM Thomas J. Watson Research Center in 1968. He was an original member of the group that designed and built the first IBM RISC machine and first RISC optimizing compiler. He has also made contributions to AIX, other operating systems, and the PowerPC architecture. He is currently working on the K42 operating system. His research interests include multiprocessor operating systems, compilers, and processor architecture.

***Maria Butrico***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (butrico@us.ibm.com).* Maria Butrico joined IBM in 1984 and is currently a Senior Software Engineer. She received a B. S. degree in Computer Science from Pace University in 1984 and an M. S. degree in Computer Science from Columbia University in 1990. Her interests include operating system, database systems, distributed systems and middleware. She received an Outstanding Technical Achievement Award for her work on the Virtual Shared Disk.

***Dilma M. da Silva***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (dilmasilva@us.ibm.com).* Dilma M. Silva received her BS and MS degrees in Computer Science from University of Sao Paulo, Brazil, and her PhD from Georgia Tech. From 1996 to 2000 she was an assistant professor in the Department of Computer Science at University of Sao Paulo, Brazil. She is currently a Research Staff Member at IBM Thomas J. Watson Research Center. Her research interests include operating systems and dynamic system configuration.

***Orran Krieger***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (okrieg@us.ibm.com).* Dr. Krieger is a manager at IBM T.J. Watson Research Center. He received a BASc from the University of Ottawa in 1985, a MASc from the University of Toronto in 1989, and a PhD from the University of Toronto in 1994, all in Electrical and Computer Engineering. He was one of the main architects and developers of the Hurricane and Tornado operating systems at the University of Toronto, and was heavily involved in the architecture and development of the Hector and NUMAchine shared-memory multiprocessors. Currently, he is project lead on the K42 operating system project at IBM T.J. Watson Research Center, and an adjunct associate professor in computer science at CMU. His research interests include operating systems, file systems, and computer architecture.

***Mark F. Mergen***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Road, Route 134, Yorktown Heights, NY 10598 (mergen@watson.ibm.com).* Dr. Mergen has worked on and managed a variety of systems and language projects at IBM Research, including the open-source Jikes Research (Java) Virtual Machine, the research effort leading to the High-Performance Compiler for Java (HPCJ) product, the research prototype leading to the first 64-bit AIX product release, and PowerPC architecture with virtual memory software to create the innovation of database memory. He is currently working on the K42 open-source, scalable, customizable OS kernel project, with interests in hypervisor interfaces, virtualization, and clustering. He has B.S. (mathematics) and M.D. degrees from the University of Wisconsin at Madison.

***Michal Ostrowski***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (e-mail mostrows@watson.ibm.com).* Michal Ostrowski is a member of the Advanced Operating Systems (K42) group at Thomas J. Watson performing research and development activities on the K42 operating system, currently focusing on developing a framework for using Linux kernel code in K42 and developing frameworks and mechanisms for efficient asynchronous I/O interfaces. He completed his Master's degree at the University of Waterloo in 2000.

***Bryan Rosenburg***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (rosnbrg@us.ibm.com).* Dr. Rosenburg received his Ph.D. degree in computer sciences at the University of Wisconsin-Madison in 1986, and immediately thereafter joined the IBM T.J. Watson Research Center as a Research Staff Member with the operating system group of the RP3 NUMA multiprocessor project. He has been a member of the K42 team since its inception. His current research interests are in the lowest levels of operating system architecture: interrupt handling, context switching, interprocess communication, and low-level scheduling.

***Robert W. Wisniewski***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (bob@watson.ibm.com, http://www.research.ibm.com/people/b/bob/).* Dr. Wisniewski is a research scientist at Watson working with the K42 operating system team and with the continuous program optimization team. He is exploring scalable, portable, and configurable next generation operating systems and the ability of performance monitoring to automatically feedback and improve system behavior. He received his Ph.D. in 1996 from the University of Rochester on "Achieving High Performance in Parallel Applications via Kernel-Application Interaction". His research interests include scalable parallel systems, first-class system customization, and performance monitoring.

***Jimi Xenidis***
*IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134, Yorktown Heights, NY 10598 (jimix@watson.ibm.com).* Mr. Xenidis is a Software Engineer at IBM Research working with several groups in addition to the K42 Project. His research interests include Operating Systems, Linkers and Libraries, and Machine Virtualization. ■