Providing a Linux API on the Scalable K42 Kernel

Jonathan Appavoo
Computer Science Department
University of Toronto
jonathan@eecg.toronto.edu

Marc Auslander, Dilma Da Silva,
David Edelsohn, Orran Krieger,
Michal Ostrowski, Bryan Rosenburg,
Robert W. Wisniewski, Jimi Xenidis
IBM T. J. Watson Research Center
k42@watson.ibm.com
http://www.research.ibm.com/K42

Abstract

K42 is an open-source research kernel targeted for 64-bit cache-coherent multiprocessor systems. It was designed to scale up to multiprocessor systems containing hundreds or thousands of processors and to scale down to perform well on 2- to 4-way multiprocessors. K42's goal was to re-design the core of an operating system, but not an entire application environment. We wanted to use a commonly available interface with a large established code base. Because Linux is open source and widely available, we chose to support its application environment by supporting the Linux API and ABI. There were some interesting complications as well as advantages that arose from K42's structure because our implementation of the Linux application environment was done primarily in user space, had to interface with K42's object-oriented technology, and used fine-grained locking. Other research systems efforts directed at achieving a high degree of scalability and maintainability exhibit similar structural characteristics.

In this paper we present the motivation behind K42, including its goals and overall structure, and describe its system interface. We then focus on the required infrastructure and mechanisms needed to efficiently support a Linux application environment. We examine the lessons learned of what was advantageous and what was disadvantageous from K42 in implementing the Linux API and ABI.

1 Introduction

The K42 project[6] is developing a new open-source operating system kernel incorporating innovative mechanisms and policies and modern programming technologies. Our goal is to start from a "clean slate" and examine the system structure needed to achieve excellent performance in a scalable, maintainable, and extensible system. Although we wanted to design from scratch, we could not, nor did we want to, implement all aspects of an operating system from scratch. Further, requiring applications to use a new API would make experimenting with the system unpalatable to potential users. We therefore did not introduce a new personality, but instead made K42 Linux API- and ABI-compatible. This paper examines what we needed to do to achieve this compatibility.

K42 has a set of design features that made implementing this compatibility an interesting task. Specifically, in keeping with the design strategy of K42, the API was implemented mostly in user space, had to interface with K42 object-oriented technology, could not use any global locks, and could not hold a lock across multiple object calls. As in many areas of system design, these features both simplified and complicated the implementation. Other scalable and maintainable systems exhibit similar characteristics. Throughout the paper we point out experiences where K42's features impacted (both positively and negatively) the implementation of the API.

To make this approach of supporting the Linux application environment effective, K42 needs to fully support the Linux API and ABI. There is no porting to K42. Any application that runs on Linux just runs if using K42's ABI support and just needs to be re-compiled to use K42's API support. Most applications, including significant benchmarks, run without recompilation. If the application does not run, we fix K42 until it does. We will describe K42 in more detail later, but currently K42 can run significant Linux applications. For example, we have run the SPEC SDET[2] benchmark suite, an Apache web server, and the full ASCI Nuclear Transport Code[21].

Our model for mapping a Linux environment onto K42 is illustrated in Figure 1. A Linux Application Environment, as defined by any of the popular distributions, consists of several layers and modes of operation and execution. As with most Unix-like operating systems, Linux can be segmented into a user level and a kernel level. The user level is the portion of Linux that interacts directly with the user processes, employing the services offered by the kernel level. The kernel level presents the

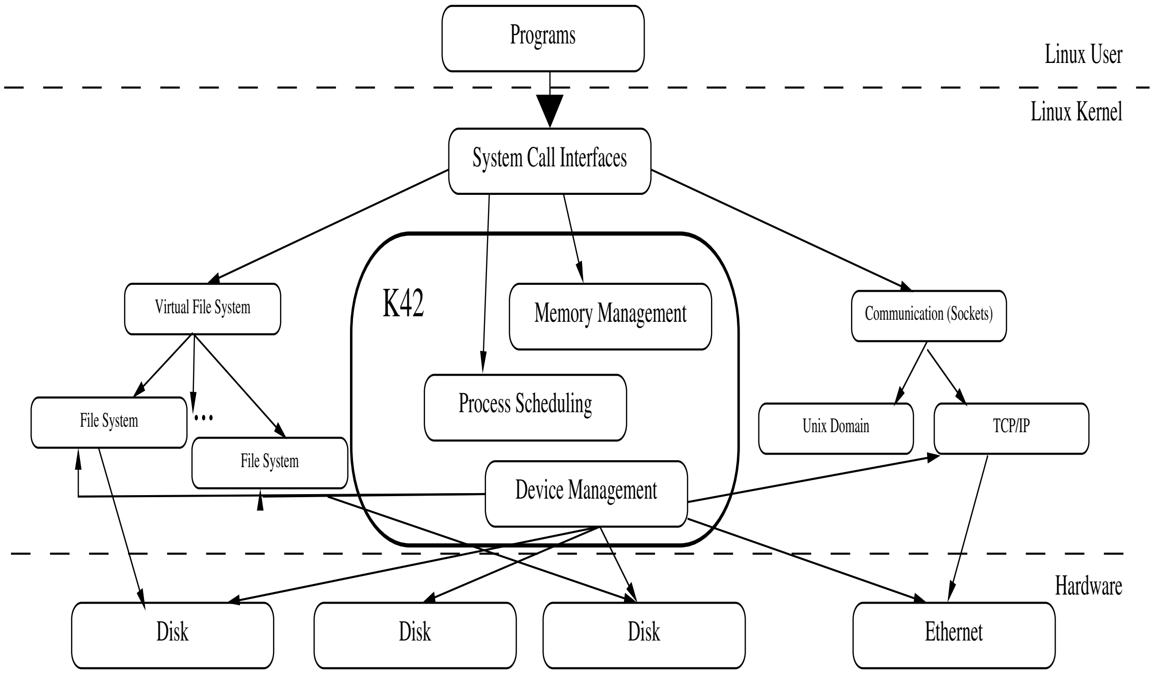

Figure 1: Mapping a Linux environment on K42

image of a single system to each user as well as each application run by that user. It is responsible for managing all resources and securely sharing them.

In order to provide a complete Linux application environment, K42 must provide a set of interfaces that deliver all system services. These come in the form of library functions, system calls, user commands, special or device files, and file formats. Though some of these services do not require kernel support, many of them expose functionality exported by the kernel.

The focus of this paper is on how we provide a Linux application environment on top of K42, a kernel designed for scalability and extensibility. As noted, there are many services that need to be provided in order to present the user with a full application environment. In this paper we examine the major categories of these services and describe how we implement them. In particular, we describe how we emulate the Linux process tree; emulate `fork`, `exec`, and `clone`; support glibc and translate its system calls, provide file systems and sockets, and implement dynamic linking via ld.so.

In addition to supporting a full user environment, we wanted to be able to use the existing Linux-kernel code base to provide the desired range of hardware drivers in K42. Although this paper focuses on the user application environment, we briefly outline our kernel strategy here (more details can be found in Auslander et. al.[8]). We use the Linux code base for hardware driver support, for networking and file-system code, and to provide a stable inter-operable environment. To be able to use Linux device-driver, networking, and file-system code we needed to provide a Linux-kernel environment (or Linux emulation environment, as in Goel and Duchamp[14]). To do so, K42 presents itself as a target hardware architecture for Linux (in the same way as real hardware architectures such as Alpha, i386, and PowerPC do). This requires implementing the basic functionality required of architecture-specific code in Linux (e.g., assembly-level constructs, definition of locking mechanisms) in an "emulation layer" that can be linked with the individual Linux-kernel components to be used in K42. For example, device-driver code uses locks; these locks need to be mapped onto K42 locks and thus the Linux device-driver code needs to be compiled to run in the K42 environment[8]. Further, K42 emulates the Linux-kernel services (e.g., kernel memory allocation) needed to support these components. As seen in Figure 1, this means that K42 replaces the core memory management, process management, and device management code of Linux, but uses file systems, device driver, and library (e.g., glibc) code from Linux.

The rest of this paper is organized as follows. Section 2 starts by introducing K42, describing its motivation and structure, and presenting its system interface. Although much of the API implementation is interrelated, we divided it into mechanisms external to the process, presented in Section 3, and internal to the process, presented in Section 4. Throughout these sections we present the advantages gained from K42's infrastructure as well as the complications that arose. In Section 5 we

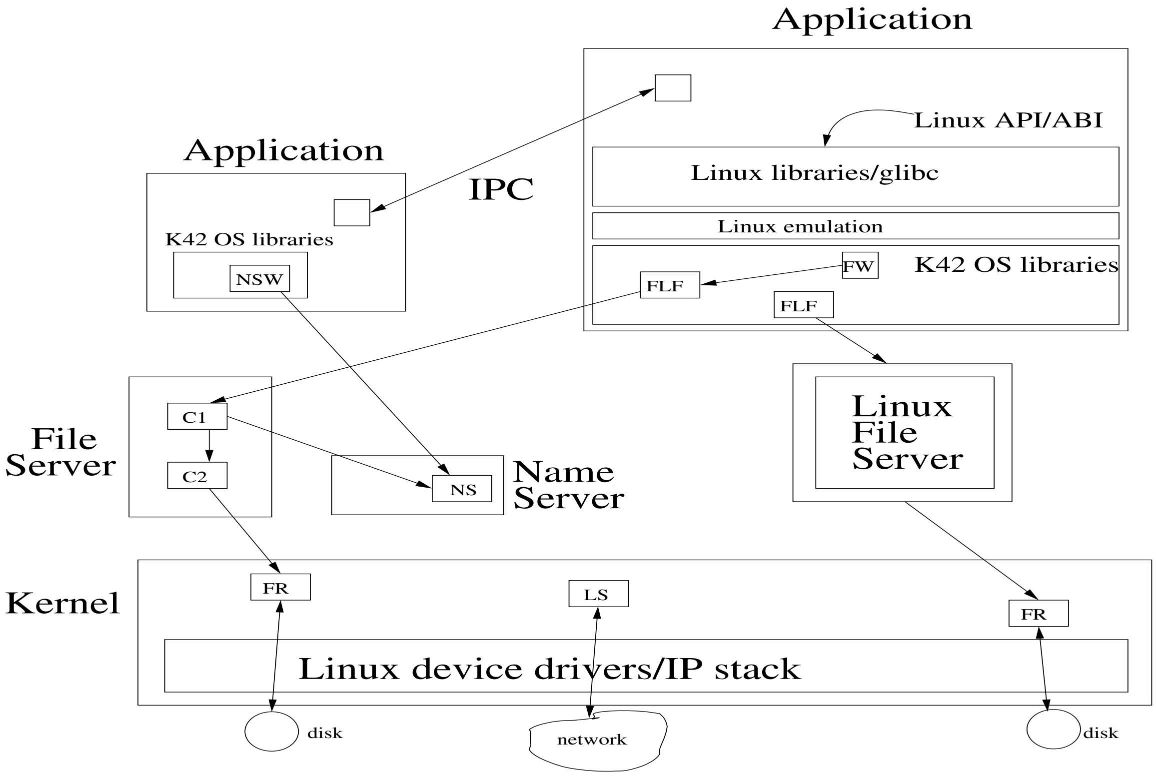

Figure 2: Structural Overview of K42: FR - File Representative Object, LS - Linux Socket Object, C1 C2 - File-system Component 1 2, NS - Name Server Object, NSW - Name Server Wrapper Object, FLF - File Linux File Object

describe the status of K42, provide a performance evaluation of K42 and Linux running the SPEC SDET benchmark, and present work related to K42. Concluding remarks are presented in Section 6.

2 K42

In this section we provide background and motivation for K42 and describe its structure and system interface. One of the primary differences between K42's Linux API and Linux is that in K42 much of the standard kernel functionality has been implemented in user space. Results from the Exokernel project[13] demonstrated that user-level implementation of operating system services can lead to significant performance gains. We have found that there are also some performance challenges (more details later, e.g., Section 3.2), especially when implementing a pre-determined model, e.g., `fork`. The Exokernel work showed that moving code into the application's own address space improves performance. Micro-kernel designs did not show performance improvements. K42 is more like the Exokernel.

Although performance was our primary motivation for user-level implementation of kernel services, there are other benefits. Code and data for services implemented in user space do not tie up kernel resources. User-space code can yield a cleaner programming model because issues such as stack violations are easier to protect against. In user space, allocating virtual stacks, versus pinned stacks in the kernel, allows quasi-infinite stacks with red pages guarding the end. Moving code into the application's own address space does not introduce protection issues. The application still only has access to resources that the prior Linux permissions would have allowed. Moving code into user-level servers introduces a potential scheduling challenge that we address by running the servers at a higher priority than applications.

K42 has addressed security in a first-class manner. All IPC calls between applications, servers, and the kernel contain a badge that securely identifies the caller and is guaranteed by the kernel. For each object invocation on the callee side there is a series of matched rights the badge is compared with to ensure the caller has the proper authentication to make the call. The infrastructure in K42 allows for different security models between different servers and different applications as desired.

In K42, all thread scheduling is done by a user-level scheduler and requests that would normally block in the kernel, e.g., page fault waiting for disk I/O, do not block in the kernel but are instead returned, with control, to the user-level scheduler. The user-level scheduler can then block the thread and continue running another thread in the same address space. This positively affects the handling of signals, asynchronous I/O, sockets, and other aspects of providing the API.

Other salient aspects of K42's design impacting the implementation of a Linux API are the pervasive use of object-oriented technology and our locking strategy of both avoiding global locks and of not holding locks while accessing multiple objects. The former locking strategy implies a pervasive methodology for fine-grained locking (i.e., no global kernel lock as in Linux). The latter implies that locks may not be held for the entire duration of performing a task, such as a fork-chain collapse (reducing the length of the tree representing forked processes), implying that in-flight requests must be algorithmically accounted for, i.e., if locks are not held across object invocations then multiple requests may occur simultaneously and need to be accounted for.

In the rest of this section we describe K42's goals and motivations, its overall structure, and the key technologies used in its design and implementation.

2.1 K42 motivation and goals

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. Supporting the Linux API and ABI makes K42 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. K42 is available under an LGPL license (see http://www.research.ibm.com/K42).

The system is fully functional for 64-bit applications and currently runs on PowerPC (SMP) platforms (hardware and simulator) and is being ported to x86-64. It runs codes ranging from scientific applications, such as the Splash Benchmark Suite[23] and a full ASCI Nuclear Transport Code[21], to complex benchmarks like SPEC SDET[2] to significant subsystems like Apache.

Providing a well-structured kernel is a primary goal of the K42 project, but performance is also a central concern. Some research operating system projects have taken particular philosophies and have followed them rigorously to extremes in order to fully examine their implications. Although we follow a set of design philosophies in K42, we are willing to make compromises for the sake of performance. The principles that guide our design include 1) structuring the system using modular, object-oriented code, 2) designing the system to scale to very large shared-memory multiprocessors, 3) leveraging performance advantages of 64-bit processors, 4) avoiding centralized code paths, global data structures, and global locks, 5) moving system functionality to application libraries, and 6) moving system functionality from the kernel to user-level server processes.

Goals of the K42 project include:

- **Performance:** A) Scale up to run well on large multiprocessors and support large-scale applications efficiently. B) Scale down to run well on small multiprocessors. C) Support small-scale applications as efficiently on large multiprocessors as on small multiprocessors.
- **Customizability:** A) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. B) Autonomically have 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 to high-end enterprise servers.
- **Wide availability:** A) Be available to a large open-source and research community. B) Make it easy to add specialized components for experimenting with policies and implementation strategies. C) Open up for experimentation parts of the system that are traditionally accessible only to experts.

2.2 K42 structure

K42 is structured around a client-server model (see Figure 2). The kernel is one of the core servers, currently providing memory management, process management, inter-process communication (IPC) infrastructure, base scheduling, networking, device support, etc. (In the future we plan to move networking and device support into user-mode servers).

Above the kernel are applications and system servers, including the NFS file server, name space 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 IPC is between objects in the client and server address spaces. We use a *stub compiler* with decorations (additional keywords) on the C++ class declarations to automatically generate IPC calls from a client to a server. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls, including specific identification of the client being granted access. We have optimized the IPC path as described in Gamsa et. al. [12] and obtained good performance via efficient IPCs similar to L4[20].

From an application's perspective, K42 supports the Linux API and 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 directing them to their K42 implementation, as described in Section 4.1. Although Linux is the first and currently only personality we support, the base facilities of K42 were designed to be personality-independent. As mentioned in the introduction K42 also supports a Linux-kernel *internal personality* allowing us to use the large code base of drivers, networking, and file-system code.

2.3 K42 key technologies

To achieve the above mentioned goals, we have incorporated many technologies into K42. We have written several white papers (available on our web site) describing these technologies in greater detail. This section provides an overview of the key technologies used in K42. At the beginning of this section we highlighted the ones impacting our Linux API implementation.

- Object-oriented technology has been applied to the entire system. This has been used to achieve good performance through customization, to achieve good MP performance by increasing locality, to increase maintainability by isolating modifications, and to perform autonomic functions by allowing components to be hot swapped[24].
- Much traditional kernel functionality is implemented in libraries in the application's own address space, providing a large degree of customizability and reducing overhead by avoiding crossing address space boundaries to invoke system services.
- A structure that permits machine specific features such as the PowerPC inverted page table and the MIPS software-controlled TLB to be exploited in an isolated manner without compromising portability.
- System functionality implemented in user-level servers with good performance maintained via efficient IPCs similar to L4[20].
- The use of *processor-specific* memory (the same virtual address on different processors maps to different physical addresses) to achieve good scalable NUMA performance. This technology, combined with avoiding global data, global code paths, and global locks, allows K42's design to scale to thousands of processors.
- A K42 specific object-oriented structure, *clustered objects*[12], which provide an infrastructure to implement scalable services with the degree of distribution transparent to the client. This also facilitates autonomic multiprocessor computing[4] as K42 can dynamically swap between uniprocessor and multiprocessor clustered objects.
- K42 is designed to run on 64-bit architectures and we have taken advantage of 64 bits to make performance gains by, for example, using large virtually sparse arrays rather than hash tables.
- K42 is fully preemptable and most of the kernel data is pageable.
- K42 is designed to support a simultaneous mix of time-shared, real-time, and fine-grained gang-scheduled applications.
- K42 has developed deferred object deletion [12] similar to RCU [22] allowing objects to release their locks before calling other objects. This efficient programming model is crucial for multiprocessor performance and is similar to type-safe memory [16].

3 Linux process external environment

The next two sections describe K42's implementation of the Linux API. Much of the code and therefore description is interrelated. This section presents the infrastructure and code mostly external to a Linux process.

3.1 The Linux process tree

In order to implement a Linux environment the Linux process tree must be supported. In K42 this functionality is provided by the ProcessLinuxServer. This server was implemented primarily by using Linux data structures, such as the `task struct`, with only minor changes to track Linux processes in K42.

Linux processes are backed by native K42 processes and contain a pointer to the underlying K42 process. In addition to backing Linux processes, K42 processes are used to implement core facilities. Linux processes are used for user applications and many of K42's servers, such as the filesystem. Should additional personalities be added to K42, they would add additional types of processes.

The ProcessLinuxServer keeps track of the relationship between different Linux processes. It tracks the parent-child relationship when a process is created via `fork` (we only intend to support forking of Linux processes). When a parent process finishes or is terminated, the ProcessLinuxServer re-assigns the child to `init`. In addition to the process tree, the ProcessLinuxServer keeps track of two other trees or sets. It tracks the sessions used to associate processes that are related by their starting tty. It allows any process to get access to the session tty even though the parent has not passed the child an open fd for it. In addition to sessions, the ProcessLinuxServer keeps track of the set of process groups. As in Linux, every process knows what process group and what session it belongs to.

In addition to maintaining the three structures mentioned above, the ProcessLinuxServer manages Unix credentials, and as in Linux, matches process to process groups where appropriate. The ProcessLinuxServer is also responsible for the delivery of signals (discussed in more detail later). The Linux process tree, combined with session and process groups, forms the core tracking infrastructure for many of the services described in the rest of the paper.

3.2 `Fork`

Unlike many implementations of Unix, K42 executes much of the fork code in the client, both in the parent and in the forked child, and does not grab a global tree-lock thereby improving concurrency. Concretely, the parent sets up the memory state of the child. It has a record of all the memory mappings (regions) associated with itself, much the same as the Linux kernel keeps track of all regions associated with a process. After setting up the memory of the child, it passes the rest of the state to the child. In Linux, execution normally starts in the child at the instruction of the `fork` system call. However, in K42, a significant amount of code is executed in the child before branching to the instruction following the `fork`. This code is similar to what would have been executed in the kernel, such as setting up the file descriptor table, etc.

The K42 model of removing this code from the kernel has several advantages. It reduces the amount of code in the kernel. Any errors that occur in it do not crash the system, but only bring down the process involved in the fork. Perhaps most importantly, it yields an easier programming model as the code does not have to be written assuming fixed sized stacks or written to use a limited amount of pinned memory.

Although implementing the code in user space simplifies programming, the K42 requirement of not holding a global lock throughout such an operation requires careful coding to avoid potential race conditions. Another performance advantage, but programming difficulty, is that when performing a fork-chain collapse (reducing the length of the tree representing forked processes), we do not hold a lock across all the operations but rather lock each individual node independently. Thus, the algorithm must allow for the possibility of multiple in-flight requests (remove and insert). This is discussed in greater detail below.

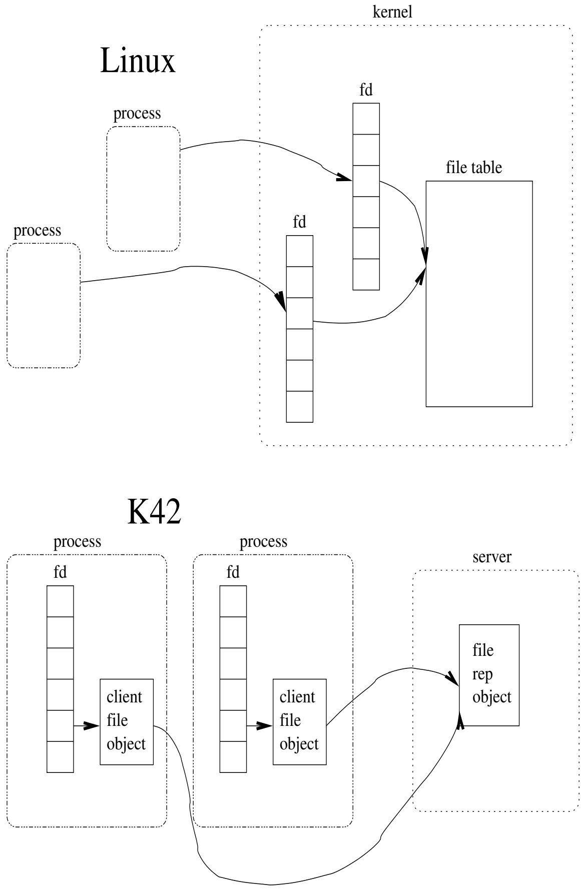

Figure 3: The fd table in Linux and equivalent structure in K42

The object-oriented nature of K42 has serious implications when replicating structures such as the file descriptor (fd) table. In Linux, a fd is a pointer into a table (see Figure 3) with associated file reference counts that are incremented when a `fork` occurs. However, in Linux, the fd table takes kernel memory for each process because it is stored in the kernel's data structure for the process image. In K42, the fd array, which is maintained in user space, points to a series of client file object instances that are in each process's address space (see Figure 3, more information on file descriptor manipulation in K42 is presented in Section 4.5). There is a little more memory that must be replicated on a `fork` for the fd table in K42.

Although K42's approach avoids explicit replication of the fd array, it introduces post-`fork` authentication issues. Each of the object instances contains authentication information providing a particular client the right to access a given file. The authentication information contains a PID and thus, although valid for the parent, does not provide the child with permission to access that same file. A call on each file object must be made for the child to obtain permission. This could be an expensive operation if the parent had a large number of open files. It is especially unfortunate because it is unlikely the child will access many of those files as its mostly likely action is to `exec`. We avoid this potential performance bottleneck by lazily granting access to the child on first access to the file server.

As is frequently the case in programming large systems, the tradeoffs are not clear, and although some aspects of K42's implementation of `fork` simplified programming or improved performance, other aspects made it more complicated or hurt performance. For K42 native processes (that do not support `fork`) the model of performing much of the process creation code in user space has been a win. Unfortunately, `fork` has difficult performance issues. The overhead of implementing `fork` and `exec` in user level comes from (1) the cost of passing state from parent to child, rather than just copying in kernel, (2) the extra fault overhead to map in data structures the parent passed to child copy-on-write, (3) the cost of initializing (on `exec`) and re-initialing (on `fork`) the system function implemented in user level (e.g., the fd array, memory allocation, etc), and (4) the overhead from our end-to-end authentication scheme (for each file being passed to the child, the server needs to be involved to provide access). Although we bought back most of the fork performance lost due to these issues by lazy (re)initialization, there is still some performance loss and work is underway to reclaim it. Fortunately the performance of `fork` for most server applications (some benchmarks notwithstanding) is not critical.

3.3 Reproducing the memory image on a `fork`

After a `fork`, the memory image of the parent must be reproduced in the child. Any private mappings for a file cause a snapshot of that file at the time of the `fork` to be produced in the child. Any shared mappings cause a shared mapping to be created in the child with modifications made by the parent to be replicated in the child, with the possible caveat of files mapped in read-only but modified by the debugger through the use of ptrace.

Traditionally the operation of reproducing these mappings is done in the kernel. However, in K42, before starting the child, the parent reproduces these mappings with the help of a few underlying K42 kernel services. K42 provides the ability to create a new non-executing process, the ability to make memory mappings in it, and the ability for a region and associated objects to be fork copied. In K42, a contiguous piece of memory is represented by a region, which in turn is backed by a File Cache Manager (FCM) that caches the in-core pages,

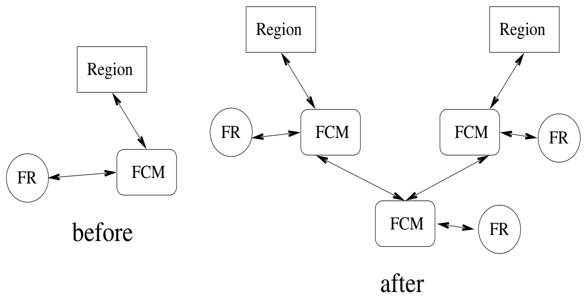

Figure 4: Example of Region, FCM, and FR objects after a fork copy

and a File Representative (FR) that maps all the pages associated with a file. Figure 4 illustrates a standard fork copy. Prior to the fork (*before*), the three kernel structures (FR, FCM, Region) represent a contiguous portion of the memory in the parent. After the fork (*after*), the upper-left three kernel structures still represent the portion of memory in the parent, and the upper-right three structures represent the same portion of memory in the child. All the frames backing the region are immediately transferred to an internal FCM (the root FCM in this figure) and lazily copied back to the parent and child as faults occur.

As additional forks occur, a binary fork-tree is produced. When nodes on the tree are removed because the process for which they are representing memory finishes or is terminated, it is important to "collapse" the tree, otherwise inefficiencies result because page-fault processing needs to traverse the unnecessarily long chain of nodes. The standard way to address this is to acquire a lock and walk the tree performing the collapse. Because K42 is programmed with independent object instances representing each of the regions, and with a programming model not allowing the acquisition of a single lock across multiple object instances, we acquire and release a fine-grain lock for each node in the tree. The FCM objects therefore need to be able to handle in-flight page fault requests occurring during the collapse operation. Although this does make the programming more complicated, fine-grain locking schemes provide better scalability.

The collapse algorithm begins when a node in the tree needs to be removed. That node removes itself and it tells its sibling to combine with their parent. If the sibling is not a leaf node then all frames are transferred to the parent and the sibling node removes itself. If the sibling is a leaf node then no action is taken. To prevent a chain from forming in this case, on the next fork copy, the new child is made a child of the existing parent (no new parent is created). For this algorithm to work, frames are not copied to an internal child node (except in one case for optimization purpose only), instead they are

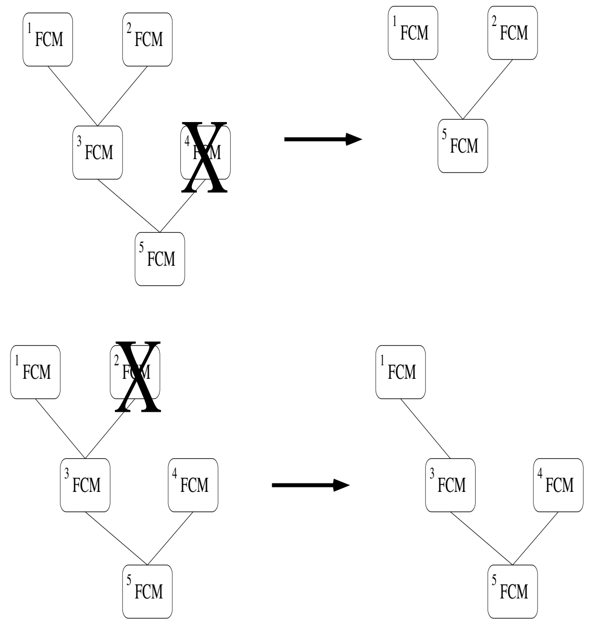

Figure 5: Standard scenarios in fork-chain collapse

copied to the leaf node requesting the page. As examples, two standard scenarios are illustrated in Figure 5. In the first, node 4 is being deleted. This results in a collapse with node 5 becoming the parent of nodes 1 and 2. In the second, node 2 is being deleted. As above, because it is a leaf node, its removal does not induce a collapse.

In the normal `fork`/`exec` case we do not perform any collapse. The algorithm is in place to allow K42 to continue to perform acceptably in scenarios that would generate long fork-tree chains. This algorithm provides efficient collapsing of fork trees and does not require a global lock.

3.4 Reproducing the fd and signal state on a `fork`

In addition to reproducing the memory in the forked child, the filesystem state and signal state must also be copied. Most of this state is replicated during the reproduction of the memory state. A mechanism is needed to indicate the child is now using the files referenced by the open fds and to receive the signals set up by the handlers.

In Unix, after a `fork`, the kernel indicates the child is using the files by traversing the fd list and incrementing the usage count on each of the files pointed to by that fd list. As mentioned above, in K42 there is an individual object instance that manages each file. User-space FileLinuxFile objects have permission to talk to the underlying file through the objectRef they hold to that file. The vast majority of calls to `fork` are followed by calls to `exec` in the forked child. In K42, we lazily provide permission to each of the files in the child. In the common case of a succeeding `exec`, we therefore avoid the work altogether. In Unix, this would be equivalent to not actually updating the reference count until an access is made (but also being careful to not decrement the count if no reference is made). In K42 we accomplish this by creating an object in the kernel to represent the client file being reproduced in the child. In a `fork` intensive workload, where many files are not accessed after fork, we pay the same overhead as a kernel implementation like Linux (because the parent pushes this information to the kernel), but, when files start being used, the K42 kernel interacts with the server to perform a proper setup (the child lazily initializes the structures), and from then on we get the advantages of our user-level implementation.

The signal state of the parent is copied to the child entirely by the memory reproduction. Thus, all signal handlers installed in the parent are reproduced in the child. The disposition of pending signals due to, for example asynchronous I/O, is undefined. In K42 the signal will be delivered only to the old object that was originally waiting for it.

3.5 Signals

Signal delivery in K42's Linux application environment differs from most implementations in that most operations can be carried out without kernel involvement. State information such as signal masks, `sigaction` specifications (ignore, default action, or handler to call when signal arrives), and set of pending signals, is kept in the application's address space. Operations such as `sigsuspend` and `pause` simply block in the client without any interaction with the kernel.

Inter-process signal delivery goes through the ProcessLinuxServer. The target K42 process is identified, and in the general case, the server delivers the signal to the client through an IPC. For signals that can not be ignored (`SIGKILL`, `SIGSTOP`), the server contacts the kernel.

Intra-process signal delivery is carried out entirely in the client space. This results in significant performance advantages for the current implementation of Linux Threads[19], which uses signals to implement synchronization among threads. In the upcoming Linux threads package(NPTL)[10], synchronization is implemented using the *futex* (fast light-weight user-space semaphores) kernel synchronization service. Although futexes are more lightweight than the current implementation, they still involve kernel interaction. With the new threading model, we still expect to be able to implement the synchronization at user level.

4 Linux process internal environment

4.1 Linux system calls and glibc

The largest component of the Linux API/ABI that must be supported is the set of system call interfaces. For true ABI compatibility this must be supported by performing system call trap reflections from the code making the Linux system call to the implementation of the system call. In K42, the functions that implement system calls reside within `exec.so`, a pre-loaded library within the process described in Section 4.2.

Support for the trap reflection mechanism is intended for strict compatibility. We have designed but not yet implemented this mechanism. In general, we expect applications running on K42 to use a version of the GNU C Library (glibc) targeted for K42 to obtain access to the Linux system call interface without using trap reflection. Most applications gain access to Linux system call interfaces via glibc wrapper functions that present a C interface thereby hiding the architecture specific details of making a system call. Glibc provides a mechanism that allows different architecture targets to define how these wrapper functions work. This is accomplished by providing the assembly-level mechanisms for making system calls. Except for this low-level assembly system call interface level, K42 reuses all of glibc. We have developed a glibc targeted for K42 that efficiently accesses the implementations of system calls within a process. We use the system call number as an index into a *system-call transfer-table*. This table is at a well-known location and contains the address of the function that implements a particular system call. In effect, we short-circuit the normal trap-reflection and simply jump to the correct function.

The system-call transfer-table permits us to handle Linux system calls within an application, but at the same time avoids the need for us to explicitly link applications against K42 code. As a result, dynamically linked PowerPC64 Linux binaries that use glibc wrappers to make system calls run without modification. Applications linked against a static non-K42 glibc still require trap reflection.

As previously noted, in K42 we have moved kernel functionality into the application's address space. This includes some of the processing for system calls. Moving this code into the application's address space does not sacrifice security. The portion of the system call that is handled in the user's address space is the part that is allowed by the Linux credentials the user had. The model enforces calling privileged servers to perform requests where the user credentials are insufficient. Under either model, users can cause incorrect behavior (e.g., if an application has read/write access to a file one thread may remove the file another thread is attempting to write). This model does introduce additional places where the user may ”shoot themselves in the foot”, but it does not sacrifice security.

When a thread makes a Linux system call, a wrapper object in our Linux emulation layer is invoked. At this point, the thread is marked as being in a system call. This code is re-entrant allowing additional calls from within the process. If a signal is delivered to a thread in this state, it is deferred until the return from the outermost level of system call. The thread will not block when it is in this state, but instead will return out of the system call layer and then block[7].

4.2 Loading

K42 provides a library called exec.so. This library provides the OS functionality that K42 expects to be implemented within a process. It also provides the system-call transfer-table and the functions it points to. Exec.so maintains data such as stateful I/O interfaces that implement select and poll, and must be initialized before any Linux code executes. This early initialization needs to occur so that we can provide support for system calls made by the dynamic linker and glibc.

A K42 process can be created using native K42 facilities (not `fork`/`exec`). This creation occurs by constructing the objects that represent that new process in the parent. During this construction, the parent loads exec.so into this new or *child* process. It also loads information containing what executable that child process should run. The new child process starts running in exec.so and loads the appropriate Linux executable, and if necessary, the dynamic linker. As part of its initialization, exec.so creates and initializes the system-call transfer-table, and then jumps to the appropriate entry point of the Linux application. The Linux application is now able to make system calls that are handled by the code in exec.so via the system-call transfer-table, or by trap-reflection, which also uses the table. This mechanism is established without any symbol dependencies between the Linux application and exec.so.

We must also provide a mechanism to allow applications to access K42 library interfaces directly. However, we cannot simply allow the dynamic linker to load a new image of exec.so because the existing K42 code in exec.so contains important state information. Instead, the library image that the dynamic linker loads contains specially modified ELF headers that direct the dynamic linker to look for K42 library code, data, and symbol information in the location where exec.so has been already loaded. In a 64-bit address space it is easy to always load exec.so at the same location allowing us to avoid the need to relocate exec.so at run-time.

4.3 Exec

The exec system call conceptually creates a new process based on a set of predefined specifications. The memory image of the new process is clean, files not marked close on exec are maintained open in the new process, and signals not set to ignore are reset to default.

Like other services in K42, exec is implemented primarily in user space. Given the above model for loading, a K42 process may be considered an exec.so image that loads or overlays Linux executables when asked to exec. With this model, the same K42 process containing exec.so, can, when asked to perform an exec, unload the Linux executable and any shared libraries it has loaded, re-initialize the appropriate aspects of the Linux personality, and load the new Linux executable. In the common case, exec can be performed entirely in user space.

It is not possible to use the overlay model if the exec is for a setuid program (or for clone or native K42 processes). Because the old program could have stored permissions not granted to the new process anywhere in its memory image, we have a privileged server create a clean process, and copy the necessary state. This server uses the underlying K42 mechanisms used by fork.

The overlay strategy for implementing exec permits us to efficiently handle most Linux applications and perform the exec in user space. For the cases where we can not use the overlay strategy, most operations are still performed in user space.

4.4 Clone

The clone system call creates a new process like fork does, but it offers more control of which elements in the execution context (memory space, table of file descriptors, table of signal handlers, file system information, process id, etc) are intended to be shared between the child and parent processes. The main use of clone is to implement LinuxThreads[19], which are multiple threads of control in a program that run concurrently in a shared memory space.

The implementation of clone is carried out completely in the application address space, i.e, no interactions with the kernel or servers are necessary. clone builds on K42's user-level thread model[7]. Even though the kernel does not allocate these thread PIDs we still need a mechanism to guarantee unique PIDs across the system. We do this by reserving a bit range in the PID space for cline PIDs. PIDs are 32 bit in Linux. K42 uses 20 bits to identify the target K42 process and 12 bits to identify the clone within the process. This allows clone PIDs to be assigned locally by each process and still ensure that clone PIDs are globally unique.

4.5 Files, file descriptors, and name space

Usual implementations of UNIX APIs store client (application) specific state information regarding IO/IPC (files, sockets, pipes, etc) elements in the kernel. For example, for files, the kernel not only manages attributes such as ownership and file length, but also is responsible for keeping state information (e.g. file position) for each client. File descriptors *fds* (offsets in a per-process table kept by the kernel) are provided as an argument by the clients to identify the target of the operation. Most operations on such elements may need to block. If so, the blocking is also carried out in the kernel.

K42 takes a different approach. The client-specific information for open files, sockets, and pipes resides in the application's address space. K42 stores information for stateful interfaces in the object implementing that interface, which is placed in the application's address space. When blocking is necessary, the thread will block until notified by the server that the operation can proceed. As a consequence, kernel resources (kernel thread stack, process state, register state, etc.) are not tied up. This avoids the difficulties encountered in *m* on *n* scheduling models. In some scenarios, the operations can execute entirely in the application space, thereby achieving significant performance gains.

Operations such as socket(), pipe(), creat(), and open() create a new IO/IPC element, returning a file descriptor for it. K42's implementation of these system calls does the following: (1) the client contacts the name-space server (or MountPointServer) to discover the appropriate server for that resource, (2) the client initiates an interaction with the appropriate server. The server identifies the server object representing the resource (creating one, if necessary), checks credentials, and returns to the client a handle for the server object. This handle includes capabilities indicating that the client is allowed to invoke the server object directly, and (3) the client creates an object to represent the client side of the open file, socket, or pipe, storing in it the object handle to the server object. A file descriptor is associated with the newly created client object, and this mapping is stored in a file descriptor array kept by the application.

Subsequent operations on the file descriptor will be delegated to the client object. The client object implementation usually invokes the corresponding operation on the server object representing the resource, and updates its own local information. In some cases, the client object is able to carry out the operation completely. For example, for a small file with a single client, file position, file data, and all stat information (including file length), can be managed in the client. The information is propagated to the server when the file is closed, becomes too large to be reasonably cached in the client, or is accessed by additional clients.

K42's synchronous I/O model is similar to other asynchronous models in the sense that threads do not block in the kernel or servers. To illustrate how blocking and unblocking occurs in the application space even for synchronous requests, we describe our implementation of sockets. All socket interfaces in K42 are similar to Unix ones, except there is an extra parameter to pass back to the client information about the state of the object (for example, whether it is readable, writable, or if there is any exceptional condition on it). The client uses this state information to decide if it should block. The server makes an asynchronous upcall to notify the client of state changes (e.g. data becomes available). This scheme allows us to implement `select` and or `poll` purely in user space. Also, asynchronous notifications are piggybacked on synchronous responses to client invocations. Our implementation is also able to detect when a client is ignoring notifications (for example, a forked child that inherited the file descriptor but is not interested in the socket), and to stop sending them.

Many interfaces such as signals, select and poll, and cursor management in files, require synchronization. Linux provides this synchronization in the kernel. K42 either provides this in the application space (if that application is the only user of the resource), or in the server object managing that resource. In fact, we have an interesting example of the hot-swapping mechanism[24] that is used to switch between these two implementations when the requests for a given resource change from coming from a single application to originating from multiple applications.

K42's support for namespace traversal involved in pathname-based operations (similar to Welch and Ousterhout[25]) has performance and scalability advantages over the usual pathname lookup schemes due to its fine-grained locking and ability to resolve in the application address space the parts of the pathname that identify mount points. A MountPointServer stores the association between parts of the name space and specific file-system servers. The information available in this server is cached in the application's address space during its initialization phase. The MountPointServer publishes the version number for its up-to-date information. An application can use these version numbers to check efficiently if it has out-of-date information.

In the uncommon case that the information is not up-to-date, the application requests the current information from the MountPointServer. Once the mounting information is used to resolve part of the pathname and identify the corresponding file-system server, the client contacts the file-system server and passes arguments of the operation to be performed and the unresolved part of the pathname.

The file-system independent layer in K42 implements caching of directory and file entries recently resolved. Each file-system instance has its own caching data structure. Fine-grained locking is used when manipulating this data structure, avoiding well-known scalability bottlenecks in name resolution.

5 Status, Performance, and Related Work

In this section we describe the status of K42, describe work related to K42, and finish with a performance evaluation of K42 and Linux running the SPEC SDET benchmark[2].

K42 is available under an LGPL license and a CVS (Concurrent Version System) source tree is available. Directions on how to obtain it at are available at http://www.research.ibm.com/K42. An early version of a complete environment including build infrastructure, debug tools, a simulator, and source is available as of March 2003.

The modular structure of the system makes it a good teaching, research, and prototyping vehicle. Policies and implementations studied in this framework have been transferred into Linux. For example, a kernel scalable queue lock originally designed in K42 was transferred to Linux, RCU[22] is similar to the safe memory and garbage collection in K42, and lockless scalable tracing technology has been integrated into LTT[26]. K42's framework will allow continuing technology transfer.

K42 currently runs on PowerPC (SMP) hardware and simulators (SimOS and Mambo), and is being ported to x86-64. As stated, K42 is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET to significant subsystems like Apache. Currently, K42 can directly execute 64-bit binaries compiled for Linux, and soon will be able to do the same for 32-bit binaries.

There are still some missing holes in K42's full Linux compatibility. There are system calls with unimplemented (less common) cases. For example, mmap protection only works on common cases. We have not yet implemented /proc (except for ps, this is primarily an impediment only to running administration tools). Our approach has been to add additional Linux functionality as applications require it. In some sense, K42 will never be fully one hundred percent compatible. We do not intend to be bug compatible, in fact, stack overflows and other such error conditions should they occur would be at different places in K42 than in Linux.

Although Linux is currently the only personality K42 supports, the base K42 mechanisms would allow support of other interfaces as well. Other flavors of Unix such as AIX and BSD would be fairly easy to support. Most of the challenging technical work is already in place. There would still be considerable detailed work to be done in ensuring structures get correctly translated. For example, getting the exact semantics and contents of the `stat` structure correct on such systems can be difficult because the structure is different under each of the Unix systems mentioned above. Other potential challenges involve providing dynamic linking for systems like AIX that use XCOFF instead of ELF. Supporting a significantly different interface, e.g., Windows, while doable, would be considerably more work. K42 is not much further along being able to support a Windows API than Linux is. The original intent in K42 was to allow multiple personalities to be efficiently supported, but we have not pursued this avenue of research. Other work[18] has examined supporting multiple personalities on a micro-kernel-like architecture.

Other operating systems emulate Linux, for example, all the BSD Unix systems do. They do so by reflecting traps to a vector with minor translation from Linux to native system calls. That is one part of our solution, the more straightforward part. The more difficult part for K42 is that it is not a real Unix system under the covers. Thus, one of the key challenges relate to the different abstractions the base system supports. Mach is another operating system with an implementation of Linux. The MkLinux Linux Server [1] has the entire Linux functionality in one single Mach task (instead of smaller specialized tasks communicating through Mach RPCs) in order to maximize reuse of the existing monolithic kernel.

K42 has similarities to a micro-kernel such as Mach[3] but more closely resembles the Exokernel[11]. The kernel, filesystem, etc., servers do not provide all the operating services, rather part of this functionality is integrated into the application's own address space. For the Linux API, a large majority of the conversion occurs in each application's address space. This approach is unique to K42.

There are other operating system projects that have aspects similar to K42. The ability to perform efficient IPCs as in L4[20] is important to K42's structure of employing user-level servers. K42's strategy for fault tolerance is similar to Disco[15] in that the plan is to run multiple simultaneous instances of K42 across varying sized machines. Other operating systems such as Spring[17] and Choices[9] have similar object-oriented goals but were motivated more by distributed system concerns. Our approach is best summarized by what was stated earlier, namely that performance was a central concern and although we follow a set of design philosophies in K42, we are willing to make compromises for the sake of performance.

5.1 Performance

K42 has been designed to achieve scalable performance. To date, this has been our primary focus. More recently, we have started to tune uniprocessor performance. The goal of our K42 design is to achieve near perfect scalability while still maintaining uniprocessor performance very close to that of other operating systems. Moreover, as Linux makes additions for multiprocessor performance, K42 should be able to match or better Linux's uniprocessor performance through our use of specialization and hot-swapping[5][24].

In this section we describe the SPEC SDET benchmark[2] and its performance on both K42 and Linux. The experiment shows that Linux out-performs K42 on a uniprocessor (we continue to work to reduce this gap), but that K42 significantly outperforms Linux on a medium size (24-way) multiprocessor.

The Standard Performance Evaluation Corporation (SPEC) Software Development Environment Throughput (SDET) benchmark consists of a script that executes a series common Unix commands and programs including ls, nroff, gcc, grep, etc. Due to missing infrastructure, for our experiemnts (both K42 and Linux), the SDET benchmark was modified by removing the system utilities `ps` and `df`. Each of the commands in the script are run in sequence. To examine scalability we ran one script per processor. We ran the same script on both K42 and Linux 2.4.19 as distributed by SuSE with the O(1) scheduler patch. All the user programs (bash, gcc, ls, etc.) are the exact same binary. The same version of glibc 2.2.5 was used, but modified on K42 to intercept and direct the system calls to the K42 implementations. The experiments were run on an S85 Enterprise Server IBM RS/6000 PowerPC bus-based cache-coherent multiprocessors with 24 600MHZ RS64-IV processors and 16GB of main memory.

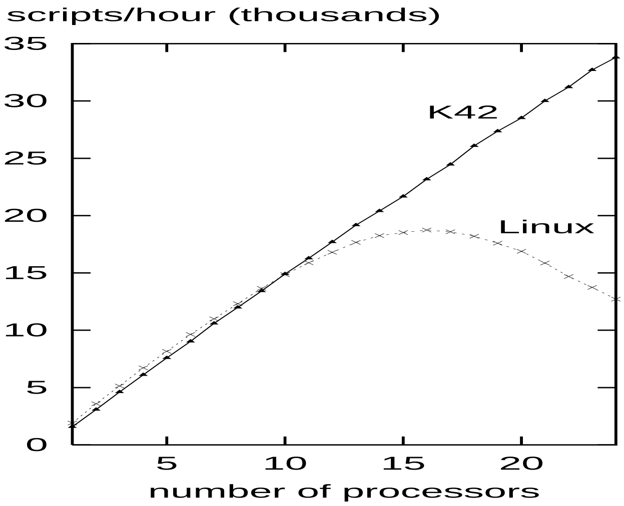

Figure 6: Results of running SDET on K42 and Linux on a 24-way multiprocessor

Figure 6 shows the results from the experiment. The script is timed and results are reported in thousands of scripts per hour. On a uniprocessor, Linux achieves 1905.2 scripts/hour and K42 runs 1575.0. K42 suffers about a twenty percent performance degradation on a uniprocessor. Linux reaches a peak of 18749.0 at 16 processors and by 24 processors executes at a rate of 12710.7 scripts/hour. K42's performance surpasses Linux by 10 processors, at which point K42 executes 14912.6 scripts/hour while Linux executes 14856.0 scripts/hour. K42 continues to scale well through 24 processors where its peak of 33808.1 scripts/hour is achieved yielding an efficiency of 89.4 percent. These results demonstrate the effectiveness of K42's scaling.

We continue to work on our scaling to increase our efficiency to 100 percent. There is also continuing work to help Linux to scale better, and 2.6 is expected to demonstrate better scalable performance. Concurrently, we are working on K42's uniprocessor performance.

One of the advantages of K42 is the object-oriented model and the resulting well-modularized structure allowing well-contained coding experiments to be implemented. We hope to see an increase in interest in using K42 for a rapid prototyping tool as well as a platform to pursue scalable and first-class customization research. Recently, there has been an increase in interest from academic collaborators looking to use K42 as a base to pursue research, and we look forward to continuing to support increased activity with K42.

6 Conclusions

K42 is a new open-source research operating system kernel designed from the ground up for scalable cache-coherent 64-bit multiprocessor systems. To provide access to a wide community and code base, we implemented mechanisms to support a Linux API and ABI. The desire to have a high performance, scalable, and maintainable operating system has resulted in several interesting features impacting our implementation of the Linux application environment on top of K42. These features include the implementation of kernel services in user space, object-oriented technology, avoidance of global locks, and avoidance of locking across calls to multiple objects. These characteristics in whole or in part will be representative of future systems that are designed to be scalable and maintainable. We described our experiences and lessons learned where these features impacted the implementation of the Linux API.

K42 is under active development at IBM T. J. Watson, and collaborating Universities. Interested parties may check out the project at: http://www.research.ibm.com/K42.

Acknowledgments

Thank you to the referees and especially our shepherd, Ray Bryant, for their careful reading of this document and their helpful suggestions. This work was supported by IBM's ASSR (Adventurous Systems and Software Research) and SSI (Server Systems Institute) initiatives.

References

[1] Mklinux. http://www.ota.be/linux/workshops/19970607/Generale/.

[2] SPEC SDM suite. http://www.spec.org/osg/sdm91/, 1996.

[3] M. Acceta, R. Baron, W. Bolosky., D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *USENIX*, pages 93–112, July 1986.

[4] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[5] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger, and Craig Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, 2002.

[6] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. *K42 Overview*. IBM Research, http://www.research.ibm.com/K42, August 2002.

[7] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. *Scheduling in K42*. IBM Research, http://www.research.ibm.com/K42, August 2002.

[8] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. *Utilizing Linux Kernel Components in K42*. IBM Research, http://www.research.ibm.com/K42, August 2002.

[9] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[10] Ulrich Drepper and Ingo Molnar. The new native posix thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf, October 2002.

[11] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, volume 29, 3–6 December 1995.

[12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, 22-25 February 1999.

[13] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceno, Russell Hunt, and Thomas Pinkney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[14] Shantanu Goel and Dan Duchamp. Linux device driver emulation in mach. In *USENIX Annual Technical Conference*, pages 65–74, 1996.

[15] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 18(3):229–262, 2000.

[16] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136. ACM Press, 1996.

[17] Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. In *Summer USENIX Conference*, pages 147–160, June 1993.

[18] Freeman L. Rawson III. Experience with the development of a microkernel-based, multi-server operating system. In *HotOS - Workshop on Hot Topics in Operating Systems*, pages 2–7, 1997.

[19] Xavier Leroy. The linuxthreads library. http://pauillac.inria.fr/˜xleroy/linuxthreads/.

[20] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.

[21] Mark M. Mathis, Nancy M. Amato, and Marvin L. Adams. A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. In *ICS Proceedings of the 14th ACM International Conference on Supercomputing*, pages 255–263, May 2000.

[22] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 26–29 June 2002.

[23] J. P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *CAN*, 20(1):pp. 5–44, March 1992.

[24] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX*, page to appear, San Antonio, TX, June 2003.

[25] B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pages 184–189, 1986.

[26] Karim Yaghmour. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.