The KeyKOS® Nanokernel Architecture

Alan C. Bomberger
A. Peri Frantz
William S. Frantz
Ann C. Hardy

Norman Hardy
Charles R. Landau
Jonathan S. Shapiro

ABSTRACT

The KeyKOS nanokernel is a capability-based object-oriented operating system that has been in production use since 1983. Its original implementation was motivated by the need to provide security, reliability, and 24-hour availability for applications on the Tymnet® hosts. Requirements included the ability to run multiple instantiations of several operating systems on a single hardware system. KeyKOS was implemented on the System/370, and has since been ported to the 680x0 and 88x00 processor families. Implementations of EDX, RPS, VM, MVS, and UNIX have been constructed. The nanokernel is approximately 20,000 lines of C code, including capability, checkpoint, and virtual memory support. The nanokernel itself can run in less than 100 Kilobytes of memory.

KeyKOS is characterized by a small set of powerful and highly optimized primitives that allow it to achieve performance competitive with the macrokernel operating systems that it replaces. Objects are exclusively invoked through protected capabilities, supporting high levels of security and intervals between failures in excess of one year. Messages between agents may contain both capabilities and data. Checkpoints at tunable intervals provide system-wide backup, fail-over support, and system restart times typically less than 30 seconds. In addition, a journaling mechanism provides support for high-performance transaction processing. On restart, all processes are restored to their exact state at the time of checkpoint, including registers and virtual memory.

This paper describes the KeyKOS architecture, and the binary compatible UNIX implementation that it supports.

1. Introduction

This paper describes the KeyKOS nanokernel, a small capability-based system originally designed to provide security sufficient to support mutually antagonistic users. KeyKOS consists of the nanokernel, which can run in as little as 100 Kilobytes of memory and includes all of the system privileged code, plus additional facilities necessary to support operating systems and applications. KeyKOS presents each application with its own abstract machine interface. KeyKOS applications can use this abstract machine layer to implement KeyKOS services directly or to implement other operating system interfaces. Implementations of EDX, RPS, VM/370, an MVS subset, and UNIX have been ported to the KeyKOS platform using this facility.

Tymshare, Inc. developed the earliest versions of KeyKOS to solve the security, data sharing, pricing, reliability, and extensibility requirements of a commercial computer service in a network environment.

Development on the KeyKOS system began in 1975, and was motivated by three key requirements: accounting accuracy that exceeded any then available; 24-hour uninterrupted service; and the ability to support simultaneous, mutually suspicious time sharing customers with an unprecedented level of security. Today, KeyKOS is the only commercially available operating system that meets these requirements.

KeyKOS began supporting production applications on an IBM 4341 in January 1983. KeyKOS has run on Amdahl 470V/8, IBM 3090/200 (in uniprocessor System/370 mode), IBM 158, and NAS 8023. In 1985, Key Logic was formed to take over development of KeyKOS. In 1988, Key Logic began a rewrite of the nanokernel in C. After 10 staff months of effort a nanokernel ran on the ARIX Corporation 68020 system, and the project was set aside. The project resumed in July of 1990 on a different processor, and by October of 1990 a complete nanokernel was running on the Omron Luna/88K. The current nanokernel contains approximately 20,000 lines of C code and less than 2,000 lines of assembler code.

This paper presents the architecture and design of the KeyKOS nanokernel, and the UNIX system that runs on top of it. In the interest of a clear presentation of the KeyKOS architecture, we have omitted a description of the underlying kernel implementation.

2. Architectural Foundations

KeyKOS is founded on three architectural concepts that are unfamiliar to most of the UNIX community: a stateless kernel, single-level store, and capabilities. Our experience indicates that understanding a single-level store model requires a fundamental shift in perspective for developers accustomed to less reliable architectures. It therefore seems appropriate to present these concepts first as a foundation on which to build the balance of the KeyKOS architectural description.

Stateless Kernel

An early decision in the KeyKOS design was to hold no critical state in the kernel. All nanokernel state is derived from information that persists across system restarts and power failures. For reasons of efficiency, the nanokernel does reformat state information in private storage. All private storage is merely a cache of the persistent state, and can be recycled at any time. When the discarded information is needed again, it is reconstructed from the information in nodes and pages (which are described below)

As a consequence, the nanokernel performs no dynamic allocation of kernel storage. This has several ramifications:

- The kernel is faster, since no complicated storage allocation code is ever run.
- The kernel never runs out of space.
- There is no nanokernel storage (such as message queues) that must be a part of the checkpoint.

The absence of dynamic allocation means that there can be no interaction between dynamic allocation strategies, which is the predominant source of deadlock and consistency problems in most operating systems.

The system outside the nanokernel is completely described by the contents of nodes and pages (see below), which are persistent. This state includes files, programs, program variables, instruction counters, I/O status, and any other information needed to restart the system.

In addition, the ability to recover all run-time kernel data from checkpointed state means that an interruption of power does not disrupt running programs. Typically, the system loses only the last few seconds of keyboard input. At UNIFORUM '90, Key Logic pulled the plug on our UNIX system on demand. Within 30 seconds of power restoration, the system had resumed processing, complete with all windows and state that had previously been on the display. We are aware of no other UNIX implementation with this feature today.

Single-Level Store

KeyKOS presents a persistent single-level store model. To the KeyKOS application, all data lives in persistent virtual memory. Only the nanokernel is aware of the distinction between main memory and disk pages. Periodic system-wide checkpoints guarantee the persistence of all system data. The paging system is tied to the checkpoint mechanism, and is discussed in the section on checkpointing, below. Persistence extends across system shutdown and power failure. Several IBM 4341 systems ran for more than three years across power failures without a logical interruption of service.

Like memory pages, KeyKOS applications are persistent. An application continues to execute until it is explicitly demolished. To the application, the shutdown period is visible only as an unexplained jump in the value of the real time clock, if at all. As a result, the usual issues surrounding orderly startup and shutdown do not apply to KeyKOS applications. Most operating systems implement a transient model of programs; persistence is the exception rather than the rule. A client operating system emulator may provide transient applications by dismantling its processes when they terminate.

The single-level store model allows far-reaching simplifications in the design of the KeyKOS system. Among the questions that the nanokernel does *not* have to answer are:

- How does the system proceed when it runs out of swap space? (It checkpoints.)
- How does the kernel handle the tear-down of a process? (It doesn't.)
- How is kernel state retained across restarts? (The kernel contains no state that requires checkpointing.)

Each of these areas is a source of significant complexity in other systems, and a consequent source of reliability problems.

Capabilities

KeyKOS is a capability system. For brevity, KeyKOS refers to capabilities as ***keys***. Every object in the system is exclusively referred to by one or more associated keys. Keys are analogous in some ways to Mach's ports. KeyKOS entities call upon the services of other entities by sending messages via a key. Message calls include a kernel-constructed ***return key*** that may be used by the recipient to issue a reply. Messages are most commonly exchanged in an RPC-like fashion.

What sets KeyKOS apart from other microkernels is the total reliance on capabilities without any other mechanisms. There are no other mechanisms that add complexity to the ideas or to the implementation. Holding a key implies the authority to send messages to the entity or to pass the key to a third party. If *A* does not have a key for *B*, then *A* cannot communicate with *B*. Applications may duplicate keys that they hold, but the creation of keys is a privileged operation. The actual bits that identify the object named by a key are accesible only to the nanokernel.

Through its use of capabilities and message passing, KeyKOS programs achieve the same encapsulation advantages of object-oriented designs. Encapsulation is enforced by the operating system, and is

available in any programming language. It is the complete security of this information hiding mechanism that makes it possible to support mutually suspicious users.

A fundamental concept in KeyKOS is that a program should obey the "principle of least privilege". To that end, the design of KeyKOS gives objects *no* intrinsic authority, and relies totally upon their keys to convey what authority they have. Using these facilities, the system is conveniently divided into small modules, each structured so as to hold the minimal privilege sufficient for its operation.

Entities may be referred to by multiple, distinct keys. This allows an entity that communicates with multiple clients to grant different access rights to the clients. Every key has an associated 8-bit field that can be used by the recipient to distinguish between clients. When the entity hands out a key, it can set the field to a known value. Because all messages received by the entity include the 8-bit value held in the key, this mechanism can be used to partition clients into service classes or privilege levels by giving each class a different key.

It is worthwhile to contrast this approach with the ring-structured security model pioneered in Multics and propagated in the modern Intel 80x86 family. The capability model is intrinsically more secure. A ring-structured security policy is not powerful enough to allow a subsystem to depend on the services of a subsystem with lesser access rights. Ring policies intrinsically violate the principle of least privilege. In addition, ring-based security mechanisms convey categorical authority: any code running in a given layer has access to all of the data in that layer. Capability systems allow authority to be minimized to just that required to do the job at hand.

Using a capability model offers significant simplifications in the nanokernel. Among the questions that the nanokernel does *not* have to answer are:

- Does this user have the authority to perform this operation? (Yes – if you hold the key you can send the message.)
- How do I allocate enough kernel memory to perform name resolution on a variable length name? (The kernel never deals with names, only keys.)
- Where does this file name get inserted in this directory? (The nanokernel does not deal with file names or directories.)

Because the nanokernel has no naming mechanism other than capabilities, entity naming is intrinsically decentralized. As a result, extending KeyKOS to multiprocessors is straightforward. KeyKOS applications cannot tell if they are running on a uniprocessor or a multiprocessor.

3. Major Nanokernel Features

The nanokernel includes all of the supervisor-mode code of the system. The entire kernel is implemented in approximately 20,000 lines of reasonably portable C code, and 2,000 lines of 88x00 assembly code. Of the assembly lines, 1,000 lines are in the context switch implementation. This compiles to roughly 60 Kilobytes of executable code. While running, the nanokernel requires as little as 100 Kilobytes of main memory.

The nanokernel is the only portion of the system that interprets keys. No other program has direct access to the bits contained in the keys, which prevents key forgery. In addition, the nanokernel includes code that defines the primitive system objects. These objects are sufficient to build the higher-level abstractions supported by more conventional operating systems. The nanokernel provides:

- multiprogramming support, primitive scheduling, and hooks for more sophisticated schedulers running as applications;
- a single-level store, as discussed above;
- separate virtual address space(s) for each KeyKOS process;
- redundant disk storage for system-critical information;
- a system-wide checkpoint-restart feature;
- journaling pages exempt from checkpoint for database and transaction processing support;
- keys by which messages are sent from one application to another;
- primitive and limited access to individual I/O devices;
- interpretation of keys that hides the location of the object on disk or in main memory.

During normal operation, KeyKOS executes a system-wide checkpoint every few minutes to protect from power failures, most kernel bugs, and detected hardware errors. Both data *and processes* are checkpointed. All run-time state in the nanokernel can be reconstructed from the checkpoint information. Except for the initial installation, the system restarts from the most recent checkpoint on power up.

In addition to local checkpoint support, the nanokernel provides for checkpoints to magnetic tape or remote hot-standby systems. This allows a standby system to immediately pick up execution in the event of primary system failure.

4. Fundamental KeyKOS Objects

The KeyKOS kernel supports six types of fundamental objects: ***devices***, ***pages***, ***nodes***, ***segments***, ***domains***, and ***meters***.

Devices

The nanokernel implements low-level hardware drivers in privileged code. The supervisor-mode driver performs message encapsulation and hardware register manipulation. Except where performance compels otherwise, KeyKOS applications implement the actual device drivers.

Pages

The simplest KeyKOS object is the page. Page size is dependent on the underlying hardware and storage architectures, but in all current implementations is 4 Kilobytes. Every page has one or more persistent locations on some disk device, known as its ***home location***. The KeyKOS system manages a fixed number of pages that are allocated when the system is first initialized. This number can be increased by attaching additional mass storage devices to the system.

A page is designated by one or more ***page keys***. Pages honor two basic message types: read, and write. When pages are mapped into a process address space, loads and stores to locations in a page are isomorphic to read and write messages on the page key. When a message is sent to a page that is not in memory, the page is transparently faulted in from backing store so that the operation can be performed.

Applications that perform dynamic space allocation hold a key to a ***space bank***. Space banks are used to manage disk resource allocation. The system has a master space bank that holds keys to all of the pages and nodes in the system.[1] One of the operations supported by space banks is creating subbanks, which are subbanks of the master space bank. If your department has bought the right to a megabyte of storage, it is given a key to a space bank that holds 256 page keys. Space banks are a type of domain.

Nodes

A node is a collection of keys. All keys in the system reside in nodes. A ***node key*** conveys access rights to a node, and can be used to insert or remove keys from a node. Like pages, nodes can be obtained from space banks. In all current KeyKOS implementations, a node holds precisely 16 keys.

Nodes are critical to the integrity of the system. The KeyKOS system vitally depends on the data integrity of node contents. As a result, all nodes are replicated in two (or more) locations on backing store. In keeping with the general policy of not performing dynamic allocation in the kernel, and because the integrity requirements for nodes are so critical, KeyKOS does not interconvert nodes and pages.

Segments

A segment is a collection of pages or other segments. Segments are used as address spaces, but also subsume the function of files in a conventional operating system. Segments can be combined to form larger segments. Segments may be sparse; they do not necessarily describe a contiguous range of addresses.

Nodes are the glue that holds segments together. KeyKOS implements segments as a tree of nodes with pages as the leaves of the tree. This facilitates efficient construction of host architecture page tables. Because nodes and pages persist, so do segments. The system does not need to checkpoint page table data structures because they are built exclusively from the information contained in segments.

Meters

Meters control the allocation of CPU resources. A ***meter key*** provides the holder with the right to execute for the unit of time held by the meter. The KeyKOS kernel maintains a ***prime meter*** that represents the time interval from the present until the end of time. Like space banks, meters can be subdivided into submeters. Every running process holds a meter key that authorizes the process to execute for some amount of time.

KeyKOS processes can be preempted. Holding a key to a meter that provides 3 seconds of CPU time does not guarantee that the process will run for 3 contiguous seconds. In the actual KeyKOS implementation, time slicing is enforced by allowing a process to run for the minimum of its entitled time or the time slice unit. Political scheduling policies may be implemented external to the kernel.

Domains

Domains perform program execution services. They are analogous to the virtual processors of the POSIX threads mechanism. It was a design goal not to restrict the architecture available to the user. A consequence is that KeyKOS supports virtual machines. Domains model all of the non-privileged state of the underlying architecture, including the general purpose register set, floating point register set, status

---

1 The system can support multiple master space banks. In a B3 implementation, system pages would be partitioned into multiple security classes, and there would be one master space bank for each class.

registers, instruction set architecture, etc. A domain interprets a program according to the hardware user-mode architecture. Domains are machine-specific, though we have considered the implementation of domains that perform architecture emulation (e.g. for DOS emulation on a RISC machine).

In addition to modeling the machine architecture, domains contain 16 general key slots and several special slots. The 16 general slots hold the keys associated with the running program. When a key occupies one of the slots of a domain, we say that the program executing in that domain holds the key. One of the special slots of the domain is the ***address slot***. The address slot holds a segment key for the segment that is acting as the address space for the program. On architectures with separate instruction and data spaces, the domain will have an address slot for each space. Each domain also holds a meter key. The meter key allows the domain to execute for the amount of time specified by the meter.

KeyKOS processes are created by building a segment that will become the program address space, obtaining a fresh domain, and inserting the segment key in the domain's address slot. The domain is created in the *waiting* state, which means that it is waiting for a message. A threads paradigm can be supported by having two or more domains share a common address space segment.

Because domain initialization is such a common operation, KeyKOS provides a mechanism to generate "prepackaged" domains. A ***factory*** is an entity that constructs other domains. Every factory creates a particular type of domain. For example, the queue factory creates domains that provide queuing services. An important aspect of factories is the ability of the client to determine their trustworthiness. It is possible for a client to determine whether an object created by a factory is secure. Understanding factories is crucial to a real understanding of KeyKOS, but in the interest of brevity we have elected to treat factories as "black boxes" for the purposes of this paper. To understand the UNIX implementation it is sufficient to think of factories as a mechanism for cheaply creating domains of a given type.

5. Message Passing

The most important operation supported by the nanokernel is message passing. Messages sent from one domain to another involve a context switch. In order to encourage the separation of applications into components of minimal privilege, the nanokernel's message transfer path has been carefully optimized. The KeyKOS inter-domain message transfer path ranges from 90 instructions on the System/370 to 500 cycles on the MC88x00.

Messages are composed of a parameter word (commonly interpreted as a method code), a string of up to 4096 bytes, and four keys. A domain constructs a message by specifying an integer, contiguous data from its address segment, and the keys to be sent. Only keys held by the sender can be incorporated into a message. Once constructed, the message is sent to the object named by a specified key. Sending a message is sometimes referred to as key invocation.

KeyKOS supplies three mechanisms for sending messages. The ***call*** operation creates a ***resume key***, sends the message to the recipient, and waits for the recipient to reply using the message's resume key. While waiting, the calling domain will not accept other messages. A variant is ***fork***, which sends a message without waiting for a response. The resume key is most commonly invoked using a ***return*** operation, but creative use of call operations on a resume keys can achieve synchronous coroutine behavior. The return operation sends a message and leaves the sending domain available to respond to new messages. All message sends have copy semantics.

The nanokernel does not buffer messages; a message is both sent and consumed in the same instant. If necessary, invocation of a key is deferred until the recipient is ready to accept the message. Message buffering can be implemented transparently by an intervening domain if needed. The decision not to buffer messages within the nanokernel was prompted by the desire to avoid dynamic memory allocation, limit I/O overhead, keep the context switch path length short, and simplify the checkpoint operation.

A message recipient has the option to selectively ignore parts of a message. It may choose to accept the parameter word and all or part of the byte string without accepting the keys, or accept the parameter word and the keys without the data.

6. Checkpointing and Journaling

KeyKOS provides for regular system-wide checkpoints and individual page journaling. Checkpoints guarantee rapid system restart and fail-over support, while journaling provides for databases that must make commit guarantees.

The Checkpoint Mechanism

The KeyKOS nanokernel takes system-wide checkpoints every few minutes. Checkpoint frequency can be adjusted by the administrator at any time without interruption of service.

The KeyKOS system maintains two disk regions as checkpoint areas. When a checkpoint is taken, all processes are briefly suspended while a rapid sweep is done through system memory to locate modified pages. No disk I/O is done while processes are frozen. Once the sweep has been done, processes are resumed and all modified pages are written to the current checkpoint area. Once the checkpoint has completed, the system makes the other checkpoint area current, and begins migrating pages from the first checkpoint area back to their home locations. Checkpoint frequency is automatically tuned to guarantee that the page migration process will complete before a second checkpoint is taken. Because the migration process is incremental, a power failure during migration never leads to a corrupt system.

An implementation consequence of this approach to checkpointing is unusually efficient disk bandwidth utilization. Checkpoint, paging, and page migration I/O is optimized to take advantage of disk interleave and compensates for arm latencies to minimize seek delays. This accounts for all page writes. The aggregate result is that KeyKOS achieves much higher disk efficiency than most operating systems. If the system bus is fast enough, KeyKOS achieves disk bandwidth utilization in excess of 90% on all channels.

It is worth emphasizing that the checkpoint is not simply of files, but consists of all processes as well. If an update of a file involves two different pages and only one of the pages has been modified at the time of the checkpoint, the file will not be damaged if the system is restarted. When the system is restarted the process that was performing the update is also restarted and the second page of the file is modified as if there had been no interruption. A power outage or hardware fault does not leave the system in some confused and damaged state. The state at the last checkpoint is completely consistent and the system may be restarted from that state without concern about damaged files.

The Journaling Mechanism

For most applications, it is acceptable for the system as a whole to lose the last few minutes work after a power outage. Transaction processing and database systems require the additional ability to commit individual pages to permanent backing store on demand. Using the journaling mechanism, a domain may request that changes to a particular page be synchronously committed to permanent storage. If a system failure occurs between the commit and the next completed checkpoint, the journaled page will remain committed after the system restarts. It is the responsibility of the requesting domain to see to the semantic consistency of such pages.

The journaling mechanism commits pages by appending them to the most recent committed checkpoint. As a result, journaling does not lead to excessive disk arm motion. A curious consequence of this implementation is that transaction performance under KeyKOS *improves* under load.[2] This is due to locality at two levels. As load increases, it becomes common for multiple transactions to be committed by a single page write. In addition, performing these writes to the checkpoint area frequently allows the journaling facility to batch disk I/O, minimizing seek activity. The KeyKOS transaction system significantly exceeds the performance of competing transaction facilities running on the same hardware. CICS, for example, is unable to commit multiple transactions in a single write.

7. Exception Handling

Process exceptions are encapsulated by the nanokernel and routed to a user-level handler known as a ***keeper***. The keeper technology of KeyKOS brings all exception policy to application level programs outside of the nanokernel. A keeper is simply a domain that understands the exception messages delivered by the kernel; it is in all regards an ordinary domain. Since the UNIX implementation relies heavily on the Domain Keeper technology, the ideas and specifications concerning Keepers will be discussed before we delve into the UNIX specifics.

Recall that a KeyKOS application has an address space, a domain, and a meter. Each of these objects holds a start key to an associated domain known as its ***keeper***. When the process performs an illegal, unimplemented, or privileged instruction, the error is encapsulated in a message which is sent to the appropriate keeper, along with the keys necessary to transparently recover or abort the application. The keeper may terminate the offending program, supply a correct answer and allow execution to continue, or restart the offending instruction.

Each segment has an associated ***segment keeper***. The segment keeper is a KeyKOS process that is invoked by the kernel when an invalid operation, such as an invalid reference or protection violation, is performed on a segment. Page faults are fielded exclusively by the nanokernel.

By appropriate use of a ***meter keeper***, more sophisticated scheduling policies can be implemented. The meter keeper is invoked whenever the meter associated with a domain times out. A thread supervisor might implement a priority scheduling policy by attaching the same meter keeper to all threads, and having the meter keeper parcel out time to the individual threads according to whatever policy seemed most sensible.

The most interesting keeper for this paper is the ***domain keeper***. The domain keeper is invoked when a trap or exception is taken. When a domain encounters an exception (system call, arithmetic fault, invalid operation, etc.) the domain stops executing and the domain keeper receives a message. The message contains the non-privileged state of the domain (its registers, instruction counter, etc.), a domain key to the domain, and a form of resume key that the keeper can use to restart the domain. When the faulting domain is restarted, it resumes at the instruction pointed to by the program counter. If necessary, the domain keeper can adjust the PC value of the faulting domain before resumption.

8. A KeyKOS-Based UNIX Implementation

In July of 1990, Key Logic undertook to produce a binary-compatible prototype UNIX implementation for the Omron Luna/88K. The effort had two principle goals. The first was to rapidly construct a system that could run existing Omron application binaries. Based on Mach 2.5, the Omron implementation provides a reasonably complete version of the Berkeley UNIX system, including the X11r4 windowing

---

2 Up to a point. There ain't no such thing as a free lunch.

system. KeyNIX was implemented by a single developer over a six month period, without reference to the UNIX source code. The implementation was partly based on an earlier Minix port that had been built for KeyKOS on the System/370.

Our experience in implementing other systems was that breaking an application into separate function-oriented domains simplified the application enough to improve overall performance. A second goal of the KeyNIX implementation was to learn where such decomposition into separate domains would cause performance degradation. In several areas, multi-domain implementations were tried where the problem area was clearly a boundary case in order to explore the limitations of the domain paradigm.

UNIX Services

Broadly speaking, the UNIX system provides the following services:

- Process management (fork, exec, exit, kill),
- File system and namespace services (open, link),
- I/O services (read, write, stat, ...)
- Timing facilities (sleep, nap, sometimes socket)
- Messaging (sockets, pipes)
- Memory management (mmap, mprotect)
- Signals
- Device support
- Networking (TCP/IP, NFS)

With the exception of networking, KeyNIX implements all of these services. Adding networking support would be straightforward, but was not part of the prototype effort.

9. Structure of KeyNIX

Under KeyNIX, every UNIX process runs as a KeyKOS domain with a segment as its address space. A standard KeyKOS segment keeper is used to manage stack and heap growth within the address space segment. From the outside, the UNIX process model is essentially unchanged. No KeyNIX code is mapped in with the application, nor is special linking required. The application address spaces are bit for bit identical. This severely penalizes all trivial system calls, and is a significant departure from the implementations used by other microkernels. The penalty could be eliminated in a dynamic-library based standard such as System V Release 4.

To support UNIX processes, we implemented a domain keeper, known as the ***UNIX Keeper***. The UNIX keeper interprets the system call and either manages the call itself or directs request to other domains for servicing. The implementation includes a number of cooperating domains, is shown in Figure 1. The gray box surrounds the domains and segments that are replicated for each UNIX process.

Each of these domains in turn depends on other domains provided by the KeyKOS system. For example, a small integer allocator domain is used to allocate monotonically increasing inode numbers. To simplify

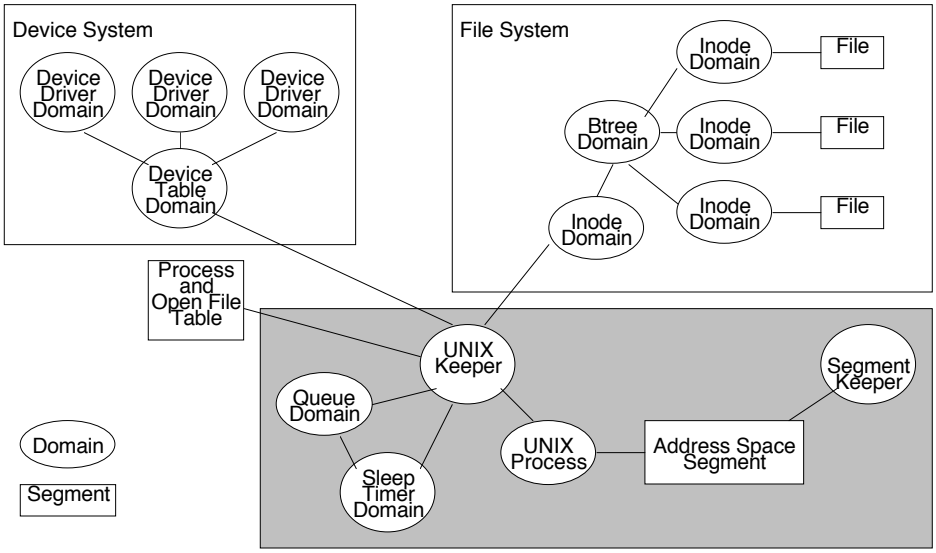

Figure 1: Structure of the UNIX Implementation

the picture, domains that are not essential to understanding the structure of the UNIX implementation have been omitted.

One Kernel per Process

An unusual aspect of the KeyNIX design is that every UNIX process has a dedicated copy of the UNIX Keeper. When a process forks, the UNIX Keeper is replicated along with the process. By providing a separate UNIX keeper to each UNIX application, the scope of UNIX system failures is reduced to a single process. If a given UNIX process *does* manage to crash its copy of the operating system, no other processes are impacted. An individual kernel is very hard to crash. To crash the entire UNIX system essentially requires physical abuse of the machine or its power supply.

State that must be shared between multiple UNIX keepers, including the process table and open file table, is kept in a segment shared by all UNIX Keepers. Each process has a description block (a process table entry) that describes the process' address space, open files, and signal handling. Process table entries contain chains of child processes and pointers to the parent process table entry. Each open file has an entry in the Open File Table which keeps track of the number of processes that have the file open, the attributes of the file, and a pointer to the data structures that buffer the file data in memory.

The UNIX keeper implements UNIX process and memory management services by calling directly on the underlying KeyKOS services. The nanokernel handles virtual memory mapping and coherency directly. When a program is loaded by *exec*(2), the UNIX keeper builds an address space segment and copies the executable file segment into it. Manipulating the KeyKOS segment structures is simpler than the equivalent structure manipulations in UNIX, and allows the UNIX keeper to be largely platform independent. The nanokernel is responsible for the construction of mapping tables for the particular hardware platform.

The KeyNIX File and Device System

The UNIX Keeper holds a key to the root inode of the KeyNIX file system. Each inode contains the usual UNIX inode information, and is implemented by a KeyKOS domain. If the inode denotes a file,

the inode domain holds a key to a KeyKOS segment containing the file data. If the inode denotes a device, the device major and minor numbers are contained in the inode.

By making each UNIX inode into a KeyKOS domain, the UNIX Keeper does not have to manage an inode cache or worry about doing I/O to read and write inodes. When the Keeper needs to read the status information from an inode it sends a message to the Inode object and waits for the reply. Similar arguments apply to other operations. The Keeper does not cache file or directory blocks, and does not maintain paging tables for support of virtual memory. All of these functions are handled by the nanokernel.

In the original KeyNIX implementation, directory inodes contained a key to a B-tree domain that was an underlying KeyKOS tool. An analysis of typical directory sizes led to the conclusion that it would be more space efficient to implement small directories (less than five entries) in the inode itself. As a result, directory protocol requests are implemented directly by the inode domain. If the inode does not denote a directory it fails the directory messages appropriately. A curious artifact of this approach is that directory order is alphabetical order. This is occasionally visible to end users as a change of behavior in programs that search directories without sorting them.

When opening a file, the UNIX Keeper issues a message to the file system root inode domain. This domain in turn calls on other domains, until ultimately the request is resolved to a segment key that holds the file content. Once the file has been located, the UNIX keeper maps the segment into the keeper address space and adds an entry to the open file table. The open file table is shared by all UNIX Keepers, and is used to hold dynamically changing information such as the file's current size and last modification date.

When opening a device, the UNIX Keeper receives the major and minor device number from the appropriate inode domain. The major number is in turn handed to the device table domain, which returns a key to the domain that implements the driver. Drivers implemented in the prototype include character I/O, graphics console (supports the X Window System), the null device, sockets, kmem, and the mouse. Support for /dev/kmem is limited to forging those responses necessary to run the *ps*(1) command. In most cases, the device driver domain consists of the original UNIX device driver code linked with a support library that maps the UNIX driver-kernel interface onto KeyKOS key invocations.

The Problem of Signals

The most difficult part of the KeyNIX implementation, was support for the *signal*(2) mechanism. One of the deliberate design decisions of KeyKOS is that domains are single threaded. A domain is either waiting for a message, waiting for a reply to a message, or processing a message. There is no mechanism for stacking messages. This decision increases the reliability of the KeyKOS system, but occasionally requires that queuing domains be inserted into an otherwise straightforward remote procedure call.

UNIX signals are asynchronous with respect to the receiving process. As a result, the implementation of the signal mechanism is one of the more complicated and pervasive (not to say perverse) aspects of the UNIX kernel.[3]

To ensure that the UNIX Keeper is always able to receive signal notifications promptly, trivial queuing domains are required where an operation might block or complete slowly. The purpose of these domains is to queue messages to devices such as ttys and pipes that might otherwise delay the receipt of signals by the UNIX Keeper. The UNIX Keeper delivers these messages through the queue domain, and waits asynchronously for the queue domain to send a message indicating completion of the requested service.

---

3 This is also a significant problem for debugging interfaces, such as */proc*(4) and *ptrace*(2).

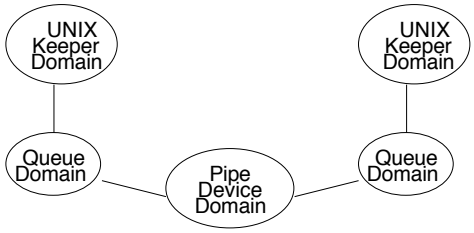

Figure 2: Domains in a Pipe

In effect, a series of fork messages are used to implement a non-blocking remote procedure call to the device domain in order to ensure that the UNIX kernel is always ready to receive another message.

The queue insertion approach has unfortunate consequences for slow devices (with disk devices one can reasonably assume instant service and duck the issue), and severly impacted communication facilities such as pipes or sockets, as shown in Figure 2.

These mechanisms are penalized by the requirement from both sides to remain able to receive signals while proceeding with the I/O transfer. The impact is easily visible in the performance of KeyNIX pipes. A better alternative is discussed below.

Expected Performance

To the best of our knowledge, the KeyNIX system uses far more processes than any other microkernel-based UNIX implementation. Reactions to the KeyNIX design from UNIX developers range from shocked to appalled at the profligate use of processes. UNIX developers find it difficult to accept that the task switch cost can be lower than the data management code that it replaces. We find this ironic, as one of the major innovations of the UNIX system was the notion that processes were cheap.

The object paradigm was at the heart of the design of the KeyKOS system and, as a result, the task switch costs are very much lower than in traditional systems and several times lower than in competing microkernels such as MACH and Chorus. On the Motorola 88x00 series, a typical message send takes less than 500 cycles.[4] The low cost of task switches makes it possible to obtain better performance with much simpler software by taking an object-oriented approach to the decomposition of the system. The UNIX implementation described here takes considerable advantage of KeyKOS building blocks. The complete UNIX kernel implementation is approximately 16,000 lines of C code.

Known Incompatibilities

The KeyNIX implementation is 99% compatible with the Omron BSD 4.3 implementation. While KeyNIX could be equally compatible with MACH 2.5, the existing prototype is not. There are four significant incompatibilities in the prototype:

1. The application prolog ("crt0") in MACH 2.5 initializes certain MACH ports. Because KeyNIX does not yet implement MACH ports, applications built with the MACH 2.5 crt0.o do not run under KeyNIX.

---

4 This time includes the context switch and copying both data and keys. The Motorola implementation is the slowest implementation to date.

2. MACH 2.5 port functions are accessed by a trap instruction in the same fashion as are UNIX system calls. KeyNIX does not implement these traps.

3. In MACH 2.5, the *fork*(2) system call does the same port initialization for the new task that was done by "crt0" in the parent task. This change is not implemented in KeyNIX.

4. MACH 2.5 does not implement the *sbrk*(2) system call. This call is handled by a library routine that uses the "VMALLOC" of MACH 2.5 to handle memory expansion and contraction.

5. The KeyNIX text segment is writable, which can impact buggy programs. This is the result of a quick and dirty implementation, and could be easily fixed.

Programs compiled on the Luna 88K under MACH 2.5 that are to be run in the KeyNIX system must be linked with a new prolog and new library stubs for *fork*(2) and *sbrk*(2). In cases where the ".o" files exist, there is no need to recompile the programs, but the programs must be relinked.

The existing prototype does not support all BSD 4.3 system calls. The major criterion for choosing what to implement and what not to implement was the need to run X-Windows, *csh*(1), *ls*(1) and similar useful utilities. If the system call is not needed to run these applications then it is not implemented. There are a number of calls that are implemented in a limited fashion, again sufficiently to run the required applications. As an example, *csh*(1) makes *usage*(2) calls but does not depend on the answers for correct behavior. *Usage*(2) always returns the same fixed values and is not useful as a measuring tool as a result.

To get an intuitive sense of the compatibility achieved, it may suffice to say that all of the application binaries running on KeyKOS were obtained by copying the binary file from the existing BSD 4.3 system. The X Window System, compilers, shells, file system utilities, etc. all run without change under KeyNIX.

10. Performance Comparison

A limited performance comparison was made between the KeyNIX prototype and the Omron MACH 2.5 implementation. A more careful analysis would be required for any serious evaluation of the two systems for production use. KeyNIX got mixed results for common system call sequences:

Operation	Iterations	KeyNIX	MACH 2.5	Ratio
getpid();	10,000	12,000/sec	30,000/sec	0.4
open();close();	1,000	714/sec	2777/sec	0.26
fork();exit();	100	64/sec	10/sec	6.4
exec();	100	151/sec	12/sec	11.6
sbrk(4096);sbrk(-4096)	100	2564/sec	181/sec	14

I/O performance was equally mixed:

Operation	KeyNIX	MACH 2.5	Ratio
Pipe (round trip)	.588 Mbyte/sec	1.05 Mbyte/sec	.56
Disk access program	4 seconds	26 seconds	6.5

As anticipated, the simplification achieved by adding domains doesn't always lead to better performance. The cases that the KeyNIX prototype handled poorly have straightforward corrections which are discussed below.

Simple System Calls

Simple system calls include calls such as *getprocid*(2), *putprocid*(2), and *gettimeofday*(2), which are essentially accessor functions. A trap is taken, but the system call itself performs little or no interesting activity within the kernel. The KeyNIX system is binary compatible with this approach.

The MACH 2.5 implementation is able to execute these system calls 2.5 times as fast as the KeyNIX system because no context switch is involved. MACH 3 uses special system call libraries to implement some of these functions in the UNIX process address space. A similar approach would be possible in KeyNIX if the system calls were implemented in dynamic libraries, as in System V Release 4, or if binary compatibility could be sacrificed. We were surprised that KeyNIX did so well on this comparison.

Open and Close

To explore the limits of domain performance, we elected to implement each inode as an individual domain. On the basis of our previous experiences, it seemed likely that the simplification achieved by this approach would overcome the overhead of multiple domains. With the benefit of hindsight, we were mistaken, and the performance of *open*(2) suffered excessively. The namei() routine within the UNIX kernel is heavily used, and the decision to use multiple domains in effect inserted four context switches into the inner loop(for two round-trip RPC's).[5] In a small program that simply opens and closes a single file 1,000 times, the MACH 2.5 system outperformed the KeyNIX system by nearly four to one (3.89). Alternative implementations are discussed below.

Fork and Exit

Because the UNIX programming model assumes that processes are cheap, the performance of *fork*(2) is critical to the overall performance of the system. In KeyKOS, the equivalent to *fork*(2) is even more critical, and is possibly the most carefully optimized path in the nanokernel. We therefore expected KeyNIX to do well on *fork*(2) calls. KeyNIX outperforms MACH 2.5 by a little more than six to one.

The current KeyNIX implementation suffers from an extremely naive loader implementation in the UNIX keeper. When performing a *fork*(2), a complete copy of the process address space is made. The implementation could be improved by sharing the read-only text pages rather than copying their content. In addition, it would not be difficult to implement UNIX copy-on-write semantics as part of the segment keeper that services faults on the UNIX address space. Neither of these optimizations was performed in the prototype due to time constraints, and we would expect each to result in substantial improvements.

Exec

Given the naive loader implementation, we were pleasantly surprised to find that KeyNIX outperformed MACH 2.5 by better than eleven to one on *exec*(2) calls. The test program simply calls *exec*(2) one hundred times and exits. Implementing shared text would significantly improve the KeyNIX results.

Sbrk

In order to compare the performance of the *sbrk*(2) system call, a program was written to repeatedly grow and shrink the heap. 100 calls to sbrk(4096) and sbrk(-4096) were executed with a fetch of a byte from the newly allocated memory. The fetch of the byte forces the UNIX implementation to actually allocate

---

5 One round trip to access the inode domain, the second to access the directory domain.

the main store for the page, and consequently forces the page to be deallocated when the heap segment size is reduced. KeyNIX outperformed the MACH 2.5 implementation by fourteen to one, which was consistent with our expectation.

Pipe Bandwidth

Pipe performance is one of the areas where we expected KeyNIX to suffer. In order to compare the pipe implementations, a megabyte of data was passed through a pipe to a child process task and back in 1000 byte chunks. The MACH 2.5 implementation outperformed KeyNIX by nearly two to one.

This result is principally due to the insertion of queue domains into both ends of the pipe, imposing considerable context switch overhead. In retrospect, we could have eliminated the queues and depended on the fact that asynchronous signal delivery timing is not guaranteed by the UNIX process model. In particular, correct UNIX programs cannot depend on the fact that *interprocess* signals will interrupt a system call in the receiving process. Taking advantage of this loophole would allow for a much simpler and faster implementation.

Disk File I/O

To measure disk performance, we built a program to create a large test file and read it repeatedly. The I/O model of KeyNIX and MACH 2.5 are so radically different that other comparisons are very difficult. Uncached writes, for example, are dominated by disk arm movement, so a comparison of such activity is unenlightening. The times reported are the elapsed time to write and then read a one megabyte file ten times. KeyNIX outperforms MACH 2.5 by better than six to one.

KeyNIX I/O performance is a direct result of the underlying KeyKOS I/O design. KeyKOS never writes to disk as a direct result of writing to a file. All writes to the disk are part of the paging, checkpoint, and migration system.

To determine the impact of the checkpoint process on the test, we arranged for KeyKOS to perform a checkpoint and migration in parallel. This process increases the KeyKOS time to 4.4 seconds, giving a performance ratio of 5.9 to one. To the best of our knowledge, the prototype KeyNIX system achieves the highest I/O bandwidth utilization of any UNIX system today.[6] KeyKOS's I/O performance makes the overall performance of many applications better under KeyNIX than under a more conventional system, and appears to more than balance the prototype's performance deficiencies.

Performance Summary

The overall performance of the KeyNIX system is quite comparable with MACH 2.5. Some operations are slower and some quite a bit faster. A user using X-Windows doing VI and using a variety of shell commands and scripts is unaware of any significant performance difference between MACH 2.5 and KeyNIX.

---

[6] We are well aware of the significance of the I/O subsystem design in this claim, and believe that the claim would hold up when examined with other I/O subsystems and bus architectures. On the System/370, KeyKOS achieves channel utilization of better than 95% on all channels. With current SCSI technology, KeyKOS's disk utilization is limited by the SCSI channel performance.

11. Implementation Alternatives

In the course of the prototype effort, we came up with several ways to simplify the UNIX keeper and to cut down on some of the overhead. Each of these ideas represents a compromise in the use of domains and multiple instantiation.

Domains for Process and File Table Manipulation

The current process table segment is an array of process table entries. The UNIX process id is used to index the table. Process numbers are reallocated quickly, which leads to certain problems in the human interface for system maintenance. Also there are circumstances when process table entries should be chained so that children can be located more quickly. This is best handled by introducing a domain for process table entry manipulation that allocates and chains process table entries. The UNIX keeper continues to reference its own process table entry directly, but accesses other process table entries (to obtain a signal key) using the process table management domain. Similarly, the open file table could be implemented by a domain. These modifications would both simplify the UNIX keeper and remove the primary impediment to distribution of the KeyNIX implementation on loosely coupled architectures.

Small Files

The data for small files could be kept in nodes instead of segments. A small file might be a single-level tree of nodes with up to 16 leaf nodes each holding 176 bytes of data. When the 17th node is required the file is converted to a segment. The inode domain would convert the file to a segment when it is opened, and on the last close would convert it back into node form if it is small enough. This would allow KeyNIX to achieve more efficient storage of small files than current UNIX systems.

File System Domain

Opening files is a crucial operation in UNIX systems, and the domain-per-inode approach is not nearly fast enough. Two alternative implementations would have delivered competitive performance.

The first approach is to build the entire directory and inode support structure for a file system into a single domain, while continuing to implement files as individual segments. This would eliminate almost all of the context switching performed in the file subsystem, and would probably outperform the MACH 2.5 implementation.

The second alternative is to implement a compatibility library that would enable us to simply compile a vnodes-compatible file system into a domain. Using this approach, the entire file system would reside in a single KeyKOS segment, and bug-for-bug compatibility is achievable. This approach is something like the File Manager tasks of CHORUS and MACH 3. In practice, supporting vnodes file systems is probably a compatibility requirement for a commercial UNIX implementation, but system reliability suffers greatly from this requirement.

Our current preference would be the first alternative, mainly to eliminate the bugs of the existing file system implementations. In addition, we feel that this approach significantly simplifies recovery in the event of a disk block failure, as it eliminates the need for a complicated file system consistency checker.

12. Conclusions

The KeyKOS nanokernel has been running in production environments for nine years. It is proven technology, and we feel that the architecture and implementation have much to offer to the computing community at large. A serious development project could far exceed the performance that we obtained from the six month UNIX prototype effort.

KeyKOS represents a pardigmatic shift in operating system technology. It is therefore difficult to make direct comparisons with other approaches. A pure capability architecture brings fundamentally greater discipline, control, and reliability to application construction. In the long term, we feel that this degree of reliability is necessary to realize the productivity promises of the information age.

For further information on KeyKOS:

U.S. Mail: Norman Hardy
143 Ramona Road
Portola Valley, CA 94028

Phone: (415) 851-2582

Email: norm@xanadu.com

13. Bibliography

1. Theodore A. Linden, "Operating System Structures to Support Security and Reliable Software," NBS Technical Note 919, U.S. Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology, August 1976. (Also published in ACM *Computing Surveys*, 8, 4, December 1976, pp. 409-445).

2. Norman Hardy, "The Keykos Architecture," Operating Systems Review, September, 1985.

3. *Introduction to KeyKOS Concepts*, KL004, Key Logic, 1988.

4. *KeyKOS/370 Principles of Operation*, KL002, Key Logic, 1988.

5. *KeyKOS Architecture*, KL028, Key Logic, 1988.

6. Butler Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, 16, 10, October 1973.

7. Henry M. Levy, *Capability Based Computer Systems*, Digital Press, 1984.

8. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *Communications of the ACM*, July, 1974.

9. *System/370 Principles of Operation*, GA22-7000-9, IBM, 1983.

10. Patent number 4,584,639 (describes the secure factory mechanism).

11. William A. Wulf, Roy Levin, and Samuel P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill Book Company, 1981.