Eighth Edition (December 1990)

This paper was written by Norman Hardy and appeared in a slightly different form in *Operating Systems Review*, September, 1985.

Key Logic
5200 Great America Parkway
Santa Clara, CA 95054-1108
(408) 255-9496

Introduction

This publication provides a description of KeyKOS, a microkernel-based system, at a level sufficient to understand its basic architectural aspects. Thus, a designer of program systems could use the information here to guide alternative design efforts, but a programmer will find insufficient detail to produce code for the system.

Because of the formal aspects of the KeyKOS architecture, it is possible to reason about many characteristics of systems implemented within it in useful ways. The information presented in this publication is sufficient to allow one to begin such reasoning about KeyKOS and programming systems built using its principles.

Other Key Logic publications of a theoretical nature include *KeyKOS Security: Formal Security Model, A KeyKOS Solution to the Confinement Problem* , and *Security in KeyKOS*. They are available from Key Logic upon request.

This publication is presented primarily for those who are interested in reasoning about complex computer programming systems, and for those interested in foundations for such systems. It assumes that readers have a basic knowledge of computer architectures, and some awareness of both objects and capabilities.

Largely because of an attempt at conciseness, but also because of the compactness of KeyKOS and the mutual reliance of the abstractions from which it is developed, certain words and phrases appear, of necessity, earlier in this publication than the principle discussions explaining them. The information in this publication is ordered with the intent that a reader will need to assume as little as possible about as-yet-unexplained topics. Bold references in the index locate their definitions.

KeyKOS is currently implemented on the Motorola 88000, where it supports UNIX, and the IBM System/370 [2], where it supports UNIX, VM/CMS, and a high performance transaction processor. A description of the C language interface to the KeyKOS functions is provided in Key Logic's publication *KeyKOS C Programmer's Reference* (KL113).

A description of the IBM System/370 implementation of KeyKOS at a level sufficient to implement assembly-level programs is provided in Key Logic's publication *KeyKOS/370 Principles of Operation* (KL002). Other Key Logic publications about KeyKOS and its facilities are available at various introductory and programming levels.

If you would like further information about Key Logic or its products, please contact us at (408) 496-1090.

Overview

This publication introduces the architecture of KeyKOS, a capability-based system written in software for implementation on a wide variety of hardware platforms. KeyKOS consists of a microkernel, which executes in privileged-mode, plus additional facilities necessary to support operating systems and applications. The KeyKOS functionality combined with the hardware can be thought of as a "machine" to which one can port operating systems or on which one can write applications. The operating systems which have been ported to KeyKOS include UNIX and, on the System/370, VM/CMS.

This paper attempts to tell enough so that certain arguments and conclusions about the KeyKOS properties can be formed. The description of the *microkernel* functions is essentially complete. Only the more important additional facilities are described.

KeyKOS was originally designed to solve the security, data sharing, pricing, reliability, and extensibility requirements of a commercial computer service in a network environment. By using the microkernel approach to preserve operating system compatibility, KeyKOS functions equally well on a range of hardware, from workstations to mainframes. Because it was developed in a commercial environment its performance consistently meets or exceeds that of the "native" operating systems while remaining semantically identical.

KeyKOS for the System/370 began supporting production applications on an IBM 4341 in January 1983. It has run on Amdahl 470V/8, IBM 3090/200 (in uni-processor System/370 mode), IBM 158, and NAS 8023 processors. KeyKOS also executes on the Omron LUNA/88K.

KeyKOS is in the tradition of message-based systems. **Messages** are the primary interaction between components of the system.

For brevity we use "**key**" where most literature uses "capability".

KeyKOS supports object-style programming. Indeed, nearly all of the code written so far for KeyKOS serves to define some particular type of object. Objects call upon the services of other objects by sending key-addressed messages to request a service, and accepting a key-addressed message in return. This is within the paradigm of "remote procedure calls," and is also similar to the Smalltalk mechanism.

In object-style programming, objects are implemented by some combination of "ordinary programming" and use of more primitive objects. In KeyKOS the microkernel terminates this recursion by implementing several types of primitive objects. The code that defines such objects is in the microkernel.

A pervasive principle in KeyKOS is that a program "module" should obey the "principle of least privilege" [1]. 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 to observe the principle of least privilege.

Objects, keys, and messages

Programs running in capability systems, such as KeyKOS, **hold** some representations of authority called **keys**.

Keys are tokens of authority. A program may only cause **actions** warranted by the keys it holds. Any action that a program causes is enabled by some key that it holds and explicitly identifies in an **invocation** of that key.

A key **designates** a specific **object**. The authority conveyed by a key is specific to just the object designated by that key. Different keys may designate the same object but convey different authority over that object. Exercising authority represented by a key is done by invoking the key. Some objects have state which may be modified by a holder of a key that designates the object. Weaker keys to that object may only let the holder sense the state of the object.

A **slot** holds a key. All keys reside in slots. Just as a register holds different numbers at different times, so may a slot hold different keys at different times.

An invocation involves a message, specified by the invoker and delivered to the object designated by the invoked key. The invoker may include in a message keys and/or data that he holds, whereupon the designated object may also hold them.

In summary: If object A holds a key b to object B then A may request B's service by **invoking** b thus sending a message to B.

Introduction to domains

Domains are the actors of KeyKOS. Any event in KeyKOS is the direct and immediate result of an action by some domain. Domains are primitive objects that **obey** programs and thus assume the role of non-primitive objects.

A domain has sixteen general slots and several special slots. The use of the general slots is determined by the program obeyed by the domain. The keys in the general slots are invoked or included in messages by the explicit direction of the program obeyed by the domain. To say that a program holds a key means only that the key is in some general slot of a domain that obeys the program. The use of the special slots is determined by the domain. Their names and use will be explained below.

As will be seen, domains can act to acquire and hold new keys. Such new keys are said to be **accessible**. Such actions must be warranted by keys already held.

The domain's **address slot** holds the key to its **address segment**, which provides the domain's address space. The domain **obeys** a program found in its address segment. Code designed to be obeyed by a domain is **domain code**.

The domain interprets the program according to the hardware problem state architecture. The general registers, the floating registers and other status are included in the domain's special slots (as degenerate number keys *q.v.*). Similarly included is hardware specific control information. KeyKOS extends many features of the hardware architecture (for example, providing access control via keys), but achieves good performance and necessary functionality by fully exploiting appropriate features of the architecture.

A **gate key** designates a domain. A message addressed via a gate key in an invocation of that key is delivered to the domain so that the program obeyed by the domain has complete control over the disposition and interpretation of the message.

KeyKOS was designed with the idea that a domain will typically spend its life obeying just one program. The function of another program may be had by invoking a start key to a domain that obeys the other program. This other domain is almost always thought of as an object by the programmer of the first domain.

KeyKOS systems are generally built of a large number of small objects. Such modularization increases reliability and simplifies maintenance, since programming errors can impact only the objects in which they occur. Such systems are characterized by a large number of context switches over time. The KeyKOS design and implementation makes these context switches very efficient, and allows application/system builders to exploit their advantages. Because small objects are efficient in KeyKOS they can be utilized in the port of operating systems to significantly in-

crease reliability while minimizing port-time by reducing complexity.

In KeyKOS, the dynamic relationship between calling and called domains is represented by a key first held by the called domain. Most other capability systems (see [3] for a survey and bibliography) represent this relationship in some sort of internal stack that may only be implicitly referenced by calling and returning operations. In KeyKOS, the CALL operation, which is primitive, produces a message that contains an implicitly produced **resume key** to the CALLing domain. The resume key is a form of gate key. To the CALLed domain this is merely another key, conventionally used to return. Alternatively, it may be passed to another domain, stored into an array of keys or anything else that may be done with a key.

Domains are like "tasks" of MVS [4] or Ada [5]. They are like "processes" in UNIX [6] and various other systems.

Major microkernel features

The KeyKOS microkernel is small, runs in privileged state, is unswapped, and runs with address translation off.

A total computing system consists of multiple levels of abstraction, including the hardware, the operating system, the file system, the database management system, etc. The function provided at each level must be carefully selected and implemented or else higher levels may be subject to penalties in performance and/or complexity of design. The KeyKOS microkernel itself does not provide all of the functionality conventionally associated with an operating system; rather it provides a set of primitives which allow such (guest operating system) functions to be implemented by problem mode programs in domains. These KeyKOS machine primitives are much easier and more productive to use than the functions of the hardware alone.

The microkernel interprets keys. No other program has direct access to the bits that represent keys (except Keybits, *q.v.*). Code defining primitive objects is in the microkernel. These primitive objects are tools sufficient to build higher-level objects. The microkernel provides:

- several types of primitive objects;
- multiprogramming support, primitive scheduling and hooks for fancy schedulers running in domains;
- single-level store. Domain programs are unaware of the distinction between main storage and disk;
- virtual memories for domains based upon the address segment using memory mapping hardware;
- redundant disk storage for selected information (to provide reliability and improve read performance);
- a system-wide checkpoint-restart feature;
- special pages exempt from checkpoint;
- gate keys by which messages are sent between domains;
- primitive and limited access to individual I/O devices;
- an invariant interpretation of keys (independent of the location of the designated objects—whether on disk or in main storage).

KeyKOS takes system-wide checkpoints every few minutes to protect from power failures, most microkernel bugs, and detected hardware errors. All data *and processes* are checkpointed. The KeyKOS microkernel keeps no internal state about the data and processes which cannot be re-constructed from the checkpoint information. After an **Initial Domain Load** [7], subsequent IPLing restarts the machine from a checkpoint. The microkernel also provides primitives to support concurrent checkpoints to magnetic tape which comprise a snapshot of the entire system.

Messages

A message is composed of a **parameter word** (commonly interpreted either as an order or method code, or return code), a string of from 0 to 4096 bytes called the **byte string**, and four keys. A domain may compose a message from an integer, from contiguous data in its address segment, and from keys it holds. The message is delivered to an object designated by a key that is held and indicated by the invoking domain. A particular value for the parameter word in conjunction with a particular kind of key is called an **order** on that key.

The microkernel does not buffer messages. Invocation of a gate key is deferred until the recipient domain is ready to accept the message. Message buffering can be implemented transparently by domain code if needed. We choose not to make the microkernel responsible for holding message data, which potentially could be held forever.

Gate keys, when invoked, deliver the message to another domain. A gate key is the authority to send a message to the designated domain. Messages sent via a gate key are delivered to the receiving domain in a way controlled by the receiving domain's program. The program may choose to accept the keys from the message, whereupon its domain will then hold those keys. It may choose to accept the parameter word and all or part of the byte string. The keys and data from the message can then be interpreted according to the logic of the receiving domain's program. The sending and receiving of the message causes the data and keys to be copied from the domain of the sender to the domain of the receiver. Like an event, a message never persists: it is consumed the same instant it is created. In this respect KeyKOS messages are like Smalltalk [8] messages.

Invocation of keys which are not gate keys delivers the message to the microkernel which immediately returns a message in response, according to the logic of some microkernel-implemented object designated by the key.

Domain states and kinds of invocations

In KeyKOS, key invocation plays the traditional role of subroutine call, and the message plays the role of parameters. The traditional calling mechanism sends the address of data in order to communicate requirements to the subroutine. In KeyKOS, the message commonly includes keys that serve the same purpose—to indicate which objects are to be operated upon by the server.

A domain is always in one of only three states: **running**, **available**, and **waiting**.

Running domains will execute instructions, barring page faults, insufficient CPUs, and a few other obstacles. CPUs are multiplexed among the running domains. Available or waiting domains do not execute instructions.

There are two kinds of gate keys: **start keys** and **resume keys**. A start key can deliver a message only when the domain it designates is available. If a running domain tries to invoke a start key to a domain that is not available, the running domain is queued along with other domains invoking start keys to the unavailable domain. They will run again when the designated domain becomes available. A resume key is created as a result of the CALL instruction and is described below.

A program invokes a key by executing an invocation instruction; bits in a register select one of three ways to invoke the key: **FORK**, **CALL**, or **RETURN**.

A FORK invocation leaves the invoking domain in the running state.

A CALL leaves the invoking domain in the waiting state and automatically generates within the message, as the last key, a resume key to the invoking domain. The invoking domain remains in the waiting state until the resume key is

invoked, whereupon the domain returns to the running state. Resume keys exist only to waiting domains, and as a resume key is invoked, all resume keys to the designated domain disappear and are everywhere efficiently replaced by null keys, *q.v.*

A RETURN invocation leaves the invoking domain in the available state. If there are domains that were queued by the unavailability of the RETURNing domain, one is promptly run.

The state of the domain is determined solely by the type of invocation, and not by the key being invoked.

Summary of gate invocations:

Two kinds of gate keys:

- *Start key*: Queues invoker until the designated domain is available.
- *Resume key:* Automatically created by CALL invocation—exists only to waiting domains.

Three ways to invoke a key:

- FORK: Leaves invoking domain running.
- CALL: Leaves invoking domain waiting and includes in the message a resume key to invoker.
- RETURN: Leaves invoking domain available. Dequeues domains on the queue of the domain which performed the RETURN.

The invoking domain's program specifies:

- which key held by the invoker is to be invoked;
- which data and keys are to be included in the message;
- which of the three kinds of invocation is to be performed;
- for CALLs and RETURNs, how to receive the message for the invocation that will next cause the domain to run.

The invoked domain's program (or initialization) has specified:

- how the components of the message are to be accepted;
- where to start execution.

Invoking keys that are not gate keys

Keys that are not gate keys are **primary keys**, and the response to their invocation is performed by the microkernel. When a primary key is CALLed, usually the microkernel immediately returns a message to the invoker (depending on the nature of the invoked key) in such a way that the invoker cannot determine from the behavior of the key that it is not a start key. Most invocations of primary keys are CALLs. In this case the microkernel needs to "return to" the last key parameter. If the invocation is not a CALL and the last key parameter is a resume key then the message produced by the microkernel will be delivered to the domain designated by the resume key. Otherwise the message is lost.

CALLing a start key

This style of invocation, together with a matching RETURN invocation of the resume key, plays the role of the classic subroutine linkage and is the primary method of putting software together in KeyKOS. (Domains may share read-write storage for communication, but seldom do.)

CALLing a start key provides access to a serially reusable resource implemented by a domain's program. Thus the microkernel need not allocate stack frames or cope with storage exhaustion. Most domains have state between invocations. This state is relevant to some context. Other contexts may have their own domains obeying the same program but with another state. The domain thus embodies the Smalltalk-style object. Domains obeying the same program implement objects of the same **type**. Such objects differ only in their state, not in their transition rules. However, KeyKOS objects exist at the system level, not the language level, and are much larger than Smalltalk objects (which average about 50 bytes).

The queuing of invokers of a start key to an unavailable domain provides a convenient and efficient queuing mechanism for the domain representing a serially reusable resource. Some services, such as a compiler, have no state between invocations and should be reentrant. In such cases the start key leads to a domain that creates another domain that obeys the compiler for one compilation and then destroys itself. Several compilations may thus proceed at once.

Co-routines—or CALLing a resume key

If domain X CALLs a resume key to domain Y the situations of the two domains are thereby reversed:

> Before CALLing: X is running and holds a resume key to Y; Y is waiting.
>
> After CALLing: Y is running and holds a resume key to X; X is waiting.

This action constitutes a co-routine linkage and has the traditional attributes of co-routines. This is the method typically used by a domain to deliver a sequence of values to another domain, especially when the values are produced or consumed as they are transmitted.

The co-routine relationship is frequently established as follows: X wishes to deliver a sequence of values to Y to whom X holds a start key. X CALLs Y's start key sending an order in the parameter word foretelling a sequence of values. Y now holds a resume key to X and CALLs it for the first value of the sequence. X now holds a resume key to Y and passes the first value by CALLing the resume key. (Note that the unavailability of Y prevents other users of Y from interfering with Y's state until Y is finished receiving the entire value sequence.)

This proceeds until X or Y indicates that it wishes to terminate the relationship, whereupon Y RETURNs to X's resume key, making Y available for new transactions.

Functions of the domain

The following functional areas are frequently provided separately in other systems but are bundled in the KeyKOS domain:

- Abstraction (information hiding)
- Instantiation
- Protection (calling some program with different authority)
- Exclusion and queuing
- Multi-programming

Pages and nodes

The microkernel provides some fixed number of **pages** and **nodes**. There are typically many thousands of pages and nodes. A page holds 4096 bytes of data and a node has sixteen slots. A node also has an associated **process-running predicate** which is TRUE if and only if the node is the distinguished node of a running domain. Pages and nodes are microkernel-implemented objects and they are accessed by keys. State-bearing objects are ultimately built of pages and nodes. Pages and nodes carry all of the state of the system. Pages and nodes are swapped by the microkernel to provide a single-level store. Pages and nodes are the only primitive microkernel objects provided in large numbers. The checkpoint-restart mechanism and checkpoint to magnetic tape deal only with pages and nodes.

A **node key** designates a node and provides authority to retrieve keys from the node or to place keys into the node. A **fetch key** also designates a node but may be used only to

retrieve keys from the node.

A page may be designated by read-write and read-only **page keys**.

Three special objects

Domains, **segments**, and **meters** are three special primitive objects. By their nature these objects must be implemented in the microkernel. For each of these special objects there are infrequent exceptional states, encountered by the microkernel, whose disposition must be determined by code outside the microkernel. (In KeyKOS, by design, the **policies** to be implemented in these exceptional situations are left to domain code, as in [9].) In such cases the key in the object's **keeper slot** is invoked with a message that includes information about the nature of the state, and the **service key** to the object. The key in the keeper slot is normally a start key to a domain obeying a program called a **keeper**. The keeper may use the service key to return the object to a state where the kernel's rules for the special object are suitable again.

A keeper invocation is one which is triggered by some exceptional state of a domain, a segment, or a meter. This invocation is in the form of a CALL invocation, and is designed such that, after the exceptional states which triggered the event are rectified, control may be transferred to the domain whose action caused the circumstance (by a RETURN invocation of the resume key) without it being aware of the interruption of its execution.

In KeyKOS, the use of keeper invocations to summon aid is motivated by the intent to remove policy concerning the handling of such events from the microkernel. Fixed policies of this kind tend (in conventional systems) to inhibit the evolution of the system. By moving such policy to programs implemented in domain code, an attempt is made to provide an avenue for future evolution and growth of the system, as well as a mechanism to allow multiple policies to be simultaneously supported in a single system.

Domain service

The key from the domain's **keeper slot** is called when the domain encounters difficulty in executing instructions. Program interrupts (beyond address translation faults) and Supervisor Calls (SVCs) cause the domain's keeper to be called. The domain keeper may thus emulate environments provided by other operating systems, or provide debugging services.

The **domain service key** designates a domain. It provides authority to retrieve and replace parts of the state of the domain including the keys in the general and special slots. The domain service key provides complete access to the state of the computation within the domain, and authority to intervene therein.

The code that interprets the domain service key invocation is in the microkernel, in contrast to the code which interprets the domain start key invocation, which is the domain code. The domain code (obeyed by the domain) has no control over and is not involved in interpreting these domain service key invocations. The domain service key is used to initialize a domain and to intervene if the domain's program should not behave as expected. Many domains hold their own domain service key. One order on a domain service key yields a start key to that domain. This order also takes a one byte value, the **data byte**. The start key includes this byte, which is delivered along with messages delivered via that start key.

Segment service

A **segment** has some specific size which is a power of sixteen. A segment is either a page or is **compound**. A compound segment comprises sixteen equal-size portions. To a compound segment there are both a **segment key** and a **segment service key**. Page keys and segment keys are **memory keys** *. The memory key to a compound segment is a node key, fetch key, sense key *(q.v.)* or segment key designating the node that is the root of the segment.

The sixteen portions of a compound segment S are each represented by the key from one of sixteen component slots in S accessible via the segment service key to S. To the segment key holder the segment appears as the seamless concatenation of the sixteen portions. A fetch or store reference to a portion of S via the segment key to S has the effect of a fetch or store reference via the key from the corresponding component slot. These keys are typically memory keys to smaller segments. A fetch or store reference to a portion of a segment whose component slot key is not a memory key is **invalid**. Store references via read-only memory keys are also invalid. Invalid fetch or store references to segments cause invocations of the key from the segment's keeper slot or lacking that, the referencing domain's keeper slot. The referencing domain is left in the state just before the reference was made. The invocation message includes the address within the segment to which the reference occurred, a segment service key, and a resume key to the referencing domain.

If M is a memory key to a segment, then one may construct, given M, the following three types of segments:

- a sub-segment of the original (specified on page boundaries)
- a read-only version of the original
- a version of the original via which it is impossible to signal the segment's keeper

A memory key to a page or segment may be used to define the address segment of a domain or it may be used to define a portion of another segment. The same memory key may be used in several contexts at once.

Meter service

Meters account for certain system resources including CPU time. To a meter there is a **meter key** and a **meter service key**. For a domain to run, its **meter slot** must hold a meter key to a valid superior meter.

Within a meter appear:

- counters of the resources consumed by domains dependent on the meter;
- a slot for a superior meter key;
- a meter keeper slot.

For a meter to be valid, its superior meter key slot must hold a meter key to a valid higher meter. There is a primitive meter which is always valid. There must be a chain of meters leading to the primitive meter. As a domain runs, the chain of meters rooted in the domain's meter slot all record the resources used. The meter service key supports fetching from and storing to the various slots of the meter while the meter key provides (but limits) the resources the holder can consume. The meter key holder can build inferior meters.

A meter's keeper slot key is called when a resource counter in the meter reaches zero. The message associated with the invocation includes a meter service key which may be used to replenish the counter if that is the plan. The message also includes a resume key to the waiting domain whose execution exhausted the meter. This key may be used to restart the waiting domain when appropriate. The meter keeper is in a position to perform scheduling (directly or indirectly) over all the domains served by that meter.

Role of nodes in the special objects

Nodes are of very general use. They play much the same

* While a page is a segment, a *page key* is not a *segment key*. A *page key* is, however, a *memory key*.

role of control blocks of some programming systems. Nodes assume a fundamental role in the definition of most other objects. The special objects (domains, segments, and meters) are actually composed of nodes. Their behavior is determined by code in the microkernel. It might be said that these extra behaviors of the node are merely alternate personalities of the node, inherent in the node itself. Segment keys, domain service keys, and meter keys designate nodes while limiting their holders to the corresponding special function. A segment service key or meter service key is merely a node key to the underlying node. We call them all service keys here merely to emphasize their parallel role. Similarly, the meter service key is the node key to the underlying node.

To build a segment or meter it suffices merely to place the appropriate keys in a new node, and request the segment key or meter key by an order on the node key. To limit the number of programs that know how domains are built from nodes, getting a domain service key takes a special key: the domain tool key *q.v.*. The domain service key is used to install the parts of a domain. Typically the domain service key is obtained from a **domain creator**, which is the only type of object holding the domain tool key.

The effects of invoking primitive keys are immediate. Invoking a node key to change the definition of a segment has immediate effects on all address spaces in which that segment occurs.

The reader may have noticed by now that some keys have other uses than invocation. In particular, memory keys are used in the address slots of domains and meter keys are used in the meter slots of domains.

Creating an object

The domain is the microkernel primitive used to implement new kinds of objects. Upon invocation of a gate key, the message is delivered to and interpreted by code which is obeyed by that domain.

To create an object: Put the code to instruct the object in a segment (perhaps just a page); request a new domain and accept the domain service key. With the domain service key:

- install the segment as the domain's address segment;
- set the domain's initial PSW;
- install a meter key in the domain's meter slot;
- install a domain keeper (if your program may fault);
- order a start key (from the domain service key).

Now the start key may be passed to intended users of the object.

The program segment of a domain is typically read-only if several domains are to obey the program. If the program has store instructions a write-able address segment must be provided for the domain. One may do this by providing, for each domain, a private segment the domain can store into and composing the domain's address segment from that segment and the shared read-only program segment. Alternatively, one may create a "copy on write" segment keeper which will share all pages which are only read, but create write-able private pages upon stores.

Many simple objects can be implemented in assembler so as to require no private pages. Such programs have no store instructions as the domain's address segment consists of just read-only pages (frequently, only one). Objects of this simple type all share the same read-only segment. Such objects typically occupy from 200 to 700 bytes of individually dedicated disk storage.

Programmers usually work at a higher level and these segment planning details are handled automatically.

In contrast to the Smalltalk object [8], there are no "variables" local to the start key invocation. There are, however, variables that keep their values between invocations (like "own" variables in Algol 60).

A start key to a domain conveys no authority to examine the state of the domain or to examine the domain code. In a computer supporting two non-privileged instruction sets, the set employed by a given domain would not be visible, given start keys to the domain.

What KeyKOS domains have

KeyKOS domains provide environments in which programs may execute. They contain some ordinary process state information and some state information unique to KeyKOS.

Slots of the Domain:

- data for most of the real hardware program status
- general and floating point registers' values
- address slot
- Domain keeper slot
- Meter slot
- sixteen general slots to be referenced by the domain code
- trap information produced by the hardware or kernel indicating if and why the program can't run
- hardware specific values such as debugging aids.
- a brand (*q.v.)* unique to its creator

What KeyKOS domains do not have

Here are a few features that analogs of domains in other systems frequently have that KeyKOS domains lack:

- *Terminal*: The domain has no special terminal with which it can converse (unless one of its accessible keys provides that).
- *Directory*: The domain has no special authority to access some directory by virtue of the fact that it was created by the directory's owner or by any other virtue except holding a key to the directory.
- *Priority*: The domain has no special scheduling or billing properties beyond those stemming from its meter key.
- *Address Space*: The domain has no special affinity to an "address space" beyond that explicitly designated in the domain's address slot.
- *Stack*: There is no "caller of this domain" implicitly known by the system that can be "referenced" only by returning to it.
- *System Call*: There are no System Calls other than:

  1) those interpreted by the microkernel (key invocations CALL, RETURN and FORK) and,

  2) those interpreted by the domain's keeper (all others).
- *Debugging*: There are no special provisions for intervening in a broken domain beyond that provided by the domain's keeper or some other holder of the domain service key. No key—no access.

Other primitive microkernel objects

Besides pages and nodes, the microkernel implements the following miscellaneous objects:

- **Node range object:** Controls a fixed set of nodes. An order on a node range object key will provide a node key to any of the nodes it controls. The node range object will also efficiently destroy all keys to a specified node that it controls. **Page range objects** do the same for pages.

- **Number keys:** Designate numbers and are convenient for keeping a few bytes of data in a slot. Invoking a number key yields its number. The **null key** is merely the number key that designates zero. The **number key creator** creates any particular number key. Number keys are described as designating numbers for the formal convenience of saying that every key designates an object. In fact the number (data) is in the key.
- **Wait objects:** Will return a message at some future specified time. That time is an internal state of the wait object. There are just a few primitive wait objects; one of them is multiplexed by domain code to produce a large number of non-primitive wait objects.
- **Device allocation object:** Produces and rescinds **device keys** that designate and control a particular I/O device. The microkernel does its own I/O to the disks that it owns which hold pages and nodes. I/O to other devices is done by the kernel in response to calls to device keys that designate the device.

Microkernel code also defines these individual, miscellaneous one-of-a-kind objects:

- **Keybits**: An object that provides the bits that are characteristic of a provided key. Keybits returns the same bits for the same key and different bits for different keys. The sole current use of Keybits is to provide for a sorted list of keys.
- **Peek**: A very powerful object that displays the real storage of the system. Peek is closely held by domains used for system debugging.
- **Domain tool**: An object that will produce a domain service key to a node given a node key to that node. Given a gate key or domain service key designating a node and a key that matches the brand of that domain, the domain tool will return a node key to that node. The **brand** of a domain is found in its **brand slot**. A node key to the domain root is required to install the brand. Presumably only the creator of the domain has the brand. This is the primitive KeyKOS **rights amplification** mechanism.
- **Discrim**: Compares two keys for equality.
- **Returner**: An object that merely returns any message sent to it.

Sensory keys

A universal paradigm in software design is the linking together of data structures with pointers. This idea is carried over into capability systems by using keys as pointers. Consider a tree of nodes with pages at the leaves. The nodes hold node keys and page keys. To our knowledge, in previous capability-based systems a node key to the root node allowed the holder to peruse and modify the collection. There was no "read-only" key to the structure. A fetch key to the root node is too strong because it may be used to fetch a node key to a lower node which, in turn, may be used to change lower nodes of the tree.

To solve this problem KeyKOS provides **sense keys** to nodes. A sense key is weaker than a fetch key. A sense key appears to the holder as a fetch key except that any key delivered to the holder as a result of an invocation of a sense key is the **sensory** version of the key being fetched.

The sensory version of a node, sense, or fetch key to node N is the sense key to node N. The sensory version of a segment key is a segment key without authority to invoke a segment keeper. The sensory version of a page key is the read-only key to the same page.

If K is a number key or a key to one of the following objects then the sensory version of K is K:

- Discrim
- the Number Key Creator
- the Returner

The sensory version of other keys is the null key.

The important thing about a sense key is that it conveys no authority to influence a structure. The holder of the sense key can browse through the structure but is unable to affect it, nor may a non-sensory key be obtained using a sense key. If the structure holds page keys (a common situation) then those pages will be available for reading but not writing.

Fundamental objects implemented by domains

The remainder of this document is a sketch of the major fundamental software outside the microkernel. These are the major features of the KeyKOS programming environment. There are now more than 150 types of ready-made objects available to the KeyKOS programmer. Up to this point adjectives used to describe keys have depicted key classes of significance to the microkernel. Hereafter we differentiate the types of start keys that designate domains that obey different code.

A **bank** holds keys to node and page range objects. The bank keeps track of which pages and nodes are in use. The holder of a key to a bank can buy a node. The bank selects a currently unused node, creates a node key to it, records that the node is in-use, places null keys in each of the node's slots, and returns the node key to the requestor, thus providing a key to a "newly created node". As the bank invocation finishes, the bank and the requestor hold the only copies of any key to that node. New pages from the bank are similarly acquired, and are zero-filled. The bank holds the only node range object key that controls the space from which the node was created. Another order on a bank will sell the node back (reclaim the storage). Just after such an invocation all slots with keys to that node are efficiently filled with null keys. There is also an invocation on the bank to reclaim all of the pages and nodes ever bought from that bank.

Banks also provide for measuring and limiting the degree of storage use. New banks may be created inferior to a given bank. It is as if the inferior bank bought its material from its superior. The inferior bank may have its own limits. Inferior banks may have their own inferior banks, etc.

A domain creator will accept a bank key and build a new domain out of nodes from that bank. It will return a domain service key to the new domain. A domain fresh from a domain creator has no address segment, meter or domain keeper. They must be installed by use of the domain service key. Each domain creator holds a unique key which it installs in the domain's **brand slot**. Another order on a domain creator takes a gate key to a domain as a parameter and returns the domain service key to that domain if the domain was one that this creator created (as determined by the brand).

A **factory** [10] is an object that initiates a **compartment** in which a computation may take place. A compartment is a collection of objects that collectively hold no unaudited, non-sensory keys that designate objects outside the compartment. Compartments produced under such circumstances have measured **discreetness**. A factory can certify the degree of discretion of another factory. If one trusts the (fixed) logic of a factory, one need not trust the logic of the programs obeyed by domains within the compartment not to disclose one's data. Factories allow the construction of programs which "keep secrets", and address containment

and mutually suspicious user security problems [11].

A top secret user might have a discreet compartment created for him by a factory to house his workspace. His work will go unobserved by users in other compartments.

Factories can produce segments with keepers that support the illusion of initially zero segments. Real storage is acquired as pages are read from or stored into for the first time.

A significant object that has been implemented using sense keys is the **virtual copy segment**. Such a segment will produce, on demand, a source of segment copies whose initial state is that of the original at the instant of the demand. Such a copy is modifiable but such modification is insensible except by the key to the copy. The cost of the copy is proportional to the number of modified pages in the copy. Such segments are said to be **discreet**. Factories can certify that this source of new segments produces only discreet segments. The virtual copy functionality is available in some operating systems and is generally known as "copy on write". The segment keeper function of KeyKOS allows one to implement this function in unprivileged code.

Objects called **record collections** serve as directories and indexed files in KeyKOS. Record collections provide the services other systems achieve by the combination of files and access methods; for example they are used to provide UNIX directories, and, on the System/370 the functionality of IBM VSAM data sets.

A **symbolic machine language debugger** is in the tradition of the MIT DDTs. The debugger runs as a domain keeper and is thus safe from the flailing of a sick program. The debugger can be connected to any terminal for which appropriate keys are available. For all but a few basic KeyKOS system domains this debugger is available without "preplanning" and with no cost until used.

Programming facilities

Knowing a password to a KeyKOS system allows the user to connect a terminal to some object created for that user. Commonly this object is a switch that lets the terminal talk to one of a set of objects.

An individual who programs KeyKOS or understands keys has a command system (also an object) that holds the key to his **directory**. A directory associates names with keys, and provides a key name space of the immediately executed commands expressed in the command language. There are normally a number of command systems and directories available to a user.

The command system is the only program that has "natural" access to the directory. If the user wants a program to run with access to some of his data, the user must invoke the start key to the domain containing the program, passing the key to the data. Such a program lacks access to the user's directory unless a key to the directory is explicitly passed.

Keepers can emulate the behavior of operating systems. Operating systems that have been emulated include BSD4.3, Minix, VM/CP (S/370), a reduced MVS (S/370), and EDX (Series/1). These emulations provide binary compatibility at the application level.

C and PL/I source are expanded (through preprocessor and runtime library routines) to produce programs for KeyKOS that explicitly manipulate keys. The compiler is bundled into an object that reads source programs and creates a factory which produces objects that obey the compiled program. Keys are named symbolically and the program is relieved of the sixteen key limit for held keys.

Since all KeyKOS functionality is packaged with keys, and the key invocation facility is general, a program can use any function authorized to it by a key without resorting to the assembly code necessary in older systems. The KeyKOS facility provides extensive sharing: all objects written in a given language share the same library, and objects from the same factory share the same compiled code. Data segments may be flexibly shared by programs written in high level languages.

Transaction processing foundation

KeyTXF, a transaction processing foundation program product, provides locking, commit/abort, journaling, monitoring, and similar function in domains appropriate for the development and operation of transaction processing applications.

Operator functions

The KeyKOS microkernel has no built-in operator interface. Such function, such as a tape operator interface, is implemented in domains.

References

[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, pages 409-445.).

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

[3] Henry M. Levy, *Capability Based Computer Systems,* Digital Press, 1984.

[4] *OS/VS2 MVS Supervisor Services and Macro Instructions,* GC28-1114-1, IBM, 1983.

[5] *Reference Manual for the Ada Programming Language,* United States Department of Defense, ANSI/MIL-STD-1815A-1983, 1983.

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

[7] *KeyKOS /370 Principles of Operation,* KL002, Key Logic, 1988.

[8] Adele Goldberg and David Robson, *Smalltalk–80 The Language and its Implementation.,* Addison Wesley, 1983.

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

[10] U.S. Patent number 4,584,639.

[11] Butler Lampson, "A Note on the Confinement Problem", *Communications of the ACM*, V 16, N 10, October, 1973.

[12] *Virtual Machine/System Product CMS Command and Macro Reference,* SC19-6209-1, IBM, 1983.

[13] Huberman , B. A. et al., *The Ecology of Computation,* North Holland, 1988.

[14] Hardy, Norm., "The Confused Deputy", *Operating System Review, Oct. 1988 vol. 22 #4, pp 36:38*