KSOS—The design of a secure operating system*

by E. J. McCAULEY and P. J. DRONGOWSKI
Ford Aerospace and Communications Corporation
Palo Alto, California

INTRODUCTION

This paper discusses the design of the Department of Defense (DoD) Kernelized Secure Operating System (KSOS, formerly called Secure UNIX).** KSOS is intended to provide a provably secure operating system for larger minicomputers. KSOS will provide a system call interface closely compatible with the UNIX operating system. The initial implementation of KSOS will be on a Digital Equipment Corporation PDP-11/70 computer system. A group from Honeywell is also proceeding with an implementation for a modified version of the Honeywell Level 6 computer system.

KSOS will be capable of handling information at various security levels (a security level is a combination of a hierarchically-ordered classification category, like SECRET or TOP SECRET, and a possibly null set of compartments, like "No Foreign Dissemination" or specialized need-to-know compartments). The goal of the system is to provide strong assurances that it is impossible for an unprivileged user to cause an information compromise.

At its outer interface, KSOS will appear to be closely similar to the UNIX operating system.[13] The only changes are to tighten the security checking on some of the operating system calls, and to add several new calls which individual UNIX sites had previously added to their systems. Existing applications programs written for UNIX will run without modification or recompilation on KSOS, providing that they do not violate the security rules of the system. At last count there were several hundred application programs for UNIX, ranging from simple utilities through sophisticated compilers, data management systems, text processing systems, and powerful editors. (This paper was completely prepared on a UNIX system, as is all documentation for the KSOS project.) All of these programs should run on KSOS without modification.

This UNIX-like interface is provided by a software component called the UNIX Emulator. The UNIX Emulator transforms the user's UNIX operating system calls into (sequences of) calls to the Security Kernel. The Security Kernel is the heart of the system. The Kernel implements the reference monitor concept.[1] Briefly, through a combination of hardware and software checking, the Kernel monitors every access attempt by each user process. The Kernel will be shown to make the correct decision on whether to permit or deny the access attempt.

One important distinguishing characteristic of KSOS over the prototypes which have preceded it[5,8] is that it contains a full range of support software. Included in this "Non-Kernel System Software" (also called Non-Kernel Security-Related Software) are components which support the day-to-day operational functions of the system: secure spooling of line printer output, portions of the interface to a packet-switched computer network, etc. Also included are components for the continuing maintenance of the system such as consistency checks of the file system, and system generation support. Finally, there are components to support the administration of the system, such as adding and deleting users, changing the security levels that a given user may access, and other functions.

The schedule for KSOS calls for its delivery in the fall of 1979 after the conclusion of a full series of testing. The KSOS development contract specifies that the system shall have a full MIL SPEC documentation package. The primary documents defining KSOS are detailed "design to" specifications which are called "B5 Specifications."[3,6,9] The Kernel B5 Specifications[6] include formal, mathematical descriptions of the Kernel written in a language developed by SRI International called SPECIAL.[15] SPECIAL is a formal, non-procedural language for describing the behavior of systems in the manner suggested by Parnas.[10] In addition, technical reports have been delivered detailing our plans for verification of the system's security properties,[16] for the tools and techniques to be used in implementation,[4] and for the long term maintenance and support of the system.[7]

The remainder of this paper begins with a discussion of the influences on the design. As with any design project, it is impossible to identify all of the factors which cause a given course to be taken, so only the strongest influences are discussed. Next the design itself is presented. Here the emphasis is on the more novel aspects of the design. In addition to the usual things expected from an operating

* The work described in this paper was performed under ARPA Order 3319, Contract MDA903-77-C-0333 administered by the Defense Supply Service Washington. Various DoD Agencies are funding the work. The conclusions presented are those of the author and are not necessarily those of the Government or Ford Aerospace.
** UNIX and PWB/UNIX are trademarks of the Bell System.

system. KSOS provides a number of features that aid in the creation of encapsulated secure environments. The paper concludes with a few remarks on how KSOS may be used effectively.

INFLUENCES ON THE DESIGN

External design goals

The overall design goals for KSOS are:

1. The system must provide provable security, i.e. its design and mechanization must be oriented towards the proof of its security properties.
2. The emulation of the UNIX system call interface must be as faithful as possible given the constraints of the security model.
3. The performance of the system should be "good," specifically, the performance should be comparable to that of a UNIX system.
4. The Kernel should be usable by itself as a simple, secure operating system.
5. The design should be amenable to implementation on other hardware bases.

The need for provable security had the most profound impact on the design. First, it dictated the basic structure of the system. A Security Kernel would function as a reference monitor.[1] The Kernel would mediate all access attempts in the system. Because the Kernel would potentially be proven to operate correctly, its behavior would have to be formally specified. Further, the size of the Kernel would have to be kept to a minimum to make formal specification and eventual verification tractable. Although only representative code proofs were planned, the Kernel would have to be implemented in a language suitable for code proofs.

Because the UNIX call interface had to be emulated faithfully and efficiently, the Kernel interface became "UNIX-flavored." However, because non-UNIX applications of the Kernel were planned, there was strong pressure to keep UNIX-specific structures out of the Kernel. As will be seen below, the Kernel has no knowledge of the format, or semantics of UNIX-specific constructs such as directories or load modules (UNIX a.out files). This knowledge is encapsulated outside the Kernel.

It was recognized that a large class of KSOS applications would not require the flexibility and added power of the UNIX interface. Rather, many of them would be built directly on the Kernel. Thus, the Kernel had to provide all of the features commonly found in an operating system. This meant that the Kernel would include somewhat more functionality than the absolute minimum.

Hardware limitations

Although KSOS was intended to be a machine-independent design, it will be implemented on real machines with various hardware limitations. The PDP-11/70 has two significant limitations. First, process switching is expensive because a large number of processor and memory management registers must be individually saved and restored. Thus, architectures which require extensive process switching are to be avoided.

The PDP-11/70 does not lend itself to the creation of virtual machine environments that include direct control of single user i/o devices. The problem stems from the granularity of the virtual address to real address mapping, and from the logical addressing of i/o registers. In KSOS on the PDP-11/70, all devices are managed by the Kernel: no attempt is made to provide devices in the user's "virtual machine."

In fairness to the PDP-11 design it should be remarked that none of these hardware limitations are especially burdensome; they merely influence the design to take advantage of the strengths, and to avoid the weaknesses of the hardware base.

The design methodology

The design of KSOS is strongly influenced by the design methodology used on the project. KSOS is being designed and implemented using a blend of the "classical" methods with the formalism of the Hierarchical Development Methodology (HDM)[14] developed by SRI International. HDM emphasizes formalism throughout the project. The system's security requirements are formally stated as properties to be satisfied by an abstract description of the design. This design is described in a mathematical, non-procedural language, SPECIAL.[15] The security properties of the design are established by proving theorems that are derived from the design and the mathematical model of the security requirements. The implementation language is selected to allow its correspondence with the specifications to be proven. All of these steps force the designer to be precise and exacting in the statement of the system design. They make "kludges" very obvious at an early date. The design methodology strongly encourages a hierarchical decomposition of the design.

KSOS DESIGN

KSOS is composed of three components:

1. The Security Kernel
2. The UNIX Emulator
3. The Non-Kernel System Software

The relationship of these components is shown in Figure 1.

The Security Kernel's function is to provide a simple operating system which can be shown to be secure. The Kernel centralizes the control of all the resources in the system. It mediates each access attempt by a user process and only permits those accesses which comply with the access control policy. The Kernel resides in the most privileged address space of the machine (called "kernel mode"

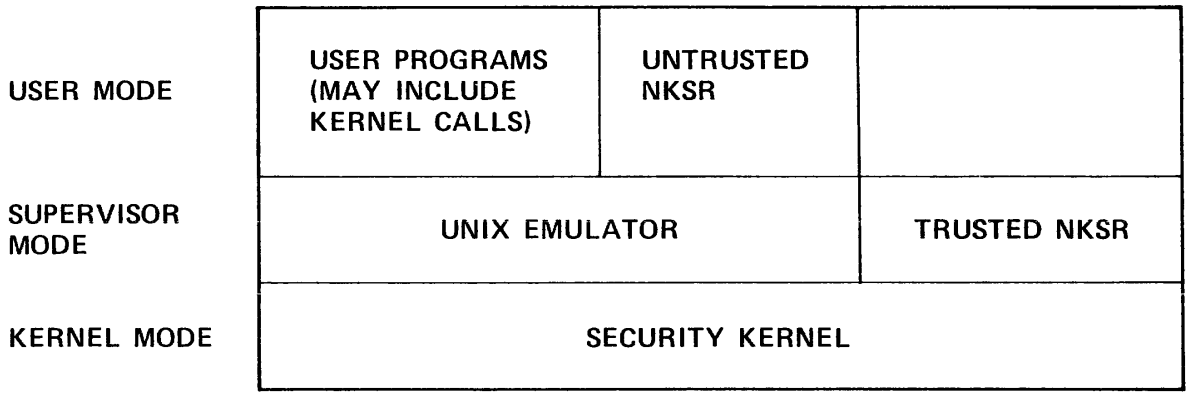

Figure 1—KSOS system structure.

on the PDP-11/70) where it has access to all of the raw hardware and memory management facilities.

Logically, the UNIX Emulator is a part of each UNIX process which on the PDP-11/70 resides in the "supervisor mode" address space of the process. Its function is to map the user's UNIX system calls into the corresponding Kernel call(s).

The Non-Kernel System Software is a collection of autonomous processes performing support services for the system. Like UNIX, KSOS does not have services like login embedded in the operating system. Rather, these services are performed by "trusted processes" which reside outside of the Kernel. Except for the fact that these processes have the privilege to selectively violate the rules of the Kernel, they are just like any other process. Because the Emulator is "untrusted" and is not intended to be verified, it cannot be used by trusted software; rather, such software must use the Kernel directly.

The KSOS Security Kernel

Viewed as an abstract machine, the Kernel's function is to create the objects of its interface (processes, process segments, files, devices, and subtypes) from the basic hardware resources of the system, and to mediate all access attempts to these objects.

The Kernel enforces three distinct types of access checking. The first is the enforcement of DoD security policy. This checking is the verification of that fact that the user has the proper clearance and need-to-know for reading the information (the "simple security property"), and that information cannot be downgraded by writing it to a file at a lower security level (the "security *-property").

The second type is the enforcement of an integrity policy described in Reference 2. Integrity is a mechanism for protecting system data bases, programs, etc. against modification while allowing them to be read by any process. It is formally defined to be the mathematical dual of the security model. We have found this integrity model to be overly restrictive, as its originator suspected. However, it does provide an additional, essential dimension of protection. Development of a more effective integrity model would seem to be a meaningful research topic.

The third type of access checking performed by the Kernel is discretionary access checking. Unlike the first two types of checking, the discretionary access checking is completely under the control of the user. The user may, at his discretion, permit or deny access by other users to the objects he owns. KSOS enforces a discretionary access policy similar to that of UNIX. For each object there are (logically) nine bits that specify read, write, and execute/search access by the owner, others in the same group as the object, and all others. We recognize that this discretionary access policy has limitations when compared to more sophisticated schemes, such as the access control lists used in Multics. However, it is simple, and requires a small fraction of the support mechanisms needed for access control lists.

The Kernel supports five different types of objects:

1. Processes
2. Process segments
3. Files
4. Devices
5. File subtypes

All Kernel objects have the same type of name called a SEID (Secure Entity IDentifier). Further, every object, regardless of its type, has a block of information associated with it that includes all the information needed by the Kernel to mediate access attempts to the object. This block is called the "type independent" information. Because objects, regardless of the object type, have homogeneous type independent information, access checking by the Kernel is greatly simplified. All that must be checked is that information may flow from the source to the destination. For example, if a process wishes to read a file, the source is the file and the destination is the process. In the KSOS Kernel,

two functions perform all the access checking (one for security and integrity checking and one for discretionary access checking).

Processes

Processes are the only active agents in the KSOS design. To adequately emulate UNIX, KSOS processes must be cheap and plentiful. For example, each UNIX command is run as a separate process. Processes in KSOS will require only modest amounts of Kernel resources. Most of the Kernel data for a process will be swapped in and out with the process, reducing the amount of locked down Kernel memory space for the process tables.

Processes may possess privileges ("trusted processes") that enable them to perform functions that require reduced checking by the Kernel (e.g. changing the classification of a file) or which may require that additional checking be performed in the process (e.g. logically mounting part of the file system). The privileges that may be given to a process have been designed following the concept of "least privilege." That is, the granularity of the privileges is quite fine, and quite specific. Many service processes possess only a single privilege, and many privileges are possessed by only one process. Thus, the KSOS Kernel is designed to create encapsulated environments for critical functions. Privileges are obtained from the process image file (load module) from which the process was initialized. Two Kernel calls, K_invoke and K_spawn, are used for the controlled invocation of privileged software. K_invoke functions by replacing the entire process with a user-specified intermediary process. For the invocation of trusted software, this intermediary is a trusted "bootstrap" that, in turn, replaces itself with the requested process image file, and sets the privileges of the process from the values in the image file. K_spawn performs the same function in a new process created as part of the K_spawn function. This mechanism allows knowlege of the format and semantics of process image files to be kept out of the Kernel. Thus, the bootstrap encapsulates the function of initiating trusted software with minimal Kernel support.

In addition to the K_spawn mechanism, new processes may be created by the K_fork call, which is similar to the UNIX fork call. K_fork creates a "clone" of the caller, a new process that is an exact copy of the caller. The only difference between the two processes (parent and child) is the return value from the K_fork call. Such a mechanism is required for the accurate emulation of the UNIX fork call.

Processes normally run at a single security level. The only exception to this is the part of the Non-Kernel System Software that changes the user's working security level. For inherently multi-level applications, the preferred design would be to create a trusted multiplex/demultiplex ("mux/demux") process which directs commands and i/o to processes running at each level needed. This would be preferable to having these per-level functions performed within one process which changes its level because such a process would be larger and more complicated than the mux/demux process. Verification of the correctness of a process becomes significantly more difficult as the process size and complexity increase. One example of this preferred architecture is the KSOS network interface. A small trusted process separates the multi-level data stream from the network into several streams. Each stream has data of only one security level in it. The mono-level streams from the processes are similarly combined by the trusted process into a single, multi-level stream.

Standard UNIX is acknowledged to be deficient in the area of Inter-Process Communication (IPC). KSOS provides significant improvements in this area. The Kernel supports both an event IPC mechanism and shared segments. The event mechanism allows one process to send a message to another process, and (optionally) to cause the receiving process to be interrupted analogously to receiving a hardware interrupt. The full set to security checks is performed for each IPC attempt. That is, information must be able to flow from the sender to the recipient, and the recipient must have permitted such information flow. Finally, a process may enable and disable the pseudo interrupt mechanism, so that it will not be interrupted during some critical operation. (Shared segment IPC is discussed below.)

Process segments

A process segment is a portion of the virtual address space of a process. The process segment is not tied to the native memory management hardware of a particular machine. The KSOS process segment may be of any size from a hardware-limited lower bound up to the entire virtual address space of a process. A process may have only some of its segments actually mapped into its address space. At its creation the segment may be declared to be sharable, in which case other processes can "rendezvous" with it and map it into their address spaces. This allows for very high bandwidth communication between the processes. Naturally, they must establish a protocol that guarantees that the segment will not be corrupted through unsequenced use. The process may elect to have only some of its segments actually mapped into its address space. In particular, several segments for the same part of the address space could exist. This mechanism is used by the trusted mux/demux processes discussed above. The data segments are shared between the trusted mux/demux and the processes servicing each logical stream. The mux/demux maps in a particular segment to a well known location and puts/extracts the data for that stream into/out of the segment.

One other use for shared segments is shared text (program) segments. It is possible to have a pure text segment shared between multiple processes, thus reducing the overall memory requirements for the system. KSOS allows a segment to be locked in memory, or to be retained in the swap area for faster accessing. The designer of the KSOS-based system is offered considerable latitude in trading space for time.

Files and devices

The Kernel file structure is flat and uniform. That is, there are no Kernel assumptions about the internal structure or contents of files. Directories and other higher-level constructs are mechanized outside the Kernel. The UNIX Emulator creates UNIX-like directories by interpreting the contents of Kernel files. This allows a designer working directly with the Kernel to create a different type of directory structure if desired. Kernel files are accessed by blocks. There is no Kernel buffering of file i/o. Rather, the i/o is done directly into the requesting user's address space. Kernel i/o is asynchronous, that is, the call returns to the user as soon as the i/o has been internally queued. An IPC message is sent to the user upon i/o completion. (The inclusion of asynchronous i/o is a relatively late addition to the KSOS design.)

Kernel devices are like a special type of file, as in UNIX. Terminals have only the lowest level echoing support in the Kernel. Higher level functions like erase/kill processing are done outside the Kernel.

KSOS supports removable file volumes. The mechanism is similar to the UNIX mount mechanism with some significant additions for protection. Because of the possibility for removing a volume, files are limited in size to one volume. Presently the design allows for support of at least 300 Mbyte disks, with extensibility to 600 and 1200 Mbyte disks possible. These large disks may be partitioned into one or more extents, referred to as "mini-disks" which may be independently utilized as virtual disks.

Subtypes

The KSOS subtype mechanism is one of its more novel features. The subtype mechanism is designed to allow the selective encapsulation of a class of files. Each file is a member of a subtype class. For example, "normal" files are in the null subtype class. Files which are UNIX directories are the "UNIX directory" subtype class. The accesses to files in a given subtype class may be restricted. The subtype restriction on UNIX directories is that anyone may read a directory, but only a process whose effective user ID is the Directory Manager may write them. These subtype restrictions are in addition to the other types of access checking (security, integrity and discretionary). The access restrictions for a given subtype apply to all files of that subtype.

There are many other possibilities for using subtypes. For example, they could allow "peaceful coexistence" of two separate directory structures as might occur if there were two different Emulators, say one for UNIX and one for another operating system. Subtypes could also be used to control what could be done to files that mechanized the internal structure of a data base management system. Only processes that were known to correctly manipulate the structure would be allowed to change it. The subtype mechanism provides the KSOS Kernel with a significant type extension feature in that it lets the Kernel support encapsulation and control of objects without having the Kernel be cognizant of the syntax and semantics of the object.

Secure terminal interface

In the secure system it is necessary to have an "unspoofable" path to trusted services. ("Spoofing" occurs when an unprivileged user process pretends to be a privileged process. For example, a nefarious user starts a process that imitates the login sequence, and waits for an unsuspecting victim to type in his password.) In KSOS each terminal is (logically) two devices, the normal terminal device and the secure device. Only privileged Non-Kernel System Software is able to use the secure device. When the user types a reserved attention character (currently BREAK), the normal path is blocked, and the character stream is switched to the secure path. Listening on the secure path is a service process which will cause the desired secure service to be performed. Because the normal path is blocked, rather than killing off any process using it, it is possible for the user to start doing something, temporarily abandon it while requesting some secure service, and resume the activity after the secure service is completed. This is the mechanism by which the user is able to change his working security level. The Secure Terminal Interface is illustrated in Figure 2.

Auditing

DoD security policy requires that certain security-related events be captured for auditing purposes. In KSOS this occurs in two ways, as shown in Figure 3. In the first case, the Kernel captures the events it knows about and generates an IPC message to the Audit Capture process. The second mechanism is that the Non-Kernel System Software captures the event. This second case is necessary because the Kernel cannot tell that certain significant events, like a user login, have occurred. The Audit Capture process does only a minimal amount of processing and then simply places the event record into an audit log. Although it is not within the scope of the current KSOS contract, this audit log could be processed to look for suspicious (sequences of) events.

The UNIX Emulator

The UNIX Emulator is almost completely definedby its two interfaces. It must transform the system calls of the UNIX interface into sequences of Kernel calls. In the design of KSOS a serious attempt was made to get a good "impedance match" between the Emulator and the Kernel, while not having the Kernel be strongly UNIX-dependent.

Our view of the Emulator has evolved significantly. Initially, the Emulator was viewed as not much more than a set of subroutines that resided in a different address space. The functions performed by the Emulator were isolated to one process, except for the obvious cases of interaction with

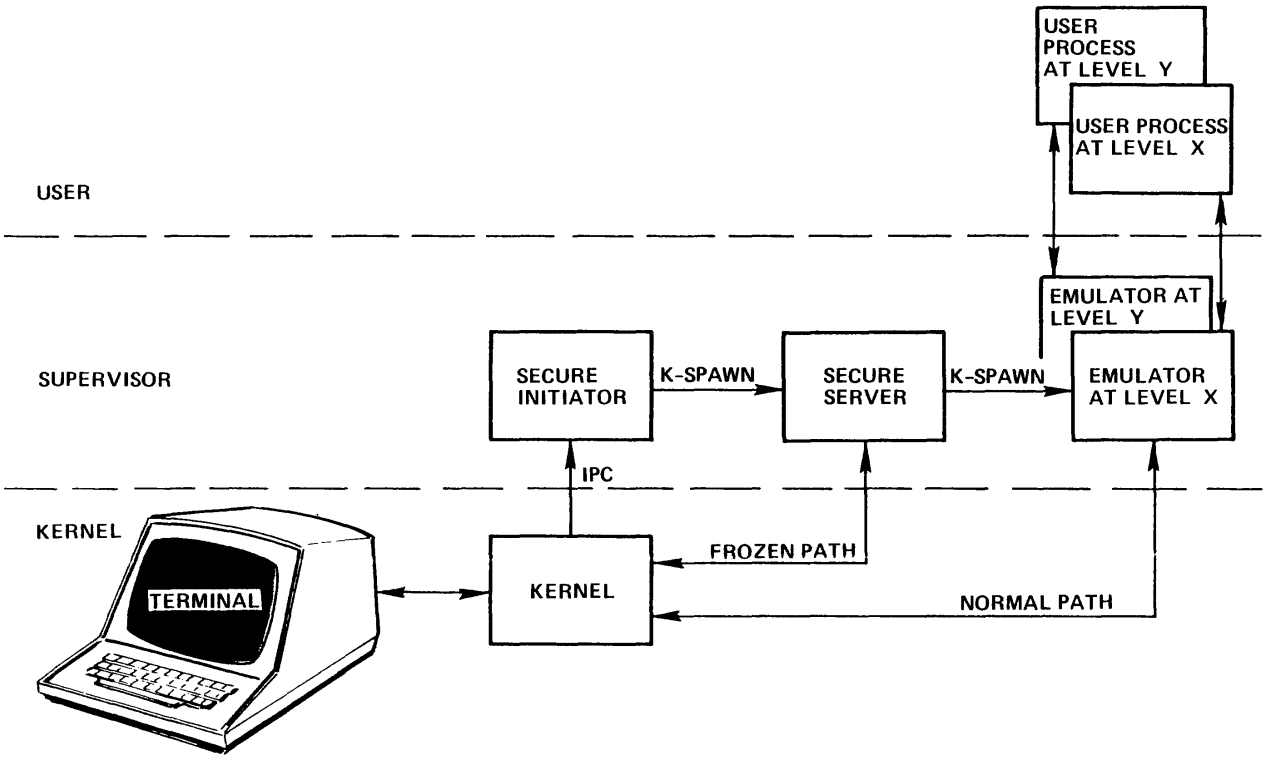

Figure 2—Secure terminal interface.

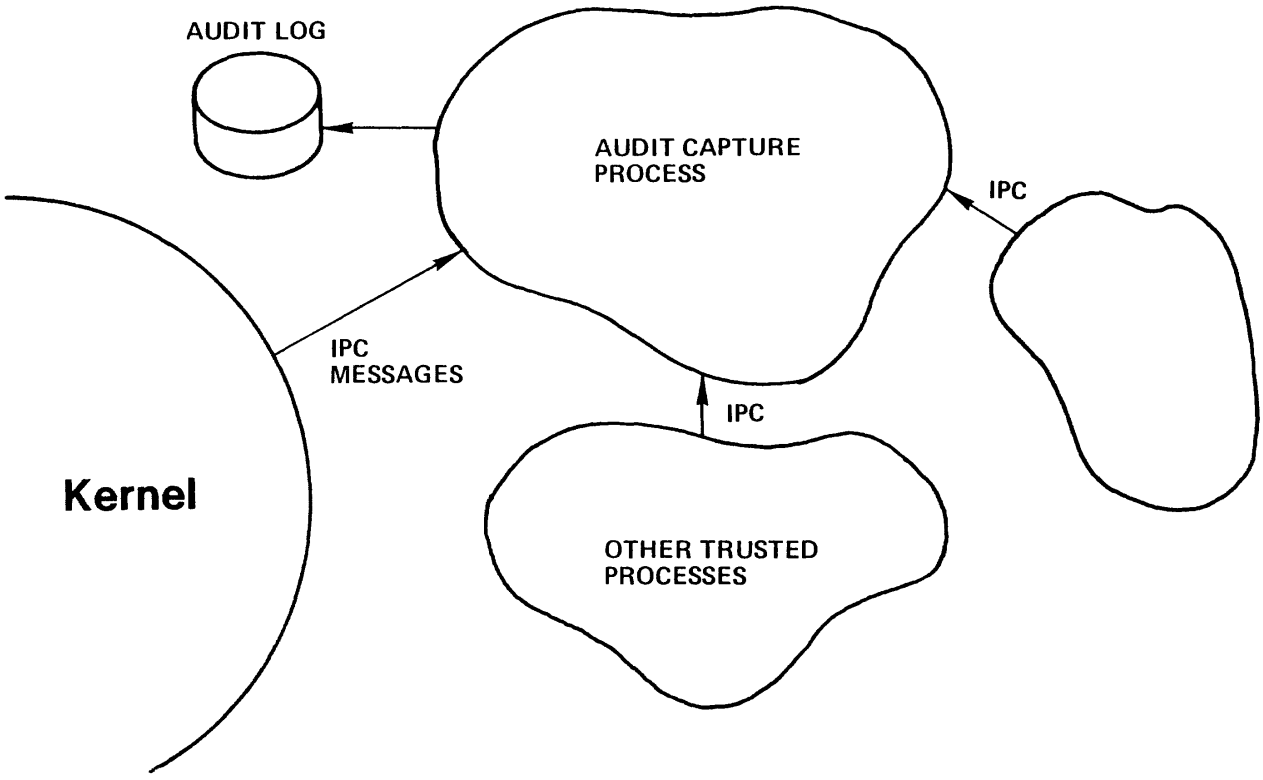

Figure 3—Audit information flow.

other processes via the UNIX pipe mechanism and the "ptrace" system call.

While this view simplifies the Emulator, it is incorrect. At the UNIX interface, a process is indirectly aware of the presence of other members of its "process family" (a process family consists of all the processes that are descendents of the process started at login for a given user). In particular, things like the seek pointers to open files are shared among the members of a process family. Our view of the Emulator now is that it provides an operating system for the process family. The Emulator not only creates the UNIX-level objects from the Kernel-level objects but also provides for controlled sharing of these UNIX-level objects.

It should be remarked that the UNIX interface is perhaps not as "clean" as one would like. There are several subtle ways in which a great deal of the internal mechanization of the system is manifest at the interface. It is debatable whether these things are "bugs" or are "features!"

UNIX directory management

One of the major functions of the Emulator is the creation of the UNIX file system from the more primitive file system provided by the Kernel. The Emulator caches the block i/o supported by the Kernel to provide the byte stream i/o supported by the UNIX interface. The Emulator also is where UNIX directories are managed. The final design of the UNIX directory management function is the result of a long series of (occasionally heated) debates on where directories would be mechanized. Initially they were to be completely managed by the Emulator. However, this was prior to the birth of the subtype notion, and there was no way to guarantee the integrity of the directory structure. In particular, trusted software could not depend upon the directory structure. Then it was proposed to move part or all of the directory management function into the Kernel. This seemed to solve the integrity problem, but opened a new and more serious problem of making the Kernel cognizant of the structure and semantics of directory files, and thereby making the Kernel very UNIX-specific. Finally, the subtype idea was proposed. The Kernel would know that directories were "special," and would aid in the preservation of their integrity. However, the Kernel would not be aware of the internal structure or semantics of directories.

The current design has the Emulator performing all the directory interpretation functions (i.e. recursively searching for names in directories), but writing directories is only done by the Directory Manager. The Directory Manager is a program that is K_spawned into execution whenever an Emulator needs to modify a directory. It starts its life running as the user "dir_mgr" who owns the directory subtype. After getting permission for write access to directory subtyped objects, the Directory Manager reverts its identity to that of the requesting user. From there on, the Kernel will enforce security, integrity and discretionary access checking. Thus, the user cannot trick the Directory Manager into modifying a directory that the user cannot access. This architecture may be criticized as being too slow, since creating a new process via K_spawn is moderately time-consuming. However, measurements on one of our UNIX systems in a software development environment suggest that modification of directories is a fairly infrequent occurrence.

Computer network support

The Emulator contains the bulk of the support for the computer network interface. Initially, KSOS will "speak" Version 4 of the Transmission Control Protocol (TCP)[12] including the Internet Datagram Layer.[11] This protocol appears to be on its way to becoming a future standard within DoD.

Although no networks presently exist that can handle multiple security levels, this architecture envisages their development and is designed to support them. To support a multi-level network, the Network Daemon would be trusted, so it could handle the multi-level stream to/from the network. The remainder of the TCP functions performed by the Emulator would be untrusted, since they are at only one level.

The basic structure of the KSOS network interface was discussed above, and is illustrated in Figure 4. There is a Network Daemon which handles the Internet Datagram protocol, and enough of the TCP to separate the i/o stream from the network into separate streams for each connection. In each Emulator is the majority of the TCP functionality. All of the functions relating to sequence number maintenance, window maintenance, acknowledgment, and retransmission are in the Emulator. This is possible because these are per connection functions, and need not be globally managed. These two portions of the TCP function communicate using the Kernel-supported IPC mechanisms. The shared segment mechanism is used for bulk data passing, and the event mechanism is used for synchronization, and for "commands" to the TCP Daemon and responses from it.

The non-Kernel system software

The purpose of this component of the KSOS system is to provide the software tools to support a KSOS system. The Non-Kernel System Software is divided into four groups:

1. *Secure User Services*—Software that manipulates the security levels of users and files. Also included in this class are all functions that require a secure ("unspoofable") path to the service.
2. *System Operation Services*—Software that performs continuing services for the system, such as the Network Daemon, line printer spooling and interuser mail.
3. *System Maintenance Services*—Software that performs occasional services primarily in the area of checking and repairing the consistency of the file system. Also included are the system generation functions. Individual KSOS sites can generate their system to suit the hardware configuration available.
4. *System Administrative Services*—Software that aids

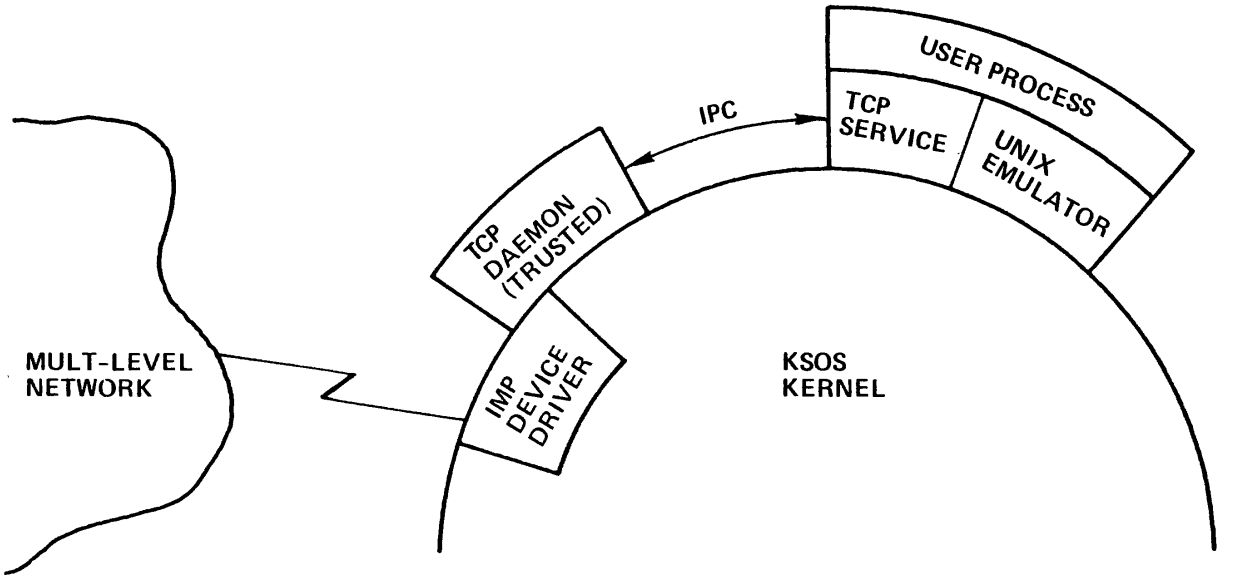

Figure 4—Network interface structure.

the System Administrator in controlling the sytem. Our goal has been that the System Administrator need not be a computer expert to perform his functions.

The Non-Kernel System Software described is a minimally complete set. Clearly there are large numbers of additional utilities that would be desirable. It is expected that this class will be supplemented extensively as KSOS matures.

KSOS APPLICATION CONSIDERATIONS

There are two broad classes of KSOS applications, each with different considerations. The first is applications that utilize the full KSOS system, i.e., applications based upon UNIX. KSOS should appear to these applications to be only slightly different than a standard UNIX operating system. Because KSOS provides a UNIX-like interface, meaningful secure applications can be built using the existing software. UNIX is one of the best systems in existence for the creation of new products by novel combinations of existing packages, and KSOS will preserve this flexibility. Such applications can, however, be made easier in some cases via the direct use of KSOS Kernel calls.

The second class of applications is those which use the Kernel directly without an Emulator. The Kernel provides many features that make it an attractive operating system in its own right. It offers excellent i/o performance, a range of IPC options, and many features that ease the design of multi-level applications. Because the Kernel is "UNIX-flavored" without being heavily UNIX-dependent, it is possible to create application environments that are an amalgamation of the features provided by different operating systems.

KSOS facilitates the creation of encapsulated environments that can be used for a variety of purposes. This encapsulation allows objects to be manipulated only by software known to perform correctly. In many cases only a small part of a multi-level application actually deals with data at different security levels. By encapsulation of these functions in a small trusted process, it is possible to build multi-level applications that minimize the amount of trusted (and therefore expensive) code.

ACKNOWLEDGMENTS

KSOS is being created by an exceptionally talented and dedicated team. It is a pleasure to acknowledge their contributions. The Government team on KSOS also deserves acknowledgment for their efforts to make KSOS a reality. Finally, credit must be given to Ken Thompson and Dennis Ritchie of Bell Laboratories for the creation of UNIX. We still marvel at the sophistication and elegance of their product.

REFERENCES

1. Bell, D. E. and L. J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Vols. I-III, MITRE Corporation, Bedford, MA, November 1973-June 1974.
2. Biba, K. J., "Integrity Considerations for Secure Computer Systems," MTR-3153, MITRE Corporation, Bedford, MA, June 1975.
3. "KSOS UNIX Emulator Computer Program Development Specification (Type B5)," WLD-TR7933, Ford Aerospace and Communications Corporation, Palo Alto, CA, September 1978.
4. "KSOS Implementation Plan," WDL-TR7799, Ford Aerospace and Communications Corporation, Palo Alto, CA, March 1978.
5. Kampe, M., C. Kline, G. Popek and E. Walton, "The UCLA Data Secure UNIX Operating System," Technical Report, University of California at Los Angeles, Los Angeles, CA, July 1977.
6. "KSOS Security Kernel Computer Program Development Specification

(Type B5)," WDL-TR7932, Ford Aerospace and Communications Corporation, Palo Alto, CA, September 1978.
7. "KSOS Maintenance and Support Plan," WDL-TR7810, Ford Aerospace and Communications Corporation, Palo Alto, CA, March 1978.
8. "Draft B5 Specifications for the MITRE Secure UNIX Prototype," Private Communication, 1977.
9. "KSOS Non-Kernel Security-Related Software Computer Program Development Specification (Type B5)," WDL-TR7934, Ford Aerospace and Communications Corporation, Palo Alto, CA, September 1978.
10. Parnas, D. L., "A Technique for Software Module Specification with Examples," *CACM*, Vol. 15, No. 5, May 1972, pp. 330-336.
11. Postel, J. B., "Internetwork Protocol Specification," Version 4, Information Sciences Institute, University of Southern California, Marina del Ray, CA, September 1978.
12. Postel, J. B., "Specification of Internetwork Transmission Control Protocol—TCP Version 4," Information Sciences Institute, University of Southern California, Marina del Rey, CA, September 1978.
13. Ritchie, D. M. and K. Thompson, "The UNIX Timesharing System," *CACM*, Vol. 17, No. 5, May 1974, pp. 365-375.
14. Robinson, L., K. N. Levitt, P. G. Neumann and A. R. Saxena, "A Formal Methodology for the Design of Operating System Software," in *Current Trends in Programming Methodology*, R. T. Yeh (ed.), Vol. 1, Prentice-Hall, April 1977.
15. Roubine, O. and L. Robinson, *SPECIAL Reference Manual*, 3rd ed., Technical Report CSG-45, SRI International, Menlo Park, CA, January 1977.
16. "KSOS Verification Plan," WDL-TR7809, Ford Aerospace and Communications Corporation, Palo Alto, CA, March 1978.