AT&T Technical Journal
Vol. 64, No. 1, January 1985
Printed in U.S.A.

System 75:

The Oryx/Pecos Operating System

By G. R. SAGER,* J. A. MELBER,† and K. T. FONG†

(Manuscript received July 11, 1984)

The System 75 Office Communication System is the first field application of the Oryx/Pecos operating system, a message-based system which supports real-time, distributed applications. Its interprocess communications mechanisms provide a structuring tool similar to monitors, capabilities, and abstract data types. This paper describes the principal concepts implemented in the operating system kernel, and presents the essential system processes. Support for application design techniques is discussed and related to proven software engineering principles, including information hiding and modularity. Specific examples are drawn from System 75 for call processing and maintenance.

I. INTRODUCTION

The Oryx/Pecos operating system provides an environment for real-time, distributed applications. By the term "real time" we mean that the performance of the system is reasonably fast and, above all, easy to predict. By the term "distributed" we mean that assignment of elements of the application to processors can be made apparent or transparent, as befits requirements.

This operating system is intended to extend the applications implementation language to include powerful structuring tools—similar to

* Currently with Sun Microsystems, Mountain View, California. † AT&T Information Systems Laboratories, an entity of AT&T Information Systems, Inc.

monitors,[1] capabilities,[2] and abstract data types[3]—as active elements of the application.

System 75 is the first Oryx/Pecos field application. In this initial application, the operating system runs on a single *Intel** 8086-based processor specially designed for System 75. All of the System 75 call processing, maintenance, and administration software is built on top of this operating system. To provide a responsive, feature rich, and extensible Private Branch Exchange (PBX)† with maximum system capacity, System 75 requires an operating system that is fast and can provide predictable performance. Use of other operating systems, such as the *UNIX*™ operating system, would not have met these needs.

The Oryx/Pecos operating system is implemented as a kernel (Oryx) and a set of essential system processes (Pecos).

II. THE ORYX KERNEL

The kernel (or "nucleus") of an operating system is a basic set of primitive operations from which the remainder of the system is constructed. The Oryx kernel appears to the programmer as an instruction set that manipulates *processes*, *messages*, and *paths*.

A *process* is the independent, sequential execution of a program. Processes may share instruction (I) space, but each process has its own stack and data (D) space. D spaces are not shared. The separation of D spaces is enforced by a memory-management device. Communication between processes is limited to messages and data transfers (as discussed below).

Processes are the source of asynchrony; the progress of independent but similar computations is modeled by processes with the same I space, each at a different stage of execution. Processes can act as monitors to solve critical sections and to enforce system policies.[4–6] Much of the operating system itself is contained in processes built on top of the kernel. Building with processes avoids a monolithic structure by enforcing physical separation of the system components. Parts of the operating system may be changed without affecting other parts (assuming interfaces are preserved). Bugs tend to be isolated. The system can be configured by adding or deleting system or application processes. Parts of the system or application may be designed to fail and recover without widespread repercussions.

A *message* is a small, fixed amount of information transmitted from a source process to a destination process. The source and destination processes may be on the same processor or on different processors.

---

* Trademark of Intel Corporation.

† Acronyms and abbreviations used in the text are defined at the back of the *Journal*.

Messages are also used to transfer arguments and results between the kernel and processes. Message transmission and reception simplifies interfaces and allows arguments and results to pass between processors for process-kernel interactions and for kernel-kernel interactions as they do for process-process interactions. In this respect, the kernel bears many similarities to a process. Similarly, device drivers are implemented as part of the kernel and interface with processes via messages.

Messages provide synchronization: they request an action or indicate that a requested action is complete. Messages are small (16 bytes) to reduce time for copying and space for buffering. Messages are fixed in size to avoid fragmentation in buffer allocation and disagreements on size between the source and destination.

Processor allocation (*dispatching*) is controlled by a combination of process priority and message transmission. When a message is sent from a lower- to a higher-priority process, the processor is allocated to the higher-priority process. This, coupled with nearly constant message transmission times, allows for greater ease in performance analysis of critical message sequences. Execution of a process is sequential; blocking occurs only as a result of waiting for a message and unblocking only as a result of message arrival. Message reception is entirely voluntary; the queueing of messages and ability to selectively receive on an "or" condition eliminates the need for signals or events and keeps the execution of processes sequential. The kernel translates device interrupts into messages and delivers them to device-controller processes. If an interrupt unblocks a higher-priority process, the currently executing process is preempted and will resume execution when all higher-priority processes have blocked.

Paths are a protection mechanism patterned after "links".[7] Message flow over a path is unidirectional: the *owner* of the path is the source, and the *creator* is the destination. In Fig. 1a, process A owns a path to process B, the path creator. For the path owner, the path represents a capability of sending messages; conversely, for the path creator, the path represents an agreement to receive messages. Paths may cross processor boundaries; this is transparent to both the owner and creator.

Messages can be used to set up new path connectivity. Process A agrees to receive messages from process B by creating a path (Fig. 1b). Process A then *passes* the path to process B over an existing path (Fig. 1c). Passing the path changes ownership, thereby enabling process B to send to process A. The initial paths among processes are controlled by the process manager (as described in Section III).

Paths have features to enhance their usefulness as a protection

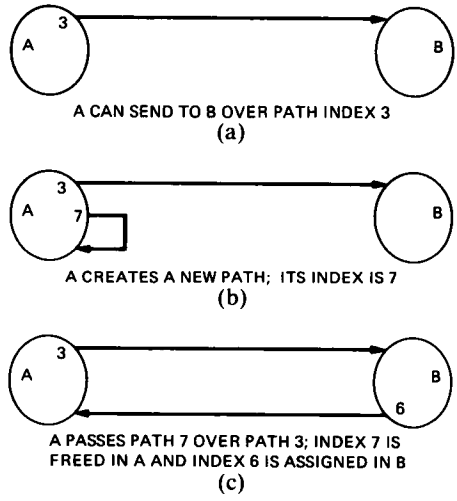

Fig. 1—Passing a path.

mechanism. Features are selected by the creator at path creation time, and they are enforced by the kernel; processes cannot "forge" a feature.

1. The *class* allows the creator to be selective in receiving messages. The creating process can specify one of seven classes, or can have the kernel select a class from a pool of classes not in use by the process. In a receive condition, if no messages are available on paths in the set of specified classes, the process blocks until one arrives. The kernel includes the path class as a part of all messages received over a path.

2. The *tag* allows the creating process to encode information concerning the reason for creating the path. The kernel includes the path tag as part of all messages received over a path to remind the creator of that reason. The tag is often a pointer to a data structure describing the state of the conversation on the path.

3. *Properties* can be used to prevent an owner from duplicating a path, to cause generation of a notification to the creator if the path is destroyed, or to limit path use to the transmission of a single message. The *duplicatable* property allows the owner of a path to make a copy of the path. Notifications enable the creator to keep an account of path sources; whenever an owner explicitly destroys a path with the notification property, the creator receives a notification with a count of how many copies of the path remain. A process death implies that paths owned by the process are destroyed; thus, a notification can inform the creator of the death of an owner. Limiting path use to a single transmission ensures that the creator can expect exactly one message back.

4. *Restrictions* limit the ability of an owner to give paths to the

creator. Restrictions help a creator avoid security problems generally classed as the Trojan Horse or cloying, i.e., false or unwanted paths.[8]

5. *Data transfer* paths and messages can set up an agreement to transfer bulk data in a manner that appears to the user as Direct Memory Access (DMA) input/output. The technique is convenient and more robust than messages for moving arbitrary amounts of data, as it avoids problems of message queue exhaustion and the need to sort the data stream from other incoming messages at the receiving end; furthermore, the creator of a data transfer path is allowed to execute while the data transfer occurs. Local or remote transfer of data is allowed.

Certain combinations of properties and restrictions occur frequently and are described below for convenience of discussion:

request—allows for the passing of *reply* paths and for the acquisition of *resource* paths. *Request* paths can be duplicatable.

resource—notifies the creator if the path is destroyed. It allows passing of *reply* paths and abstracts the allocation of a resource to the owner from the creator. *Resource* paths can be duplicatable.

reply—destroyed on use or notifies the creator if the path is destroyed. It allows for the passing of *resource* paths. *Reply* paths cannot be duplicated.

As an example of how to use processes, paths, and messages, we consider how one might implement a File System Server (FSS). In Fig. 2, process A is a client of the FSS; it may or may not be on the same processor as the FSS. Process A has the capability of opening

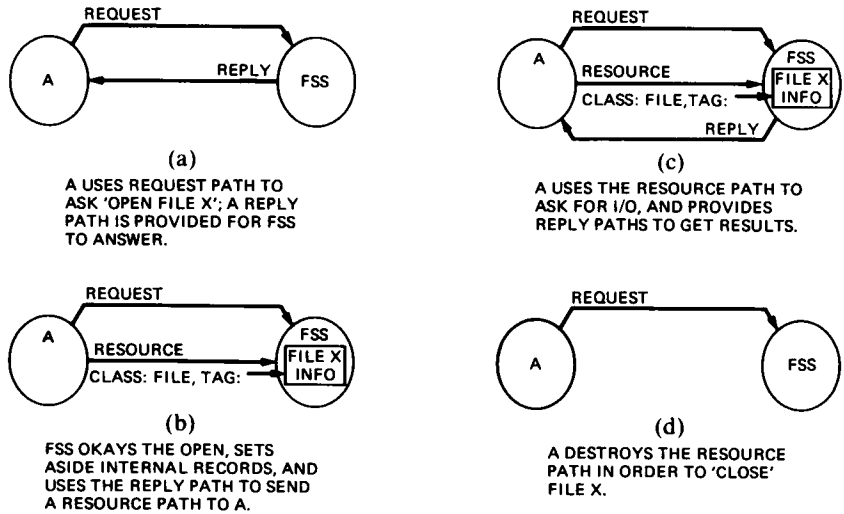

Fig. 2—File server example.

files; this capability is represented by the request path from A to the FSS, as in Fig. 2a. Process A opens a file by sending a message over the request path that says *open file x* and providing a reply path for a result (Fig. 2a). The FSS acknowledges the request by creating a resource path and using the reply path to pass it back to A (Fig. 2b); the FSS assigns the class it has reserved to represent open files and chooses the tag to point to a data structure containing data pertinent to the state of file x. The resource path represents a capability for A to ask the FSS to operate on file x. To operate on file x, A sends a message over the resource path and provides a reply path if a result is required (Fig. 2c). Note that the FSS relies (1) on the class to indicate that the message concerns an open file, and (2) on the tag to determine that the file to be operated on is x. Thus, it is impossible for A to lie (purposely or accidentally) to the FSS concerning the identity of the file to be operated on. Furthermore, A can pass only reply paths to the FSS so the FSS divests itself of paths passed from A when it sends a result. Process A closes file x by destroying the resource path (Fig. 2d). Note that closing the file occurs as a result of a destruction notification, rather than a message sent by A. Destruction assures the FSS that A can no longer communicate regarding file x, and it guarantees that the file is closed if A dies without explicitly closing.

This scenario resembles the control of files in the *UNIX* operating system (see Figs. 3 and 4). Process A's path (file) table is kept in protected space and managed by the kernel; paths (files) are referenced by an index into the path (file) table, which contains sensitive descriptor information. Automatic clean-up is possible because the information is kept in a disciplined fashion by the kernel, rather than in an undisciplined fashion by the process itself. *UNIX* system processes

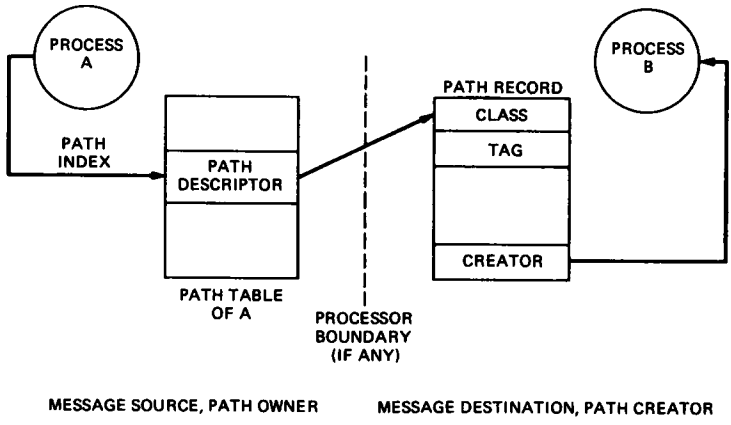

Fig. 3—The Oryx implementation of paths.

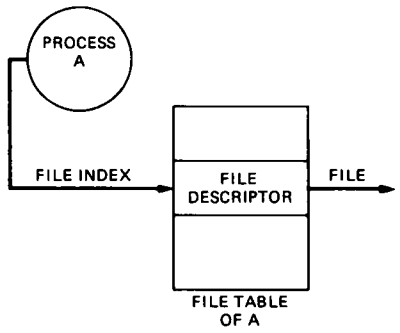

Fig. 4—The *UNIX* system implementation of files.

often use files as an abstraction mechanism to control resources. Paths are a more basic device to serve the same end in a distributed system.

It is important to note that the Oryx kernel separates the concepts of *mechanism* and *policy*. Paths provide a basic protection mechanism which processes can use to enforce protection policies. Thus, it is possible for process implementors to set and enforce policies according to their own requirements. This is important because the System 75 applications resemble an operating system in that they are concerned with the allocation and management of a complex set of resources; in many operating systems, these tasks can be accomplished efficiently only by implementing parts of the application in the operating system itself. Separation of policy and mechanism decoupled implementation of the System 75 applications from that of the operating system by reducing the need for the application developers to modify the kernel or operating system processes.

III. THE PECOS SYSTEM PROCESSES

The kernel relies on a set of system processes to provide certain maintenance and policy-making functions. In a distributed system, both the kernel and essential system processes are replicated on a per-processor basis.

The *phantom* acts on behalf of dead processes. When a process dies, processes owning paths that the dead process created execute asynchronously and may transmit messages to the dead process before learning of its death. In this case, the kernel delivers the messages to the phantom process for cleanup. The phantom destroys passed paths in the messages, thereby propagating destruction notifications to the creators of the passed paths.

The *Leisure Time Manager* (LTM) simplifies the kernel's dispatching decisions by ensuring that there is always a process to execute.

Once the LTM is allocated the processor, no other processes will execute until an interrupt occurs. The LTM runs at the next to lowest priority. Processes set to a lower priority than the LTM never execute, but otherwise appear to be normal processes. This is useful for debugging; a process being debugged can be halted without affecting the rest of a running system by placing it at the lowest priority.

The *Process Manager* (PM) is the first process to execute; it creates all other processes. The kernel defers all process creation and destruction decisions to the PM. The PM creates processes by associating memory-resident I spaces with the D spaces it allocates dynamically; processes remain in memory until they die. The PM determines the set of "standard paths" for a new process. Standard paths play a role similar to `stdin`, `stdout` and `stderr` in the *UNIX* system: a new process knows that some path indices can be used to obtain standard system services.

The *Network Manager* (NM) is created early during initialization, and all subsequent processes are created with a path to the NM. The NM allows processes to "supply" it paths with associated symbolic names and to ask for copies of paths by those symbolic names. This *name server* function allows processes to obtain paths without relying on ancestral relationships. The NM also provides a gateway to the NMs on other processors, and the NM is therefore useful for establishing contact with services on those processors. In the case of a processor failure, the NMs on functional processors will establish contact with the NM on a recovering processor and will provide a means to recover communications for the other processes in their processors.

Several other system processes are not essential, but are, nonetheless, useful in many applications. The *timer manager* provides alarm clock and time-of-day services. In System 75, the timer manager is used to time calls (for accounting records), to provide feature timing, to provide route timing (i.e., going to coverage), to recognize "no response" situations, and to schedule periodic maintenance activities. The *console manager* provides output to the system console and allows an "operator" to inspect the running system; the *error logger* saves log messages that may be useful for later debugging or accounting; and the *shuffler* provides long-term enforcement of the processor allocation policy. The console manager, shuffler, and error logger are not used in System 75. Finally, *optics* is a system process with special privileges to allow it to act as a debugger for other processes. The role of optics as a debugger is enhanced by the fact that it can track and display use of the kernel by a process. A more detailed description of optics can be found in Ref. 9.

IV. APPLICATION DESIGN

When designing an Oryx/Pecos-based application, a number of structuring techniques can be employed. We will discuss the most important and common ones in this section.

As previously indicated, processes provide *asynchrony* and *modularity*. However, these benefits would be limited without the ability for processes to communicate. Using paths and messages, it is possible to construct applications as a collection of cooperating processes. The principles for decomposing an application into processes are analogous to those used when designing with subroutines, *abstract data types*, or *monitors* (for example, see Ref. 10). The judicious use of paths and path features can enforce design decisions through a mechanism very much like *capabilities*.

When using processes for resource control, it is important to observe a *layering* of responsibility which preserves a *client-server relationship* through the layering. At any given instant of time, only *request* and *resource* paths should point down (from client to server) across layers and only *reply* paths should point up across layers, as in Figs. 5 and 6. When properly applied, this structure can make the application free of deadlock (Ref. 4 contains an excellent discussion of this).

Since processes communicate via paths, rather than directly to another process, it is possible to effectively hide the implementation of a layer consisting of many processes. This technique is used in two ways in System 75: (1) to provide multiple threads of execution and (2) to provide service through cooperation of servers.

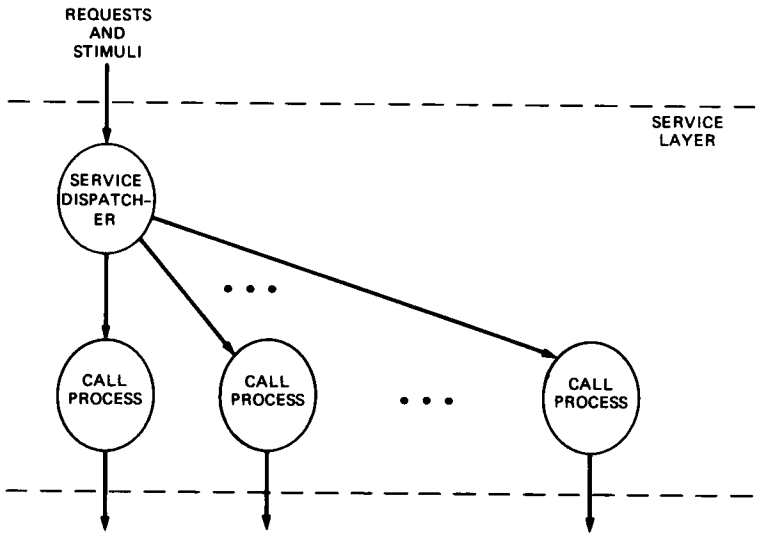

Fig. 5—Multiply threaded layer.

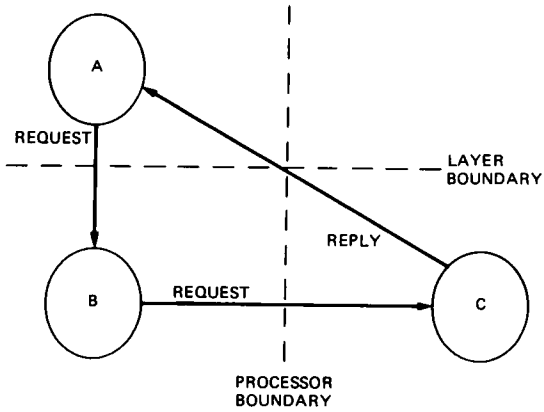

Fig. 6—Cooperative servers.

4.1 Multiple threads of execution

A PBX must handle many simultaneous phone calls, each potentially in a different stage of completion. This could be designed as one large, complex process, but System 75 implements the highest level of call control with two types of processes (Fig. 5): the Call Process (CP) and the Service Dispatcher (SD). There are many CPs sharing the instructions of a program that describes the sequence of steps required to control a phone call. The single SD process receives stimuli related to calls and directs them to the appropriate CP; when necessary, the SD will allocate an available CP to a new call or will make a CP for a disconnected call available. This structure allows asynchronous treatment of many phone calls and simplifies the coding of features by making feature control a conventional sequential programming task. Furthermore, this implementation is transparent to the other layers. A detailed discussion of this structuring technique can be found in Ref. 11. Details on the implementation of the SD and CP processes can be found in Ref. 12.

4.2 Cooperation of servers

In several cases, it is possible to subdivide the responsibilities of a layer to be shared among processes. This is sometimes useful when the coding is divided along similar boundaries. Figure 6 illustrates a typical structure for cooperative servers. It is important to note that when a request is forwarded to a peer server, the forwarding process can divest itself of all further responsibility by forwarding any passed paths with the original request. Thus, in Fig. 6, a message with a reply path is sent from A to B, B forwards the message and reply path to C, and C uses the reply path to respond directly to A when the service is complete. This technique will become more important when layers are

extended across processor boundaries; for example, messaging features (leave word calling, mail, etc.) may provide for messages stored in local memory or in a remote file server. A, of course, has no knowledge of whether or not its request is being handled by another processor.

V. APPLICATION EXAMPLE

To illustrate some of the concepts described above, this section presents a concrete example of how some Oryx/Pecos facilities are used by applications software in System 75. The communications subsystem is a set of processes that allow for direct data terminal communication with the administration and maintenance service processes. It consists of two control drivers, six identical data drivers, up to six identical Terminal Controllers (TCs), and the Communications Manager process (COM). The COM provides resource control for direct data ports and presents a uniform interface to the various service processes. Direct data communications with System 75 processes is possible through two main processor peripheral devices: a maintenance board (providing two data ports), and the data channel part of the switch interface board (providing four data ports). The service processes are generally unaware of what device they are using. Each data port is assigned a data driver, and each board is assigned a control driver. The process and path structure for a service process accessing port 1 of the maintenance board is shown in Fig. 7; the following paragraphs detail how this structure is arranged.

During initialization, the COM exchanges paths with each driver and supplies a path to the NM. All paths from data drivers have the same unique class. The tag of each path from the data drivers specifies the port being handled by that driver. All paths from control drivers have the same unique class. The tag of each path from a control driver specifies the board being controlled by the driver. Thus, when a message arrives, the COM process can determine the sending driver type from the class of the message and can determine which driver of that type from the tag of the message.

Service processes obtain a path to the COM from the NM. This path is used by service processes desiring data port access. The service processes typically send a message (detailing the request) and pass a reply path to the COM. The COM exchanges call progress messages (OFF-HOOK, ON-HOOK, etc.) with a control driver to establish a data call. It then requests creation of a TC process by message interchange with the Process Manager. Paths to the appropriate data driver are duplicated and passed to the newly created TC. Paths to the TC are in turn passed to the requesting service process (over the reply path mentioned earlier).

Thus path connectivity is established between the driver (which

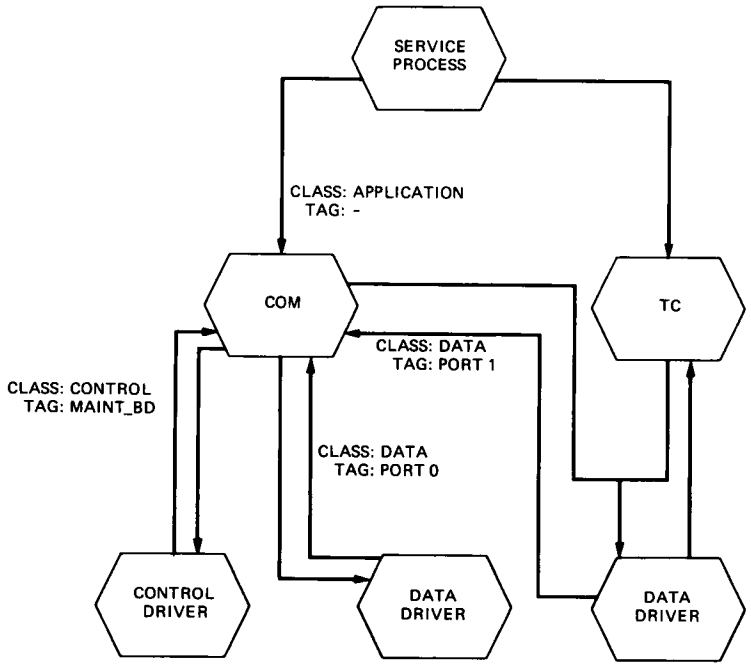

Fig. 7—Communications subsystem processes and paths for a service process with access to port 1 of the maintenance board.

manages the data port hardware), the TC (which provides higher-level data communications functions such as echoing), and the service process (which interprets the user keystrokes).

VI. MAINTENANCE FEATURES

In keeping with the overall system philosophy, the operating system attempts to provide mechanisms by which the application itself can implement a maintenance policy to detect and deal with extraordinary situations. The operating system allows for the provision of a designated application process to define the application maintenance policy; in System 75, this process is called the *High-Level Maintenance Manager* (HMM). If the HMM is not present, the operating system uses default rules to govern its behavior. If the HMM is present, the kernel and/or certain system processes will direct messages to the HMM warning of extraordinary situations and will rely on the HMM to take further action to deal with the situation.

The most basic Oryx/Pecos maintenance function is *process death.* Process death is accomplished when the PM forces a process to execute a program which causes it to release all of its system resources. In many cases, a process which encounters an error situation can correct

the problem by simply committing suicide. Certain processes are considered essential to the well-being of the application, and more drastic actions are required if they die. The PM consults the HMM before forcing the process to execute the death program; if the HMM decides that the process is essential, it will cause more extensive application maintenance to take place. A typical action would be to attempt to restart the process or a group of processes. Restart means that process data are preserved, the process stack is initialized and execution is resumed at the beginning, with arguments to indicate that a restart is taking place; if properly designed, a restarted process may be able to resume normal operation. It is possible to restart some processes after the operating system is reinitialized and thereby avoid the need to reload process instructions or data.

The NM supports asynchronous initialization and recovery by allowing processes to request a copy of a named path whenever it is supplied to the NM. A process which makes use of this feature will receive a fresh copy of the named path whenever it is supplied to the NM, and will therefore be in a position to make use of the latest version of the service provided over the path. Furthermore, the NM keeps copies of all supplied paths and will reissue copies on demand.

An example of the use of process restart and NM facilities is the Board Manager (BM) process. When the BM begins execution, it checks its arguments to determine if it is restarting. If it is, it uses a kernel audit operation to compare the contents of its path table with its internal record of indices of paths it owns. Paths in the path table but not in its records are destroyed; an example of a path which would be destroyed is a reply path whose index was stored in the stack. If its records indicate that a path is missing, a fresh copy of the path is demanded from the NM. When reinitialization is complete, the BM resumes normal operation.

Oryx/Pecos interfaces are provided to allow the HMM to estimate the overall ability of the system to provide service. For example, the kernel notifies the HMM when any available system resource (e.g., message buffers) goes below a low-water mark or above a high-water mark. The HMM uses low-water marks to detect and recover from overload situations; in this case the HMM may lower the priority of some work and/or prevent new work from entering the system. When a high-water mark is passed, the HMM will cause the system to resume normal operation.

Finally, the HMM directs auditing of operating system resources to detect problems in how the application is using those resources. For example, if a process has failed to detect or clean up properly after the death of another process, it may own a path which is no longer valid; in the kernel, this appears as a path descriptor with an invalid pointer

to a path record (see Fig. 3). Errors of this type are logged and clean-up actions are taken.

More details on the operation of the HMM, BM, and other components of System 75 maintenance can be found in Ref. 13.

VII. PERFORMANCE

The Oryx/Pecos operating system is designed to support real-time applications such as call processing. To provide this support, the operating system must be fast, and it must provide facilities so that system performance can be easily predicted.

Performance data are given for an 8-MHz *Intel* 8086 processor running with two memory wait states and equipped with memory management hardware designed specifically for System 75. Measurements were taken using a special I/O device; in-line I/O instructions output 16 bits of data to the device, which time-stamps the data and buffers it for later data reduction. The overhead for this instrumentation is very low and the granularity of the time-stamps is 10 microseconds. The measurements presented include time for kernel entry and exit, argument and result transfer, and process dispatch time, as well as time for the kernel operation itself.

There are variations in the observed message transmission times due to different possible states of the sender and receiver processes at the instant the message is sent. For example, if the sender unblocks the receiver and causes it to execute next, the observed time to transmit a message is:

send	0.62 ms
receive	0.29 ms

for a total of 0.91 ms. If, however, the sender queues up the message and the receiver dequeues it later, the times are:

send	0.50 ms
receive	0.47 ms

for a total of 0.97 ms. In many cases, the total time to transmit a message is the most important factor in performance, rather than the time attributable to either the sender or receiver. As can be seen, the total time for the two extreme cases above shows little variation and could be modeled as a constant for all cases.

When a path is passed with a message, the total time to transmit the message increases to a range of 1.35 to 1.45 ms. For any message sent over a reply path, 0.24 ms must be added to account for the implicit destruction of the path. The time required to create a path is 0.69 ms.

In practice, a typical application scenario is (see Fig. 2c):

1. A client process creates a reply path (0.69 ms),
2. The client sends the reply path with a message asking for service, causing the server to execute (1.35 ms),
3. The server sends a result message back over the reply path, queueing the message (0.97 + 0.24 ms).

The total time for the kernel operations involved is 3.25 ms. The Oryx kernel provides a specially packaged *call* operation to replace the client operations described above, thereby reducing the total time to 2.03 ms.

This type of performance data has been used to create accurate models of System 75 performance; these models were used during the design and implementation phases of System 75 call processing to maximize system capacity.

In addition to fast operation, other factors help make this operating system suitable for real-time applications. Real-time applications such as call processing are characterized by having to guarantee response times to external stimuli at maximum capacity. Thus, the Oryx/Pecos operating system provides facilities for improving response times and for making them predictable. Dispatching facilities (process priorities, priority preemption, and intimate coupling of dispatching and message transmission) enable the application designer to keep the processor allocated to the most critical work. Selective message reception using classes gives the designer the ability to deal with multiple, asynchronous message sources efficiently; this is important because real-time applications often need to respond immediately to stimuli from a number of sources, which may come in any order. Messages may be sent and received either synchronously or asynchronously. Synchronous operation is generally simpler to design and consumes less processor time, while asynchronous operation allows the designer to improve response times by breaking complex operations into short, non-atomic segments of execution and allowing time-critical requests to be handled sooner. Finally, the operating system itself is implemented so that critical operations are fast and "available"; that is, complex operating system functions are broken down into smaller segments so that the less important work can be deferred until later.

VIII. SIZE

The size of the Oryx/Pecos operating system is difficult to characterize in general terms because, to a large extent, its size is determined by the application it services; System 75 gives us a single data point from which to work. Thus, we will limit our discussion to the major factors which determine system size.

The application can directly influence system size in the following ways.

1. Use of services. If an application does not make use of a service, the process which provides that service can be eliminated (for example, the shuffler).

2. Per-process resources. Every application process requires a certain amount of system resources (for example, path tables); the number and size of these resources can be configured at compile time to suit the application.

3. Drivers. Drivers are included in the system size, but the application itself determines which drivers are needed.

Certain implementation decisions in the operating system influence size. In most cases, trade-offs between size and speed were decided in favor of speed, especially if the size came in the form of instructions. In the per-process and per-path data structures, the implementation tends to favor small size. Finally, additional instructions and data to serve the maintenance requirements of System 75 tend to increase the operating system size.

The most important Oryx/Pecos contribution to decreasing the overall size of an application is *shared libraries*. Shared libraries permit the sharing of a single physical copy of utility functions by all system and application processes. For example, any process that needs to format an output string does so by executing the same physical instructions as all other processes. The potential space savings is large: if an application has 25 different instruction spaces for processes and each makes extensive use of a 10K-byte shared library, the savings is slightly less than 240K bytes over what would be required if each process had its own copy of the library functions. The savings is not exactly 240K bytes because each process contains a few instructions to interface to each of the functions in the shared library.

It is not possible to give precise numbers for the space savings due to shared libraries in System 75 because the existence of shared libraries affects the implementation strategy. Rather than minimizing the size of library functions, and perhaps tailoring variants for individual processes, it is more advantageous to increase the generality of the functions to make sure that the maximum number of processes can make use of them. Furthermore, with the decreased size of the multiplier, there is a greater tendency to provide functionality that would otherwise have been omitted. Thus, a simple computation will overestimate the savings.

IX. CONCLUSIONS

We expect the use of the Oryx/Pecos operating system to lengthen the useful lifetime of System 75 by permitting easier modification of the applications, portability to other processor families, and expansion of feature content using distributed processing. One advantage of

implementing a PBX on top of an operating system is the potential for future integration with other computing environments; for example, the Oryx/Pecos operating system can provide a *UNIX*™ system execution environment and access to the rich set of *UNIX* system tools and applications.

X. ACKNOWLEDGMENTS

This paper represents the design and development work of many people at the Denver and Holmdel locations of AT&T Information Systems Laboratories. Many of the ideas used in the Oryx/Pecos operating system are adapted from experience with the DEMOS[7] and Thoth[4,14] operating systems and from experience gained from an earlier exploratory project.

REFERENCES

1. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," CACM, *17*, No. 10 (October 1974), pp. 549–57.
2. R. S. Fabry, "Capability-Based Addressing," CACM, *17*, No. 7 (July 1974), pp. 403–12.
3. B. Liskov et al., "Abstraction Mechanisms in CLU," CACM, *20*, No. 8 (August 1977), pp. 564–76.
4. D. R. Cheriton, *The Thoth System: Multiprocess Structuring and Portability*, New York: North Holland, 1982.
5. E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," in *Operating System Techniques*, C. A. R. Hoare and R. H. Perott, editors, New York: Academic Press, 1972, esp. pp. 91–3.
6. H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," 2nd International Colloquium on Operating Systems, IRIA (October, 1978). Reprinted in ACM Operating Systems Review, *13*, No. 2 (April, 1979), pp. 3–19.
7. F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in DEMOS," in Proc. Sixth ACM Symp. on Operating System Principles, Purdue University (November 1977), pp. 23–31.
8. T. A. Linden, "Operating System Structures to Support Security and Reliable Software," Computing Surveys, *8*, No. 4 (December 1976), pp. 409–45.
9. T. J. Pederson, J. E. Ritacco, and J. A. Santillo, "System 75: Software Development Tools," AT&T Tech. J., this issue.
10. D. L. Parnas, "On the Criteria Used in Decomposing Systems into Modules," CACM, *15*, No. 12 (December 1972), pp. 1053–8.
11. W. M. Gentleman, "Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept," Software—Practice and Experience, *11*, No. 5 (May 1981), pp. 435–66.
12. W. Densmore et al., "System 75: Switch Services Software," AT&T Tech. J., this issue.
13. K. S. Lu, J. D. Price, and T. L. Smith, "System 75: Maintenance Architecture,"AT&T Tech J., this issue.
14. R. Cheriton et al., "Thoth, a Portable Real-Time Operating System," CACM, *22*, No. 2 (February 1979), pp. 105–15.

AUTHORS

Kenneth T. Fong, B.S.E.E., 1970, California Institute of Technology; M.S.E.E., 1971, Stanford University; AT&T Bell Laboratories, 1970–1982; AT&T Information Systems Laboratories, 1983—. At AT&T, Mr. Fong has been involved with exploratory development of business communications systems, development of tools for software development, application of dis-

tributed processing techniques to real-time systems, and development of operating systems software. He is presently Head of the Operating Systems Development department. Member, IEEE.

John A. Melber, B.S.E.E., 1971, M.S.E.E., 1975, Polytechnic Institute of Brooklyn; Naval Air Development Center, 1971–1974; General Instrument Inc., 1974–1976; AT&T Bell Laboratories, 1976–1983; AT&T Information Systems Laboratories, 1983—. At the Naval Air Development Center, Mr. Melber designed automatic test equipment. At General Instrument, he was involved in simulator development, CAD database development, operating system support, and computer center operations. His initial AT&T assignment involved database and operating system work on the ACD-ESS Management Information System (AEMIS) project. In 1980, he moved into operating system work for System 75. He is presently Supervisor of the Common Component Software development group for System 75. Member, ACM, IEEE.

Gary R. Sager, B.S. (Mathematics), 1968, M.S. (Computer Science), 1969, and Ph.D. (Computer Science), 1972, University of Washington; Colorado State University, 1972–1974; Los Alamos Scientific Laboratory, 1978; University of Waterloo, 1974–1979; AT&T Bell Laboratories, 1979–1982; AT&T Information Systems Laboratories, 1983–1984; Sun Microsystems, 1984—. At AT&T Bell Laboratories and AT&T Information Systems Laboratories, Mr. Sager worked on the application of distributed processing techniques to real-time systems and the development of operating systems software.