Pilot: An Operating System for a Personal Computer

David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell
Xerox Business Systems

The Pilot operating system provides a single-user, single-language environment for higher level software on a powerful personal computer. Its features include virtual memory, a large "flat" file system, streams, network communication facilities, and concurrent programming support. Pilot thus provides rather more powerful facilities than are normally associated with personal computers. The exact facilities provided display interesting similarities to and differences from corresponding facilities provided in large multi-user systems. Pilot is implemented entirely in Mesa, a high-level system programming language. The modularization of the implementation displays some interesting aspects in terms of both the static structure and dynamic interactions of the various components.

Key Words and Phrases: personal computer, operating system, high-level language, virtual memory, file, process, network, modular programming, system structure

CR Categories: 4.32, 4.35, 4.42, 6.20

1. Introduction

As digital hardware becomes less expensive, more resources can be devoted to providing a very high grade of interactive service to computer users. One important expression of this trend is the personal computer. The dedication of a substantial computer to each individual user suggests an operating system design emphasizing close user/system cooperation, allowing full exploitation of a resource-rich environment. Such a system can also function as its user's representative in a larger community of autonomous personal computers and other information resources, but tends to deemphasize the largely ajudicatory role of a monolithic time-sharing system.

The Pilot operating system is designed for the personal computing environment. It provides a basic set of services within which higher level programs can more easily serve the user and/or communicate with other programs on other machines. Pilot omits certain functions that have been integrated into some other operating systems, such as character-string naming and user-command interpretation; such facilities are provided by higher level software, as needed. On the other hand, Pilot provides a more complete set of services than is normally associated with the "kernel" or "nucleus" of an operating system. Pilot is closely coupled to the Mesa programming langauge [16] and runs on a rather powerful personal computer, which would have been thought sufficient to support a substantial time-sharing system of a few years ago. The primary user interface is a high resolution bit-map display, with a keyboard and a pointing device. Secondary storage is provided by a sizable moving-arm disk. A local packet network provides a high bandwidth connection to other personal computers and to server systems offering such remote services as printing and shared file storage.

Much of the design of Pilot stems from an initial set of assumptions and goals rather different from those underlying most time-sharing systems. Pilot is a single-language, single-user system, with only limited features for protection and resource allocation. Pilot's protection mechanisms are *defensive*, rather than *absolute* [9], since in a single-user system, errors are a more serious problem than maliciousness. All protection in Pilot ultimately depends on the type-checking provided by Mesa, which is extremely reliable but by no means impenetrable. We have chosen to ignore such problems as "Trojan Horse" programs [20], not because they are unimportant, but because our environment allows such threats to be coped with adequately from outside the system. Similarly,

A version of this paper was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, Calif., Dec. 10–12, 1979.

Authors' address: Xerox Business Systems, 3333 Coyote Hill Rd., Palo Alto, CA 94304.

Pilot's resource allocation features are not oriented toward enforcing fair distribution of scarce resources among contending parties. In traditional multi-user systems, most resources tend to be in short supply, and prevention of inequitable distribution is a serious problem. In a single-user system like Pilot, shortage of some resource must generally be dealt with either through more effective utilization or by adding more of the resource.

The close coupling between Pilot and Mesa is based on mutual interdependence; Pilot is written in Mesa, and Mesa depends on Pilot for much of its runtime support. Since other languages are not supported, many of the language-independence arguments that tend to maintain distance between an operating system and a programming language are not relevant. In a sense, all of Pilot can be thought of as a very powerful runtime support package for the Mesa language. Naturally, none of these considerations eliminates the need for careful structuring of the combined Pilot/Mesa system to avoid accidental circular dependencies.

Since the Mesa programming language formalizes and emphasizes the distinction between an *interface* and its *implementation*, it is particularly appropriate to split the description of Pilot along these lines. As an environment for its client programs, Pilot consists of a set of Mesa interfaces, each defining a group of related types, operations, and error signals. Section 2 enumerates the major interfaces of Pilot and describes their semantics, in terms of both the formal interface and the intended behavior of the system as a whole. As a Mesa program, Pilot consists of a large collection of modules supporting the various interfaces seen by clients. Section 3 describes the interior structure of the Pilot implementation and mentions a few of the lessons learned in implementing an operating system in Mesa.

2. Pilot Interfaces

In Mesa, a large software system is constructed from two kinds of modules: *program* modules specify the algorithms and the actual data structures comprising the *implementation* of the system, while *definitions* modules formally specify the *interfaces* between program modules. Generally, a given interface, defined in a definitions module, is *exported* by one program module (its *implementor*) and *imported* by one or more other program modules (its *clients*). Both program and definitions modules are written in the Mesa source language and are compiled to produce binary object modules. The object form of a program module contains the actual code to be executed; the object form of a definitions module contains detailed specifications controlling the binding together of program modules. Modular programming in Mesa is discussed in more detail by Lauer and Satterthwaite [13].

Pilot contains two kinds of interfaces:

(1) *Public* interfaces defining the services provided by Pilot to its clients (i.e., higher level Mesa programs);
(2) *Private* interfaces, which form the connective tissue binding the implementation together.

This section describes the major features supported by the public interfaces of Pilot, including files, virtual memory, streams, network communication, and concurrent programming support. Each interface defines some number of named items, which are denoted *Interface.Item*. There are four kinds of items in interfaces: types, procedures, constants, and error signals. (For example, the interface *File* defines the type *File.Capability*, the procedure *File.Create*, the constant *file.maxPages PerFile*, and the error signal *File.Unknown*.) The discussion that follows makes no attempt at complete enumeration of the items in each interface, but focuses instead on the overall facility provided, emphasizing the more important and unusual features of Pilot.

2.1 Files

The Pilot interfaces *File* and *Volume* define the basic facilities for permanent storage of data. Files are the standard containers for information storage; volumes represent the media on which files are stored (e.g., magnetic disks). Higher level software is expected to superimpose further structure on files and volumes as necessary (e.g., an executable subsystem on a file, or a detachable directory subtree on a removable volume). The emphasis at the Pilot level is on simple but powerful primitives for accessing large bodies of information. Pilot can handle files containing up to about a million pages of English text, and volumes larger than any currently available storage device ($\sim 10^{13}$ bits). The total number of files and volumes that can exist is essentially unbounded (2^{64}). The space of files provided is "flat," in the sense that files have no recognized relationships among them (e.g., no directory hierarchy). The size of a file is adjustable in units of pages. As discussed below, the contents of a file are accessed by mapping one or more of its pages into a section of virtual memory.

The *File.Create* operation creates a new file and returns a capability for it. Pilot file capabilities are intended for *defensive* protection against errors [9]; they are mechanically similar to capabilities used in other systems for absolute protection, but are not designed to withstand determined attack by a malicious programmer. More significant than the protection aspect of capabilities is the fact that files and volumes are named by 64-bit universal identifiers (uids) which are guaranteed unique in both space and time. This means that distinct files, created anywhere at any time by any incarnation of Pilot, will always have distinct uids. This guarantee is crucial, since removable volumes are expected to be a standard method of transporting information from one

Pilot system to another. If uid ambiguity were allowed (e.g., different files on the same machine with the same uid), Pilot's life would become more difficult, and uids would be much less useful to clients. To guarantee uniqueness, Pilot essentially concatenates the machine serial number with the real time clock to produce each new uid.

Pilot attaches only a small fixed set of attributes to each file, with the expectation that a higher level directory facility will provide an extendible mechanism for associating with a file more general properties unknown to Pilot (e.g., length in bytes, date of creation, etc.). Pilot recognizes only four attributes: size, type, permanence, and immutability.

The *size* of a file is adjustable from 0 pages to 2^{23} pages, each containing 512 bytes. When the size of a file is increased, Pilot attempts to avoid fragmentation of storage on the physical device so that sequential or otherwise clustered accesses can exploit physical contiguity. On the other hand, random probes into a file are handled as efficiently as possible, by minimizing file system mapping overhead.

The *type* of a file is a 16-bit tag which is essentially uninterpreted, but is implemented at the Pilot level to aid in type-dependent recovery of the file system (e.g., after a system failure). Such recovery is discussed further in Section 3.4.

Permanence is an attribute attached to Pilot files that are intended to hold valuable permanent information. The intent is that creation of such a file proceed in four steps:

(1) The file is created using *File.Create* and has temporary status.
(2) A capability for the file is stored in some permanent directory structure.
(3) The file is made permanent using the *File.MakePermanent* operation.
(4) The valuable contents are placed in the file.

If a system failure occurs before step 3, the file will be automatically deleted (by the scavenger; see Section 3.4) when the system restarts; if a system failure occurs after step 2, the file is registered in the directory structure and is thereby accessible. (In particular, a failure between steps 2 and 3 produces a registered but nonexistent file, an eventuality which any robust directory system must be prepared to cope with.) This simple mechanism solves the "lost object problem" [25] in which inaccessible files take up space but cannot be deleted. Temporary files are also useful as scratch storage which will be reclaimed automatically in case of system failure.

A Pilot file may be made *immutable*. This means that it is permanently read-only and may never be modified again under any circumstances. The intent is that multiple physical copies of an immutable file, all sharing the *same* universal identifier, may be replicated at many physical sites to improve accessibility without danger of ambiguity concerning the contents of the file. For example, a higher level "linkage editor" program might wish to link a pair of object-code files by embedding the uid of one in the other. This would be efficient and unambiguous, but would fail if the contents were copied into a new pair of files, since they would have different uids. Making such files immutable and using a special operation (*File.ReplicateImmutable*) allows propagation of physical copies to other volumes without changing the uids, thus preserving any direct uid-level bindings.

As with files, Pilot treats volumes in a straightforward fashion, while at the same time avoiding oversimplifications that would render its facilities inadequate for demanding clients. Several different sizes and types of storage devices are supported as Pilot volumes. (All are varieties of moving-arm disk, removable or nonremovable; other nonvolatile random access storage devices could be supported.) The simplest notion of a volume would correspond one to one with a physical storage medium. This is too restrictive, and hence the abstraction presented at the *Volume* interface is actually a *logical volume*; Pilot is fairly flexible about the correspondence between logical volumes and *physical volumes* (e.g., disk packs, diskettes, etc.). On the one hand, it is possible to have a large logical volume which spans several physical volumes. Conversely, it is possible to put several small logical volumes on the same physical volume. In all cases, Pilot recognizes the comings and goings of physical volumes (e.g., mounting a disk pack) and makes accessible to client programs those logical volumes all of whose pages are on-line.

Two examples which originally motivated the flexibility of the volume machinery were database applications, in which a very large database could be cast as a multi-disk-pack volume, and the CoPilot debugger, which requires its own separate logical volume (see Section 2.5), but must be usable on a single-disk machine.

2.2 Virtual Memory

The machine architecture on which Pilot runs defines a simple linear virtual memory of up to 2^{32} 16-bit words. All computations on the machine (including Pilot itself) run in the same address space, which is unadorned with any noteworthy features, save a set of three flags attached to each page: *referenced*, *written*, and *write-protected*. Pilot structures this homogenous address space into contiguous runs of page called *spaces*, accessed through the interface *Space*. Above the level of Pilot, client software superimposes still further structure upon the contents of spaces, casting them as client-defined data structures within the Mesa language.

While the underlying linear virtual memory is conventional and fairly straightforward, the space machinery superimposed by Pilot is somewhat novel in its design, and rather more powerful than one would expect given the simplicity of the *Space* interface. A space is capable of playing three fundamental roles:

Allocation Entity. To allocate a region of virtual memory, a client creates a space of appropriate size.

Mapping Entity. To associate information content and backing store with a region of virtual memory, a client maps a space to a region of some file.

Swapping Entity. The transfer of pages between primary memory and backing store is performed in units of spaces.

Any given space may play any or all of these roles. Largely because of their multifunctional nature, it is often useful to nest spaces. A new space is always created as a subspace of some previously existing space, so that the set of all spaces forms a tree by containment, the root of which is a predefined space covering all of virtual memory.

Spaces function as allocation entities in two senses: when a space is created, by calling *Space.Create*, it is serving as the unit of allocation; if it is later broken into subspaces, it is serving as an allocation subpool within which smaller units are allocated and freed [19]. Such suballocation may be nested to several levels; at some level (typically fairly quickly) the page granularity of the space mechanism becomes too coarse, at which point finer grained allocation must be performed by higher level software.

Spaces function as mapping entities when the operation *Space.Map* is applied to them. This operation associates the space with a run of pages in a file, thus defining the content of each page of the space as the content of its associated file page, and propagating the write-protection status of the file capability to the space. At any given time, a page in virtual memory may be accessed only if its content is well-defined, i.e., if *exactly one* of the nested spaces containing it is mapped. If none of the containing spaces is mapped, the fatal error *AddressFault* is signaled. (The situation in which more than one containing space is mapped cannot arise, since the *Space.Map* operation checks that none of the ancestors or descendents of a space being mapped are themselves already mapped.) The decision to cast *AddressFault* and *WriteProtectFault* (i.e., storing into a write-protected space) as fatal errors is based on the judgment that any program which has incurred such a fault is misusing the virtual memory facilities and should be debugged; to this end, Pilot unconditionally activates the CoPilot debugger (see Section 2.5).

Spaces function as swapping entities when a page of a mapped space is found to be missing from primary memory. The swapping strategy followed is essentially to swap in the lowest level (i.e., smallest) space containing the page (see Section 3.2). A client program can thus optimize its swapping behavior by subdividing its mapped spaces into subspaces containing items whose access patterns are known to be strongly correlated. In the absence of such subdivision, the entire mapped space is swapped in. Note that while the client can always opt for demand paging (by breaking a space up into one-page subspaces), this is *not* the default, since it tends to promote thrashing. Further optimization is possible using the *Space.Activate* operation. This operation advises Pilot that a space will be used soon and should be swapped in as soon as possible. The inverse operation, *Space.Deactivate*, advises Pilot that a space is no longer needed in primary memory. The *Space.Kill* operation advises Pilot that the current contents of a space are of no further interest (i.e., will be completely overwritten before next being read) so that useless swapping of the data may be suppressed. These forms of optional advice are intended to allow tuning of heavy traffic periods by eliminating unnecessary transfers, by scheduling the disk arm efficiently, and by insuring that during the visit to a given arm position all of the appropriate transfers take place. Such advice-taking is a good example of a feature which has been deemed undesirable by most designers of timesharing systems, but which can be very useful in the context of a dedicated personal computer.

There is an intrinsic close coupling between Pilot's file and virtual memory features: virtual memory is the only access path to the contents of files, and files are the only backing store for virtual memory. An alternative would have been to provide a separate backing store for virtual memory and require that clients transfer data between virtual memory and files using explicit read/write operations. There are several reasons for preferring the mapping approach, including the following.

(1) Separating the operations of mapping and swapping decouples buffer allocation from disk scheduling, as compared with explicit file read/write operations.
(2) When a space is mapped, the read/write privileges of the file capability can propagate automatically to the space by setting a simple read/write lock in the hardware memory map, allowing illegitimate stores to be caught immediately.
(3) In either approach, there are certain cases that generate extra unnecessary disk transfers; extra "advice-taking" operations like *Space.Kill* can eliminate the extra disk transfers in the mapping approach; this does not seem to apply to the read/write approach.
(4) It is relatively easy to simulate a read/write interface given a mapping interface, and with appropriate use of advice, the efficiency can be essentially the same. The converse appears to be false.

The Pilot virtual memory also provides an advice-like operation called *Space.ForceOut*, which is designed as an underpinning for client crash-recovery algorithms. (It is advice-like in that its effect is invisible in normal operation, but becomes visible if the system crashes.) *ForceOut* causes a space's contents to be written to its backing file and does not return until the write is completed. This means that the contents will survive a subsequent system crash. Since Pilot's page replacement algorithm is also free to write the pages to the file at any time (e.g., between *ForceOuts*), this facility by itself does *not* constitute even a minimal crash recovery mechanism; it is intended only as a "toehold" for higher level software

to use in providing transactional atomicity in the face of system crashes.

2.3 Streams and I/O Devices

A Pilot client can access an I/O device in three different ways:

(1) *implicitly*, via some feature of Pilot (e.g., a Pilot file accessed via virtual memory);
(2) *directly*, via a low-level device driver interface exported from Pilot;
(3) *indirectly*, via the Pilot stream facility.

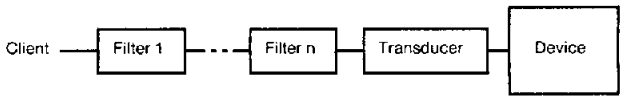
In keeping with the objectives of Pilot as an operating system for a personal computer, most I/O devices are made directly available to clients through low-level procedural interfaces. These interfaces generally do little more than convert device-specific I/O operations into appropriate procedure calls. The emphasis is on providing maximum flexibility to client programs; protection is not required. The only exception to this policy is for devices accessed implicitly by Pilot itself (e.g., disks used for files), since chaos would ensue if clients also tried to access them directly.

For most applications, direct device access via the device driver interface is rather inconvenient, since all the details of the device are exposed to view. Furthermore, many applications tend to reference devices in a basically sequential fashion, with only occasional, and usually very stylized, control or repositioning operations. For these reasons, the Pilot *stream* facility is provided, comprising the following components:

(1) The *stream* interface, which defines device independent operations for full-duplex sequential access to a source/sink of data. This is very similar in spirit to the stream facilities of other operating systems, such as OS6 [23] and UNIX [18].
(2) A standard for *stream components*, which connect streams to various devices and/or implement "on-the-fly" transformations of the data flowing through them.
(3) A means for cascading a number of primitive stream components to provide a compound stream.

There are two kinds of stream components defined by Pilot: the transducer and the filter. A *transducer* is a module which imports a device driver interface and exports an instance of the Pilot *Stream* interface. The transducer is thus the implementation of the basic sequential access facility for that device. Pilot provides standard transducers for a variety of supported devices. A *filter* is a module which imports one instance of the Pilot standard *Stream* interface and exports another. Its purpose is to transform a stream of data "on the fly" (e.g., to do code or format conversion). Naturally, clients can augment the standard set of stream components provided with Pilot by writing filters and transducers of their own. The *Stream* interface provides for dynamic binding of stream components at runtime, so that a transducer and a set of filters can be cascaded to provide a *pipeline*, as shown in Figure 1.

Fig. 1. A pipeline of cascaded stream components.



The transducer occupies the lowest position in the pipeline (i.e., nearest the device) while the client program accesses the highest position. Each filter accesses the next lower filter (or transducer) via the *Stream* interface, just as if it were a client program, so that no component need be aware of its position in the pipeline, or of the nature of the device at the end. This facility resembles the UNIX pipe and filter facility, except that it is implemented at the module level within the Pilot virtual memory, rather than as a separate system task with its own address space.

2.4 Communications

Mesa supports a shared-memory style of interprocess communication for *tightly coupled* processes [11]. Interaction between *loosely coupled* processes (e.g., suitable to reside on different machines) is provided by the Pilot *communications* facility. This facility allows client processes in different machines to communicate with each other via a hierarchically structured family of packet communication protocols. Communication software is an integral part of Pilot, rather than an optional addition, because Pilot is intended to be a suitable foundation for network-based distributed systems.

The protocols are designed to provide communication across multiple interconnected networks. An interconnection of networks is referred to as an *internet*. A Pilot internet typically consists of local, high bandwidth Ethernet broadcast networks [15], and public and private long-distance data networks like SBS, TELENET, TYMNET, DDS, and ACS. Constituent networks are interconnected by *internetwork routers* (often referred to as *gateways* in the literature) which store and forward packets to their destination using distributed routing algorithms [2, 4]. The constituent networks of an internet are used only as a transmission medium. The source, destination, and internetwork router computers are *all* Pilot machines. Pilot provides software drivers for a variety of networks; a given machine may connect directly to one or several networks of the same or different kinds.

Pilot clients identify one another by means of *network addresses* when they wish to communicate and need not know anything about the internet toplogy or each other's locations or even the structure of a network address. In particular, it is not necessary that the two communicators be on different computers. If they are on the same computer, Pilot will optimize the transmission of data between them and will avoid use of the physical network resources. This implies that an isolated computer (i.e.,

one which is not connected to any network) may still contain the communications facilities of Pilot. Pilot clients on the same computer should communicate with one another using Pilot's communications facilities, as opposed to the tightly coupled mechanisms of Mesa, if the communicators are loosely coupled subsystems that could some day be reconfigured to execute on different machines on the network. For example, printing and file storage server programs written to communicate in the loosely coupled mode could share the same machine if the combined load were light, yet be easily moved to separate machines if increased load justified the extra cost.

A network address is a resource assigned to clients by Pilot and identifies a specific *socket* on a specific machine. A socket is simply a site from which packets are transmitted and at which packets are received; it is rather like a post office box, in the sense that there is no assumed relationship among the packets being sent and received via a given socket. The identity of a socket is unique only at a given point in time; it *may* be reused, since there is no long-term static association between the socket and any other resources. Protection against dangling references (e.g., delivery of packets intended for a previous instance of a given socket) is guaranteed by higher level protocols.

A network address is, in reality, a triple consisting of a 16-bit network number, a 32-bit processor ID, and a 16-bit socket number, represented by a system-wide Mesa data type *System.NetworkAddress*. The internal structure of a network address is not used by clients, but by the communications facilities of Pilot and the internetwork routers to deliver a packet to its destination. The administrative procedures for the assignment of network numbers and processor IDs to networks and computers, respectively, are outside the scope of this paper, as are the mechanisms by which clients find out each others' network addresses.

The family of packet protocols by which Pilot provides communication is based on our experiences with the Pup Protocols [2]. The Arpa Internetwork Protocol family [8] resemble our protocols in spirit. The protocols fall naturally into three levels:

Level 0: Every packet must be encapsulated for transmission over a particular communication medium, according to the network-specific rules for that communication medium. This has been termed level 0 in our protocol hierarchy, since its definition is of no concern to the typical Pilot client.

Level 1: Level 1 defines the format of the *internetwork packet*, which specifies among other things the source and destination network addresses, a checksum field, the length of the entire packet, a transport control field that is used by internetwork routers, and a packet type field that indicates the kind of packet defined at level 2.

Level 2: A number of level 2 packet formats exist, such as error packet, connection-oriented sequenced packet, routing table update packet, and so on. Various level 2 protocols are defined according to the kinds of level 2 packets they use, and the rules governing their interaction.

The *Socket* interface provides level 1 access to the communication facilities, including the ability to create a socket at a (local) network address, and to transmit and receive internetwork packets. In the terms of Section 2.3, sockets can be thought of as *virtual devices*, accessed directly via the *Socket* (virtual driver) interface. The protocol defining the format of the internetwork packet provides end-to-end communication at the packet level. The internet is required only to be able to transport independently addressed packets from source to destination network addresses. As a consequence, packets transmitted over a socket may be expected to arrive at their destination only with *high probability* and not necessarily in the order they were transmitted. It is the responsibility of the communicating end processes to agree upon higher level protocols that provide the appropriate level of reliable communication. The *Socket* interface, therefore, provides service similar to that provided by networks that offer *datagram* services [17] and is most useful for connectionless protocols.

The interface *NetworkStream* defines the principal means by which Pilot clients can communicate reliably between any two network addreses. It provides access to the implementation of the *sequenced packet protocol*—a level 2 protocol. This protocol provides sequenced, duplicate-suppressed, error-free, flow-controlled packet communication over arbitrarily interconnected communication networks and is similar in philosophy to the Pup Byte Stream Protocol [2] or the Arpa Transmission Control Protocol [3, 24]. This protocol is implemented as a transducer, which converts the device-like *Socket* interface into a Pilot stream. Thus all data transmission via a network stream is invoked by means of the operations defined in the standard *Stream* interface.

Network streams provide reliable communication, in the sense that the data is reliably sent from the source transducer's packet buffer to the destination transducer's packet buffer. No guarantees can be made as to whether the data was successfully received by the destination client or that the data was appropriately processed. This final degree of reliability must lie with the clients of network streams, since they alone know the higher level protocol governing the data transfer. Pilot provides communication with varying degrees of reliability, since the communicating clients will, in general, have differing needs for it. This is in keeping with the design goals of Pilot, much like the provision of defensive rather than absolute protection.

A network stream can be set up between two communicators in many ways. The most typical case, in a network-based distributed system, involves a *server* (a supplier of a service) at one end and a *client* of the service at the other. Creation of such a network stream is inherently asymmetric. At one end is the server which

advertises a network address to which clients can connect to obtain its services. Clients do this by calling *NetworkStream.Create*, specifying the address of the server as parameter. It is important that concurrent requests from clients not conflict over the server's network address; to avoid this, some additional machinery is provided at the server end of the connection. When a server is operational, one of its processes *listens* for requests on its advertised network address. This is done by calling *NetworkStream.Listen*, which automatically creates a new network stream each time a request arrives at the specified network address. The newly created network stream connects the client to *another* unique network address on the server machine, leaving the server's advertised network address free for the reception of additional requests.

The switchover from one network address to another is transparent to the client, and is part of the definition of the sequenced packet protocol. At the server end, the *Stream.Handle* for the newly created stream is typically passed to an *agent*, a subsidiary process or subsystem which gives its full attention to performing the service for that particular client. These two then communicate by means of the new network stream set up between them for the duration of the service. Of course, the *NetworkStream* interface also provides mechanisms for creating connections between arbitrary network addresses, where the relationship between the processes is more general than that of server and client.

The mechanisms for establishing and deleting a connection between any two communicators and for guarding against old duplicate packets are a departure from the mechanisms used by the Pup Byte Stream Protocol [2] or the Transmission Control Protocol [22], although our protocol embodies similar principles. A network stream is terminated by calling *NetworkStream.Delete*. This call initiates no network traffic and simply deletes all the data structures associated with the network stream. It is the responsibility of the communicating processes to have decided *a priori* that they wish to terminate the stream. This is in keeping with the decision that the reliable processing of the transmitted data ultimately rests with the clients of network streams.

The manner in which server addresses are advertised by servers and discovered by clients is not defined by Pilot; this facility must be provided by the architecture of a particular distributed system built on Pilot. Generally, the binding of names of resources to their addresses is accomplished by means of a network-based database referred to as a *clearinghouse*. The manner in which the binding is structured and the way in which clearinghouses are located and accessed are outside the scope of this paper.

The communication facilities of Pilot provide clients various interfaces, which provide varying degrees of service at the internetworking level. In keeping with the overall design of Pilot, the communication facility attempts to provide a standard set of features which capture the most common needs, while still allowing clients to custom tailor their own solutions to their communications requirements if that proves necessary.

2.5 Mesa Language Support

The Mesa language provides a number of features which require a nontrivial amount of runtime support [16]. These are primarily involved with the control structure of the language [10, 11] which allow not only recursive procedure calls, but also coroutines, concurrent processes, and signals (a specialized form of dynamically bound procedure call used primarily for exception handling). The runtime support facilities are invoked in three ways:

(1) explicitly, via normal Mesa interfaces exported by Pilot (e.g., the *Process* interface);
(2) implicitly, via compiler-generated calls on built-in procedures;
(3) via traps, when machine-level op-codes encounter exceptional conditions.

Pilot's involvement in client procedure calls is limited to trap handling when the supply of activation record storage is exhausted. To support the full generality of the Mesa control structures, activation records are allocated from a heap, even when a strict LIFO usage pattern is in force. This heap is replenished and maintained by Pilot.

Coroutine calls also proceed without intervention by Pilot, except during initialization when a trap handler is provided to aid in the original setup of the coroutine linkage.

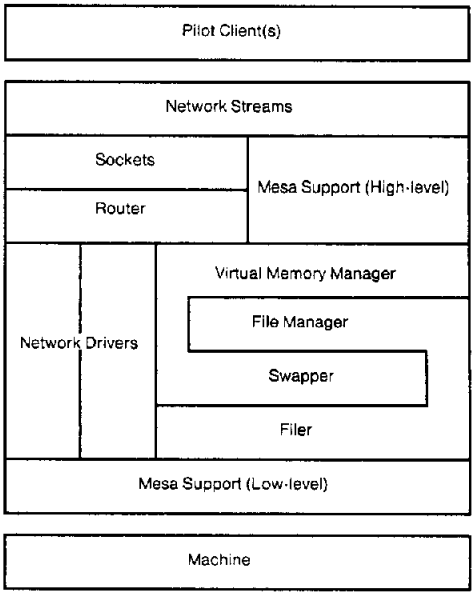
Pilot's involvement with concurrent processes is somewhat more substantial. Mesa casts process creation as a variant of a procedure call, but unlike a normal procedure call, such a FORK statement *always* invokes Pilot to create the new process. Similarly, termination of a process also involves substantial participation by Pilot. Mesa also provides monitors and condition variables for synchronized interprocess communication via shared memory; these facilities are supported directly by the machine and thus require less direct involvement of Pilot.

The Mesa control structure facilities, including concurrent processes, are light weight enough to be used in the fine-scale structuring of normal Mesa programs. A typical Pilot client program consists of some number of processes, any of which may at any time invoke Pilot facilities through the various public interfaces. It is Pilot's responsibility to maintain the semantic integrity of its interfaces in the face of such client-level concurrency (see Section 3.3). Naturally, any higher level consistency constraints invented by the client must be guaranteed by client-level synchronization, using monitors and condition variables as provided in the Mesa language.

Another important Mesa-support facility which is provided as an integral part of Pilot is a "world-swap" facility to allow a graceful exit to CoPilot, the Pilot/Mesa interactive debugger. The world-swap facility saves the

contents of memory and the total machine state and then starts CoPilot from a *boot-file*, just as if the machine's bootstrap-load button had been pressed. The original state is saved on a second boot-file so that execution can be resumed by doing a second world-swap. The state is saved with sufficient care that it is virtually always possible to resume execution without any detectable perturbation of the program being debugged. The world-swap approach to debugging yields strong isolation between the debugger and the program under test. Not only the contents of main memory, but the version of Pilot, the accessible volume(s), and even the microcode can be different in the two worlds. This is especially useful when debugging a new version of Pilot, since CoPilot can run on the old, stable version until the new version becomes trustworthy. Needless to say, this approach is not directly applicable to conventional multi-user time-sharing systems.

Fig. 2. Major components of Pilot.



3. Implementation

The implementation of Pilot consists of a large number of Mesa modules which collectively provide the client environment as decribed above. The modules are grouped into larger *components*, each of which is responsible for implementing some coherent subset of the overall Pilot functionality. The relationships among the major components are illustrated in Figure 2.

Of particular interest is the interlocking structure of the four components of the *storage system* which together implement files and virtual memory. This is an example of what we call the *manager/kernel* pattern, in which a given facility is implemented in two stages: a low-level kernel provides a basic core of function, which is extended by the higher level manager. Layers interposed between the kernel and the manager can make use of the kernel and can in turn be used by the manager. The same basic technique has been used before in other systems to good effect, as discussed by Habermann et al. [6], who refer to it as "functional hierarchy." It is also quite similar to the familiar "policy/mechanism" pattern [1, 25]. The main difference is that we place no emphasis on the possibility of using the same kernel with a variety of managers (or without any manager at all). In Pilot, the manager/kernel pattern is intended only as a fruitful decomposition tool for the design of integrated mechanisms.

3.1 Layering of the Storage System Implementation

The kernel/manager pattern can be motivated by noting that since the purpose of Pilot is to provide a more hospitable environment than the bare machine, it would clearly be more pleasant for the code implementing Pilot if it could use the facilities of Pilot in getting its job done. In particular, both components of the *storage system* (the file and virtual memory implementations) maintain internal databases which are too large to fit in primary memory, but only parts of which are needed at any one time. A client-level program would simply place such a database in a file and access it via virtual memory, but if Pilot itself did so, the resulting circular dependencies would tie the system in knots, making it unreliable and difficult to understand. One alternative would be the invention of a special separate mechanism for low-level disk access and main memory buffering, used only by the storage system to access its internal databases. This would eliminate the danger of circular dependency but would introduce more machinery, making the system bulkier and harder to understand in a different sense. A more attractive alternative is the extraction of a streamlined kernel of the storage system functionality with the following properties:

(1) It can be implemented by a small body of code which resides permanently in primary memory.
(2) It provides a powerful enough storage facility to significantly ease the implementation of the remainder of the full-fledged storage system.
(3) It can handle the majority of the "fast cases" of client-level use of the storage system.

Figure 2 shows the implementation of such a kernel storage facility by the swapper and the filer. These two subcomponents are the kernels of the virtual memory and file components, respectively, and provide a reasonably powerful environment for the nonresident subcomponents, the virtual memory manager, and the file manager, whose code and data are both swappable. The kernel environment provides somewhat restricted virtual memory access to a small number of special files and to preexisting normal files of fixed size.

The managers implement the more powerful operations, such as file creation and deletion, and the more complex virtual memory operations, such as those that

traverse subtrees of the hierarchy of nested spaces. The most frequent operations, however, are handled by the kernels essentially on their own. For example, a page fault is handled by code in the swapper, which calls the filer to read the appropriate page(s) into memory, adjusts the hardware memory map, and restarts the faulting process.

The resident data structures of the kernels serve as caches on the swappable databases maintained by the managers. Whenever a kernel finds that it cannot perform an operation using only the data in its cache, it conceptually "passes the buck" to its manager, retaining no state information about the failed operation. In this way, a circular dependency is avoided, since such failed operations become the total responsibility of the manager. The typical response of a manager in such a situation is to consult its swappable database, call the resident subcomponent to update its cache, and then retry the failed operation.

The intended dynamics of the storage system implementation described above are based on the expectation that Pilot will experience three quite different kinds of load.

(1) For short periods of time, client programs will have their essentially static working sets in primary memory and the storage system will not be needed.
(2) Most of the time, the client working set will be changing slowly, but the description of it will fit in the swapper/filer caches, so that swapping can take place with little or no extra disk activity to access the storage system databases.
(3) Periodically, the client working set will change drastically, requiring extensive reloading of the caches as well as heavy swapping.

It is intended that the Pilot storage system be able to respond reasonably to all three situations: In case (1), it should assume a low profile by allowing its swappable components (e.g., the managers) to swap out. In case (2), it should be as efficient as possible, using its caches to avoid causing spurious disk activity. In case (3), it should do the best it can, with the understanding that while continuous operation in this mode is probably not viable, short periods of heavy traffic can and must be optimized, largely via the advice-taking operations discussed in Section 2.2.

3.2 Cached Databases of the Virtual Memory Implementation

The virtual memory manager implements the client visible operations on spaces and is thus primarily concerned with checking validity and maintaining the database constituting the fundamental representation behind the *Space* interface. This database, called the *hierarchy*, represents the tree of nested spaces defined in Section 2.2. For each space, it contains a record whose fields hold attributes such as size, base page number, and mapping information.

The swapper, or virtual memory kernel, manages primary memory and supervises the swapping of data between mapped memory and files. For this purpose it needs access to information in the hierarchy. Since the hierarchy is swappable and thus off limits to the swapper, the swapper maintains a resident *space cache* which is loaded from the hierarchy in the manner described in Section 3.1.

There are several other data structures maintained by the swapper. One is a bit-table describing the allocation status of each page of primary memory. Most of the bookkeeping performed by the swapper, however, is on the basis of the *swap unit*, or smallest set of pages transferred between primary memory and file backing storage. A swap unit generally corresponds to a "leaf" space; however, if a space is only partially covered with subspaces, each maximal run of pages not containing any subspaces is also a swap unit. The swapper keeps a *swap unit cache* containing information about swap units such as extent (first page and length), containing mapped space, and state (mapped or not, swapped in or out, replacement algorithm data).

The swap unit cache is addressed by page rather than by space; for example, it is used by the page fault handler to find the swap unit in which a page fault occurred. The content of an entry in this cache is logically derived from a sequence of entries in the hierarchy, but direct implementation of this would require several file accesses to construct a single cache entry. To avoid this, we have chosen to maintain another database: the *projection*. This is a second swappable database maintained by the virtual memory manager, containing descriptions of all existing swap units, and is used to update the swap unit cache. The existence of the projection speeds up page faults which cannot be handled from the swap unit cache; it slows down space creation/deletion since then the projection must be updated. We expect this to be a useful optimization based on our assumptions about the relative frequencies and CPU times of these events; detailed measurements of a fully loaded system will be needed to evaluate the actual effectiveness of the projection.

An important detail regarding the relationship between the manager and kernel components has been ignored up to this point. That detail is avoiding "recursive" cache faults; when a manager is attempting to supply a missing cache entry, it will often incur a page fault of its own; the handling of that page fault must *not* incur a second cache fault or the fault episode will never terminate. Basically the answer is to make certain key records in the cache ineligible for replacement. This pertains to the space and swap unit caches and to the caches maintained by the filer as well.

3.3 Process Implementation

The implementation of processes and monitors in Pilot/Mesa is summarized here; more detail can be found in [11].

The task of implementing the concurrency facilities is split roughly equally among Pilot, the Mesa compiler, and the underlying machine. The basic primitives are defined as language constructs (e.g., entering a MONITOR, WAITing on a CONDITION variable, FORKing a new PROCESS) and are implemented either by machine op-codes (for heavily used constructs, e.g., WAIT) or by calls on Pilot (for less heavily used constructs, e.g., FORK). The constructs supported by the machine and the low-level Mesa support component provide procedure calls and synchronization among existing processes, allowing the remainder of Pilot to be implemented as a collection of monitors, which carefully synchronize the multiple processes executing concurrently inside them. These processes comprise a variable number of client processes (e.g., which have called into Pilot through some public interface) plus a fixed number of dedicated system processes (about a dozen) which are created specially at system initialization time. The machinery for creating and deleting processes is a monitor within the high-level Mesa support component; this places it above the virtual memory implementation; this means that it is swappable, but also means that the rest of Pilot (with the exception of network streams) cannot make use of dynamic process creation. The process implementation is thus another example of the manager/kernel pattern, in which the manager is implemented at a very high level and the kernel is pushed down to a very low level (in this case, largely into the underlying machine). To the Pilot client, the split implementation appears as a unified mechanism comprising the Mesa language features and the operations defined by the Pilot *Process* interface.

3.4 File System Robustness

One of the most important properties of the Pilot file system is robustness. This is achieved primarily through the use of *reconstructable maps.* Many previous systems have demonstrated the value of a *file scavenger*, a utility program which can repair a damaged file system, often on a more or less *ad hoc* basis [5, 12, 14, 21]. In Pilot, the scavenger is given first-class citizenship, in the sense that the file structures were all designed from the beginning with the scavenger in mind. Each file page is self-identifying by virtue of its *label*, written as a separate physical record adjacent to the one holding the actual contents of the page. (Again, this is not a new idea, but is the crucial foundation on which the file system's robustness is based.) Conceptually, one can think of a file page access proceeding by scanning all known volumes, checking the label of each page encountered until the desired one is found. In practice, this scan is performed only once by the scavenger, which leaves behind maps on each volume describing what it found there; Pilot then uses the maps and incrementally updates them as file pages are created and deleted. The logical redundancy of the maps does not, of course, imply lack of importance, since the system would be not be viable without them; the point is that since they contain *only* redundant information, they can be completely reconstructed should they be lost. In particular, this means that damage to any page on the disk can compromise only data on that page.

The primary map structure is the volume file map, a B-tree keyed on ⟨file-uid, page-number⟩ which returns the device address of the page. All file storage devices check the label of the page and abort the I/O operation in case of a mismatch; this does not occur in normal operation and generally indicates the need to scavenge the volume. The volume file map uses extensive compression of uids and run-encoding of page numbers to maximize the out-degree of the internal nodes of the B-tree and thus minimize its depth.

Equally important but much simpler is the volume allocation map, a table which describes the allocation status of each page on the disk. Each free page is a self-identifying member of a hypothetical file of free pages, allowing reconstruction of the volume allocation map.

The robustness provided by the scavenger can only guarantee the integrity of files as defined by Pilot. If a database defined by client software becomes inconsistent due to a system crash, a software bug, or some other unfortunate event, it is little comfort to know that the underlying file has been declared healthy by the scavenger. An "escape-hatch" is therefore provided to allow client software to be invoked when a file is scavenged. This is the main use of the file-type attribute mentioned in Section 2.1. After the Pilot scavenger has restored the low-level integrity of the file system, Pilot is restarted; before resuming normal processing, Pilot first invokes all client-level scavenging routines (if any) to reestablish any higher level consistency constraints that may have been violated. File types are used to determine which files should be processed by which client-level scavengers.

An interesting example of the first-class status of the scavenger is its routine use in transporting volumes between versions of Pilot. The freedom to redesign the complex map structures stored on volumes represents a crucial opportunity for continuing file system performance improvement, but this means that one version of Pilot may find the maps left by another version totally inscrutable. Since such incompatibility is just a particular form of "damage," however, the scavenger can be invoked to reconstruct the maps in the proper format, after which the corresponding version of Pilot will recognize the volume as its own.

3.5 Communication Implementation

The software that implements the packet communication protocols consists of a set of network-specific drivers, modules that implement sockets, network stream transducers, and at the heart of it all, a *router.* The router is a software switch. It routes packets among sockets, sockets and networks, and networks themselves. A router is present on *every* Pilot machine. On personal machines, the router handles only incoming, outgoing, and intra-

machine packet traffic. On internetwork router machines, the router acts as a service to other machines by transporting internetwork packets across network boundaries. The router's data structures include a list of all active sockets and networks on the local computer. The router is designed so that network drivers may easily be added to or removed from new configurations of Pilot; this can even be done dynamically during execution. Sockets come and go as clients create and delete them. Each router maintains a routing table indicating, for a given remote network, the best internetwork router to use as the next "hop" toward the final destination. Thus, the two kinds of machines are essentially special cases of the same program. An internetwork router is simply a router that spends most of its time forwarding packets between networks and exchanging routing tables with other internetwork routers. On personal machines the router updates its routing table by querying internetwork routers or by overhearing their exchanges over broadcast networks.

Pilot has taken the approach of connecting a network much like any other input/output device, so that the packet communication protocol software becomes part of the operating system and operates in the same personal computer. In particular, Pilot does *not* employ a dedicated front-end communications processor connected to the Pilot machine via a secondary interface.

Network-oriented communication differs from conventional input/output in that packets arrive at a computer *unsolicited*, implying that the intended recipient is unknown until the packet is examined. As a consequence, each incoming packet must be buffered initially in router-supplied storage for examination. The router, therefore, maintains a buffer pool shared by all the network drivers. If a packet is undamaged and its destination socket exists, then the packet is copied into a buffer associated with the socket and provided by the socket's client.

The architecture of the communication software permits the computer supporting Pilot to behave as a user's personal computer, a supplier of information, or as a dedicated internetwork router.

3.6 The Implementation Experience

The initial construction of Pilot was accomplished by a fairly small group of people (averaging about 6 to 8) in a fairly short period of time (about 18 months). We feel that this is largely due to the use of Mesa. Pilot consists of approximately 24,000 lines of Mesa, broken into about 160 modules (programs and interfaces), yielding an average module size of roughly 150 lines. The use of small modules and minimal intermodule connectivity, combined with the strongly typed interface facilities of Mesa, aided in the creation of an implementation which avoided many common kinds of errors and which is relatively rugged in the face of modification. These issues are discussed in more detail in [7] and [13].

4. Conclusion

The context of a large personal computer has motivated us to reevaluate many design decisions which characterize systems designed for more familiar situations (e.g., large shared machines or small personal computers). This has resulted in a somewhat novel system which, for example, provides sophisticated features but only minimal protection, accepts advice from client programs, and even boot-loads the machine periodically in the normal course of execution.

Aside from its novel aspects, however, Pilot's real significance is its careful integration, in a single relatively compact system, of a number of good ideas which have previously tended to appear individually, often in systems which were demonstration vehicles not intended to support serious client programs. The combination of streams, packet communications, a hierarchical virtual memory mapped to a large file space, concurrent programming support, and a modular high-level language, provides an environment with relatively few artificial limitations on the size and complexity of the client programs which can be supported.

Acknowledgments. The primary design and implementation of Pilot were done by the authors. Some of the earliest ideas were contributed by D. Gifford, R. Metcalfe, W. Shultz, and D. Stottlemyre. More recent contributions have been made by C. Fay, R. Gobbel, F. Howard, C. Jose, and D. Knutsen. Since the inception of the project, we have had continuous fruitful interaction with all the members of the Mesa language group; in particular, R. Johnsson, J. Sandman, and J. Wick have provided much of the software that stands on the border between Pilot and Mesa. We are also indebted to P. Jarvis and V. Schwartz, who designed and implemented some of the low-level input/output drivers. The success of the close integration of Mesa and Pilot with the machine architecture is largely due to the talent and energy of the people who designed and built the hardware and microcode for our personal computer.

Received June 1979; accepted September 1979; revised November 1979

References

1. Brinch-Hansen, P. The nucleus of a multiprogramming system. *Comm. ACM 13*, 4 (April 1970), 238–241.
2. Boggs, D.R., Shoch, J.F., Taft, E., and Metcalfe, R.M. Pup: An internetwork architecture. To appear in *IEEE Trans. Commun.* (Special Issue on Computer Network Architecture and Protocols).
3. Cerf, V.G., and Kahn, R.E. A protocol for packet network interconnection. *IEEE Trans. Commun. COM-22*, 5 (May 1974), 637–641.
4. Cerf, V.G., and Kirstein, P.T. Issues in packet-network interconnection. *Proc. IEEE 66*, 11 (Nov. 1978), 1386–1408.
5. Farber, D.J., and Heinrich, F.R. The structure of a distributed computer system: The distributed file system. In Proc. 1st Int. Conf. Computer Communication, 1972, pp. 364–370.
6. Habermann, A.N., Flon, L., and Cooprider, L. Modularization and hierarchy in a family of operating systems. *Comm. ACM 19*, 5 (May 1976), 266–272.
7. Horsley, T.R., and Lynch, W.C. Pilot: A software engineering

case history. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 94–99.
8. Internet Datagram Protocol, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.
9. Lampson, B.W. Redundancy and robustness in memory protection. *Proc. IFIP 1974*, North Holland, Amsterdam, pp. 128–132.
10. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. In *Lecture Notes in Computer Science 19*, Springer-Verlag, New York, 1974, pp. 181–203.
11. Lampson, B.W., and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23*, 2 (Feb. 1980), 105–117.
12. Lampson, B.W., and Sproull, R.F. An open operating system for a single user machine. Presented at the ACM 7th Symp. Operating System Principles (*Operating Syst. Rev. 13*, 5), Dec. 1979, pp. 98–105.
13. Lauer, H.C., and Satterthwaite, E.H. The impact of Mesa on system design. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 174–182.
14. Lockemann, P.C., and Knutsen, W.D. Recovery of disk contents after system failure. *Comm. ACM 11*, 8 (Aug. 1968), 542.
15. Metcalfe, R.M., and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), pp. 395–404.
16. Mitchell, J.G., Maybury, W., and Sweet, R. Mesa Language Manual. Tech. Rep., Xerox Palo Alto Res. Ctr., 1979.
17. Pouzin, L. Virtual circuits vs. datagrams—technical and political problems. Proc. 1976 NCC, AFIPS Press, Arlington, Va., pp. 483–494.
18. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
19. Ross, D.T. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.
20. Rotenberg, Leo J. Making computers keep secrets. Tech. Rep. MAC-TR-115, MIT Lab. for Computer Science.
21. Stern, J.A. Backup and recovery of on-line information in a computer utility. Tech. Rep. MAC-TR-116 (thesis), MIT Lab. for Computer Science, 1974.
22. Sunshine, C.A., and Dalal, Y.K. Connection management in transport protocol. *Comput. Networks 2*, 6 (Dec. 1978), 454–473.
23. Stoy, J.E., and Strachey, C. OS6—An experimental operating system for a small computer. *Comput. J. 15*, 2 and 3 (May, Aug. 1972).
24. Transmission Control Protocol, TCP, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.
25. Wulf, W., et. al. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM 17*, 6 (June 1974), 337–345.

Operating Systems

R. Stockton Gaines
Editor

Medusa: An Experiment in Distributed Operating System Structure

John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu
Carnegie-Mellon University

The design of Medusa, a distributed operating system for the Cm* multimicroprocessor, is discussed. The Cm* architecture combines distribution and sharing in a way that strongly impacts the organization of operating systems. Medusa is an attempt to capitalize on the architectural features to produce a system that is modular, robust, and efficient. To provide modularity and to make effective use of the distributed hardware, the operating system is partitioned into several disjoint *utilities* that communicate with each other via messages. To take advantage of the parallelism present in Cm* and to provide robustness, all programs, including the utilities, are *task forces* containing many concurrent, cooperating *activities*.

Key Words and Phrases: operating systems, distributed systems, message systems, task forces, deadlock, exception reporting
CR Categories: 4.32, 4.35

This research was supported by the Department of Defense Advanced Research Projects Agency, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Authors' present addresses: J.K. Ousterhout and P.S. Sindhu, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; D.A. Scelza, PRIME Computer, Inc., Old Connecticut Path, Framingham, MA 01701.