Pilot: An operating system for a personal computer

(summary)

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 Corporation
Systems Development Department
Palo Alto, CA

Overview

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 sometimes associated with "complete" operating systems, such as character-string naming or user-command interpretation; higher-level software provides such facilities as needed. On the other hand, Pilot provides a higher level of service than that normally associated with the "kernel" or "nucleus" of an operating system. Pilot is closely coupled to the Mesa programming language and runs on a rather powerful personal computer, which would have been thought sufficient to support a substantial timesharing 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 generally takes the form of a sizable local 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 timesharing 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*, since in a single user system, errors are a more serious problem than maliciousness. Similarly, Pilot's resource allocation features are not oriented toward enforcing fair distribution of scarce resources among contending parties.

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 eliminate the need for careful structuring of the combined Pilot/Mesa system to avoid accidental circular dependencies.

Files

Files and volumes define the basic facilities for permanent storage of data in Pilot. Files are the standard containers for information storage; volumes represent the media on which files are stored. Higher level software can superimpose further structure on files and volumes as necessary. The emphasis at the Pilot level is on simple but powerful primitives for accessing large bodies of information. The fundamental design of Pilot allows files containing the equivalent of a million pages of English text, and volumes larger than any currently conceivable storage device. The total number of files and volumes that can exist is essentially unbounded. The space of files provided is "flat," in the sense that files have no recognized relationships among them (e.g. no directory hierarchy). Pilot files are named by capabilities, which provide defensive protection against errors, and contain 64-bit universal identifiers, which are guaranteed unique in both space and time. This guarantee is crucial, since files are expected to migrate from one Pilot system to another.

The contents of a file are accessed by mapping one or more of its pages into a section of virtual memory. Pilot attempts to optimize the frequent case of sequential access to a file, but the general mechanism is designed to make random access as efficient as possible, by minimizing file system mapping overhead.

As with files, Pilot treats volumes in a relatively simple fashion; Pilot distinguishes physical and logical volumes, and is fairly flexible about the correspondence between the two. As the system runs, 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.

One of the most important properties of the Pilot file system is robustness. This is achieved primarily through the use of *reconstructable hints*. 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. 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 page contents; the intent is that damage to a single page does not damage data outside that page. All global structures of the Pilot file system can be reconstructed by the scavenger; it is intended that higher level scavengers repair damage to higher level structures in an analogous fashion.

Virtual memory

The machine architecture on which Pilot runs defines a single, fairly conventional linear virtual memory of up to 2^{32} 16-bit words, which Pilot structures into contiguous runs of pages called *spaces*. The space abstraction superimposed by Pilot is somewhat novel in its design and rather more powerful than one would expect given its simplicity. 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 vitual 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.

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.

The Pilot virtual memory also provides several advice-taking operations to allow client programs to express their intentions in ways which help optimize swapping traffic.

Streams and I/O Devices

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

> *Implicitly*, via some feature of Pilot (e.g. a Pilot file accessed via virtual memory),
>
> *Directly*, via a low-level device driver interface exported from Pilot, or
>
> *Indirectly*, via the Pilot stream facility.

Since typical applications need only simple sequential access to a device, direct access via the device driver is generally both unnecessary and inconvenient. Pilot therefore provides a *stream* facility, comprising:

> The basic stream abstraction, which defines device independent operations for full-duplex sequential access to a source/sink of data.
>
> A standard for *stream components*, which connect streams to various devices and/or implement "on-the-fly" transformations of the data flowing through them.
>
> A means for cascading a number of primitive stream components to provide a compound stream.

Communications

Mesa supports a shared-memory style of interprocess communication for *tightly coupled* processes. 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 to communicate with each other via a family of hierarchically structured 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, typically comprising local, high bandwidth networks, and public or private long-distance data networks. The networks are interconnected by *internetwork routers* which store and forward packets to their destination using distributed routing algorithms. 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.

Network streams provide the principal means by which Pilot clients can communicate reliably. They provides access to the implementation of the *sequenced packet protocol*. This protocol provides sequenced, duplicate-suppressed, error-free, flow-controlled packet communication over arbitrarily interconnected communication networks. The most typical case is the asymmetric one of a stream with a *server* at one end, and a *client* of that server at the other. Special facilities are provided to ease the creation of server/client network streams, without compromising the generality of the underlying mechanism.

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

Mesa language support

The Mesa language provides a number of features which require a non-trivial amount of run-time support. These are primarily involved with the control structures of the language, which allow not only recursive procedure calls, but also coroutines, dynamic creation of concurrent processes, and synchronization via monitors and condition variables.

The Mesa control structure facilities, including processes, are light-weight enough to be used in the fine-scale structuring of normal Mesa programs. A typical Pilot client program consists of a variable number of processes, any of which may at any time invoke Pilot facilities. It is Pilot's responsibility to maintain the semantic integrity of its abstractions in the face of such client-level concurrency. Naturally, any higher level consistency constraints invented by the client must be guaranteed by client-level synchronization, using the facilities provided in the Mesa language and supported by Pilot.