Observations on the Development of an Operating System

Hugh C. Lauer
Xerox Corporation
Palo Alto, California

The development of Pilot, an operating system for a personal computer, is reviewed, including a brief history and some of the problems and lessons encountered during this development. As part of understanding how Pilot and other operating systems come about, an hypothesis is presented that systems can be classified into five kinds according to the style and direction of their development, independent of their structure. A further hypothesis is presented that systems such as Pilot, and many others in widespread use, take about five to seven years to reach maturity, independent of the quality and quantity of the talent applied to their development. The pressures, constraints, and problems of producing Pilot are discussed in the context of these hypotheses.

Key words and phrases: Operating system, system development, software engineering, Pilot, personal computer, system classification.
CR Categories: 4.35, 4.30.

This paper contains my personal observations about the development of Pilot, an operating system for a personal computer [Redell *et al*], compared and contrasted with some other operating systems with which I have had contact. In these observations, I concentrate not on the anatomy of these systems but rather on their life cycles, particularly their formative years from conception to birth to maturity. This is a somewhat unorthodox point of view in the technical literature which abounds with papers on operating system techniques and structures, software engineering tools and methods, and general exhortations about the right and wrong ways to develop systems. But it is a useful one, not only for the student of operating systems and system development, but also for the managers or sponsors of development projects who need some understanding about why systems are so dramatically different from each other, why some succeed and others fail, and what might expected from development organizations.

In comparing Pilot with other operating systems, I have found it useful to classify operating systems into five categories according to how they came about, how successful they were, and their impact on the computing community. This classification is one of the main themes of this paper. It is interesting to observe that systems classified as the second kind, including Pilot and many of the major operating systems in widespread use, seem to take from five to seven years to grow from birth to maturity. Furthermore, it seems that this five to seven years is necessary, independent of the amount or quality of the talent applied to an operating system development project.

The paper has four main parts. In the first two parts, I chronicle the development of Pilot and some of the data and problems pertaining to it; this chronicle is presented not because it is new, different, or novel, but because we rarely talk about these things in the literature and there are lessons to be learned. In the third part, I offer the classification and some comments on systems in each category. In the final part, I make some observations about the five-to-seven year rule, why it seems to be true, and what its consequences are.

History of Pilot

We use the term 'Pilot' in three different ways, to mean an operating system kernel, a major project and system, and a way of life.

- As an operating system kernel, Pilot is the system described in [Redell, *et al*]. It consists of approximately 25,000-50,000 lines of code (depending upon how you count) in the Mesa programming language [Mitchell *et al*] and was developed over four years by a group of four to eight people, many of whom had other responsibilities at the same time.
- As a system development project, Pilot consists of approximately 250,000 lines of Mesa code in about two dozen major subsystems, including the operating system kernel, CoPilot (the Mesa debugger), a common user interface package and framework for building development tools, various utilities and communications packages, microcode for defining the Mesa architecture in several processors, and other facilities. For historical and organizational reasons, another 200,000 lines of code in compilers, binders, Mesa utilites, librarian tools, change request tools, etc.—much of which was Alto-based until early 1981—are not included in Pilot. Together, these two bodies of software represent both the operating system to be embedded in client application systems and the necessary support to develop and test those applications. Approximately 40 people have contributed to these systems over five years.
- As a way of life, Pilot defines a framework for thinking about, designing, and implementing systems and for communicating among subsystems within machines and across networks. It is the operating system for the Mesa machine architecture and hence part of a number of Xerox products, the foundation for the Mesa development environment, and a tool for supporting software research through the Cedar system [Deutsch and Taft].

Most other operating systems suffer the same multiple uses of their names. In this paper, I will use the term Pilot and any other operating sytem in the second sense, to include all the supporting software that makes the system usable. Note that Pilot is not ordinarily visible to Xerox customers; it was primarily intended to be embedded in products and to be used internally in the research and development environments.

Although it has roots in earlier work on the Alto system [Sproull and Lampson] and Mesa at the Xerox Palo Alto Research Center (PARC), serious work on Pilot began in the System Development Department (SDD) in early 1976. Most of the people who worked on Pilot over the past five years had previous experience primarily in the academic and research communities, and very few had ever built any 'production' system, let alone an operating system, to commercial pressures of schedule and budgets. We knew about the differences between the big, ponderous, ungainly systems sold by some computer manufacturers and the simple, elegant, lightweight systems imagined in the research community, and we were determined to do it right. This meant using a good, high-level language (Mesa), assigning a fairly small group of knowledgeable people to the project, carefully studying the lessons of others in the computer science community and from the Alto environment at PARC, and designing the hardware and software together. This ought to take two or perhaps three years, after which the designers would be available to work on advanced, Pilot-based applications for products, research, and development. The following is a short chronicle of our actual experience:

Jan. 1976: Architectural principles established; work began on design of a machine architecture optimized for Mesa. The Mesa machine extended the Alto architecture in a number of ways, including expansion of the basic machine address from 16 to 32 bits, stack-oriented operation, addition of virtual memory, improved handling of I/O devices, etc.

mid-1976: Mesa language and system becomes operational on the Alto [Geschke *et al*].

Dec. 1976: Pilot Functional Specification (version 1) released to clients. Unfortunately, the system specified in this document looked a lot like a traditional operating system and did not take account of the characteristics of a personal computer, the role of Mesa, or the (dimly perceived) needs of distributed applications.

early 1977: Mesa process facilities [Lampson and Redell] and Pilot file system redefined.

Sept. 1977: Pilot Functional Specification (version 2) released to clients. This set the style of Pilot as it is today and used the Mesa interface language as the primary specification tool [Lauer and Satterthwaite].

Sept. 77–Apr. 78: Design of Pilot kernel implementation.

Apr. 78–Oct. 78: Implementation of Pilot kernel and basic communication facilities.

July 1978: First Dolphin processor delivered to Pilot group (the Dolphin is one of three machines supported by Pilot). This was microcoded to be compatible with the Alto but with virtual memory; some Mesa emulator functions were still coded in BCPL and compiled into Alto machine language.

Oct. 1978: First release of Pilot to clients. This version retained compatiblility with the Alto system in many areas such as disk layout, boot file formats, and Mesa instruction set, even though they were not adequate for our long term needs. The debugger was a modification of the Alto/Mesa debugger and operated in Alto mode. Since at this time there was no established client base, there was very little testing except from our own test programs; consequently, this release was of limited use.

Nov. 78–Dec. 79: Bootstrapped away from the Alto environment. During this time, we eliminated remaining Alto compatibility from Pilot, implemented the Mesa architecture entirely in microcode, supported Pilot disk and boot file formats, etc. CoPilot, the Pilot debugger, became operational as the first major Pilot client. This was a particularly painful period for the implementors.

Dec. 1979: Second release of Pilot to clients. Note that at this time, the Mesa compiler, binder, and other facilities, as well as the program editors, were still based on the Alto. The performance, reliability, and usability of Pilot was grim; as each new client tried it, a new set of disabling bugs was uncovered. Among other things, we discovered that Pilot could read or write the disk at the rate of only one 256-word sector per revolution.

Mar. 1980: Third release of Pilot to clients. This was a cleaned up version of the second release and the first to appear in a product (the Xerox 5700 electronic printing system). We redesigned the file system to solve the disk performance problem and concentrated on the reliability issues with respect to that specific client. Even so, the application programmers had to resort to some unnatural contortions to avoid problem areas in the operating system.

May. 1980: First Dandelion processor available, but with no disk or Ethernet (the Dandelion is the basis of all Xerox 8000 series products).

July. 1980: The Pilot disk utility runs on the Dandelion (with a simulated Ethernet).

Oct. 1980: Fourth release of Pilot to clients. The primary emphasis of this release was to provide the function necessary to support Xerox 8000 series products, the Mesa development environment, and the Cedar project at PARC. Unfortunately, Pilot by this time had become too large and too slow to do any of these very well.

Dec. 1980: Pilot runs on the Dorado processor at PARC (the Dorado is a very high speed, single-user system for the research environment [Dorado]).

Feb. 1981: First Xerox 8000 network system, including Pilot, delivered to a customer.

Apr. 1981: Fifth release of Pilot to clients. The emphasis of this release was to improve the performance and reliability and to reduce the working set size of the system to acceptable levels within the memory available on the various processors. The first version of the Pilot-based Mesa development environment (i.e., compiler, binder, editor, utilities, etc.) was made available to friendly clients.

In April 1981, we received a letter from our most demanding clients in PARC indicating that with the fifth release, Pilot had become the system of choice (on processors that were capable of running either Pilot or Alto software). After five years, Pilot had come of age.

During this whole time, we always had the support of the corporation, even during the hard years when we 'should have' had a nicely running system. Of course, it was necessary to regularly show progress, which we did partly by demonstrating some early but limited applications (to show that Pilot actual worked) and partly by demonstrating Alto-based prototypes of some advanced applications (to show the feasibility of the kinds of things we were aiming for). We also had to fend off the usual kinds of pressures from other parts of the corporation—for example, that we use a commercially available language (e.g., Pascal) rather than Mesa or that we purchase OEM computers rather than design our own architecture.

Selected problems and lessons

In this section, I will recount a few of the problems we encountered and lessons we learned during the development. Most of these we should have avoided, and afterwards there were plenty of people to say they told us so (none of whom had any responsibility for actually releasing the system, of course). Nevertheless, these problems and lessons happened anyway.

Sizes of the system. Table 1 shows some statistics for the five releases of Pilot. Included are the sizes of the Pilot kernel and of the entire system in terms of lines of Mesa code, bytes of object code, and numbers of modules. From the table it can be seen that the Pilot kernel dominated the first release, but by the fifth release it represented barely more than twenty percent of the system. The growth of the total system is accounted for partly by new development and partly by absorbing and converting code which was orginally developed for other systems. Other development, in which major subsystems were completely rewritten or replaced, is not apparent in this kind of summary table, but represents a non-trivial portion of the work that went into all but the first release.

When we began work on Pilot, none of us imagined that we would be developing and managing a system so large. Yet in retrospect this was probably inevitable, given that it was intended to support several major products plus all of our software development, some research, and a number of specialized applications. As we look to the future, it is not clear whether the sizes of either the kernel or the system will stabilize soon or whether they will continue to grow as we respond to new or different needs from our clients.

Working set sizes. Table 1 also shows the size of the working set of the Pilot kernel when supporting 'typical' applications—i.e., the amount of real memory required to hold the virtual memory actively needed by Pilot without thrashing. A working set size is inferred by first artificially restricting the amount of real memory available on the machine and then timing a selected benchmark with a stopwatch. This is repeated for various memory sizes and the results plotted. The total system working set for that benchmark is defined to be at the knee of the curve and can be determined accurately within 1%. Then another experiment is run by setting the real memory size to the working set size, executing the benchmark again (most benchmarks take a few seconds), and taking a memory dump. With some detective work, it is possible to attribute specific pages swapped into real memory to the Pilot kernel, common software and other packages, and the application. This whole exercise is repeated for various benchmarks and for each release of Pilot (and also for each release of critical applications on a given version of Pilot).

An unexpected result was that the content of the working set of the Pilot kernel (i.e., the actual pages swapped in) is nearly constant across all benchmarks. We were also surprised that by April 1980, Pilot exceeded its share of the real memory of our product configurations by nearly a factor of two. In retrospect, of course, we should not have been surprised. Although the requirement for a small working set was in the front of our minds, we had no feedback or reinforcement to achieve it, even at the expense of some features or function. In this sense, the developers suffered from the availability of a good virtual memory system. It is too easy to add more memory to their machines in order to meet critical schedules, even if business reasons precluded such memory in the products. The result was a year of hard work to bring the Pilot memory requirement down to a reasonable level, at some cost in the overall schedule. (An Alto programmer, by contrast, is *forced* to make his applications fit into a non-expandable real memory and address space because he cannot proceed with his own work until they do.) Unfortunately, similar pressures affect many of our clients, and some are caught by the convenience of virtual memory the way we were.

Programmer Productivity. None of the ways that we know of for measuring the productivity of our developers is very statisfying. We do, of course, measure whether or not a release is on time, how many trouble reports are submitted against it, how big it is, etc. But none of these tell us how good the system is. We also have, on occasion, tried to make one traditional measurement of programmer productivity, namely the number of lines of code produced per work-year. There are two types of problems with this measurement, an obvious one and a subtle one. The obvious one is deciding which people and what time to count, so that an effective comparison might be made among organizations and/or programming environments (companies such as Xerox are always interested in such comparisons, even if individual development groups are not). There is also the question of how to count the lines of code, especially when someone is reorganizing or making major modifications to existing modules.

The subtle problem is illustrated by the following observation: in my organization, a group of four or five developers, including a project leader, can specify, design, implement, test, and release a complete system or subsystem of approximately 25,000 lines of Mesa code in twelve months. This includes vacations and holidays and time to attend conferences and seminars, to write professional papers or continue education, to get and train some users or clients to test the system, and to be generally effective members of the organization. In most cases, *if the same people had twice as much time, they could produce as good a system in, perhaps, half that amount of code*—i.e., the extra year yields negative productivity. Thus, one of the conflicts that we have to manage constantly is that between the need to get a system done and working satisfactorily and the desire to make it smaller, faster, easier to use, etc.

Holy wars. In the early days of Pilot development, we got bogged down in a number of basic questions of operating system design and spent a lot of time, energy, and emotions before resolving them. One of these concerned the model of processes and synchronization and whether we should have a facility based on procedures and monitors or a facility based on messages. Each side was firmly entrenched and unable to accept the position of the other; not until we developed the duality hypothesis presented in [Lauer and Needham] were we able to resolve the issue and implement the scheme described in [Lampson and Redell]. Even afterward, our organization bore serious scars from this debate.

Another basic question concerned the kind of access to the file system that Pilot would provide. One alternative was a simple read-write facility with which client programs transfer pages directly between virtual memory and files. The other alternative was a 'mapping' facility, whereby a portion of the file is made the backing store for a portion of virtual memory (for example, as in MULTICS [Bensoussan *et al*]). While this question did not inflame emotions the way the process question did, the proponents of each view felt that the two models were incompatible with each other. Pilot chose the mapping approach, providing a convenience for some but causing headaches for others, particularly those who were trying to convert

TABLE I — SIZES OF PILOT RELEASES

Release	I	II	III	IV	V
Date	Oct 78	Dec 79	Mar 80	Oct 80	Apr 81
Contributors*	~20	~27	~27	~35	~35
Pilot Kernel:					
lines	24K	30K	33K	44K	53K
codebytes	89K	111K	110K	152K	162K
modules	88	102	114	123	135
interfaces	93	132	146	183	204
Pilot System:					
lines	48K	129K	125K	171K	249K
codebytes	211K	508K	508K	754K	1025K
modules	390	420	unknown	530	710
Working Sets (256-word pages)	unknown	unknown	~320	~260	~190

* Most contributors had responsibility for other, non-Pilot software at the same time.

programs from other systems based on the read-write model. Subsequently, the perception grew that perhaps the two approaches are duals of each other, and that a client program structured for one approach might have a natural counterpart of similar performance and complexity for the other approach. However, we never found a duality transformation to support this view. Finally, we realized that neither model excludes the other. In particular, code files and certain data files of limited size are better supported by the mapping model—the swapping characteristics are understood and address space management in virtual memory is more convenient than explicit reading and writing. Large data files with known, high performance access requirements, on the other hand, are better served by the read-write approach—these files are often larger than the virtual address space, and the complexity and overhead of buffer management is worthwhile to achieve the desired performance. Pilot now supports both approaches with consistent interfaces.

Files and transactions. When the Pilot file system was being designed, the question naturally arose about whether or not it should include a transaction facility for crash recovery and atomic updating of files. We did not include one because our experience was limited and we were only just beginning to see results from research in this area at PARC. However, we did provide enough facilities so that a client could build a specialized transaction mechanism on top of the Pilot kernel. Someone built such a facility and also undertook an evangelical mission to persuade people that it offered the solution to all file reliability and recovery problems. As a consequence, several of our major clients came to depend upon the transaction concept, even in cases where it is not appropriate.

Naturally, a transaction facility built on top of Pilot could not have as high performance as one integral with it, and in this case, its interfaces were of a substantially different style than those of Pilot itself. It was also extremely unreliable. Thus we were forced to do a quick implementation of a new transaction facility as part of, and consistent with, the Pilot kernel. This has significantly better performance and is reliable in spite of known bugs; client programs that depended on the previous facility became simpler with the new one. However, the performance is still not good enough for high intensity activities such as data base accesses and updates. At the same time, some of the clients began to realize that even the best transaction facility would offer inappropriate performance for their applications and that their failure modes did not require this generality. For example, in the user level directory facility for the Xerox 8000 series products, it is much better to accept that crashes can occasionally occur in the middle of updates and to rely on a scavenger to restore things from the natural redundancy in the file system.

It is not clear what the future of transactions in Pilot will be, but since we currently satisfy no one in this area, it is likely that the facility will change substantially again.

Virtual memory implementation. In designing the Pilot virtual memory facilities, we recognized the client program would want to manage the address space, map files to pieces of virtual memory, and control (or influence) the swapping between real memory and the backing file. We began with three different interfaces and concepts, but quickly unified them into the single concept of the *space*. The Pilot space is the unit of allocation, mapping, and swapping; spaces can be declared within other spaces, so that the set of all spaces forms a hierarchy according to the containment relation. This was a remarkably simple generalization, but it was hard to implement and is not used by clients. Clients have evolved a style in which almost all mapped spaces are subspaces of the primordial space (all of virtual memory) and only a few of these are further partitioned into subspaces for swapping control. The implementation requires such large data structures for each space that they have to be swappable, and only the very active items are cached in real memory. In the end, several caches were needed and a lot of resident code was written to manage them.

An assumption of the virtual memory implementation is that disk accesses are expensive. Thus, we set up a lot of queues and expected a lot of multiprogramming to overlap computing with disk operation. In fact, disk accesses are cheap on both the Dandelion and Dolphin configurations. If no arm movement is required (this appears to be true most of the time), the computation required to field a page fault, locate the disk address, set up a disk command, receive the disk interrupt, and dispatch the faulted process takes about the same time as the average latency to read a sector. We found that the system could not accept back-to-back requests for adjacent sectors and read them on the same revolution, and thus we had to rewrite the file system to submit single disk requests for runs of pages whenever it could. Even so, it would almost be cheaper to treat the disk as a synchronous device and simply wait until each operation completes without trying to do anything else.

In view of this experience, we are currently reexamining the basic design of the Pilot kernel virtual memory system and will probably make major revisions in both the strategy and the implementation.

Pipes, filters, and streams. One of the strong features of the UNIX system [Ritchie and Thompson] is the uniform facility for input and output which allows separate programs to be connected together by 'pipes.' The UNIX programmer's toolkit includes a large number of simple programs (called 'filters') which perform simple transformations on streams of data, and it is common practice to concatenate a number of these together for a desired result. We thought that Pilot should have a similar facility but consistent with and implementable in Mesa, and so we designed Pilot *streams* (see [Redell *et al*] for an overview).

Unfortunately, although the Pilot stream facility works satisfactorily, it was not very well received and is not widely used by us or by our clients. One reason probably lies in the Mesa model of program modularity. The type-safety and interface language of Mesa make it convenient to design programs with clearly specified procedural interfaces and bind them together with a little bit of control code for a desired result. Thus the Mesa programmer's toolkit consists of a large number of modules of varying complexity and different kinds of control structures. For example, a module which produces a sequence of objects of some abstract type will export a procedure for its clients to access these one at a time. This can be easily bound to another module that expects to get objects of that type, and it is often more flexible than parsing a stream of characters. Thus, Pilot streams are used almost exclusively at the interface with terminals and other systems over industry standard communication lines and protocols. Procedural interfaces are preferred, both within Pilot-based programs and between system elements over the Ethernet.

Comparing Pilot with other operating systems

From the success of the April 1981 release, it is evident that Pilot will take its place among the ranks of mature, evolving operating systems and have a useful life long after its original designers have moved on to other pursuits. However, it did not happen as planned and its development was very different from that of the Alto system. In reflecting upon this, I found it useful to enumerate some of the other operating systems I have known, either from direct contact or from study of the literature or from contact with others. These systems seem to fall into five categories, which I shall first enumerate and then describe.

1. The Alto system. UNIX.
2. IBM's OS/360, MULTICS, Pilot, etc.
3. MTS (the Michigan Terminal System), TENEX, CP-67
4. CAL-TSS, Project SUE, HYDRA, etc.
5. DOS/360, RS-11, etc.

These categories are the result of my personal observations, not of a systematic study, and hence many systems are not listed because I don't know enough about them to classify them. The ordering of the categories is not significant. An important part of the classification is the maturity or success of a system—i.e., acceptance by its clients as a useful, economic tool for helping to get work done, for supporting applications, or for fulfilling other goals. A characteristic of a successful system is that it is accepted by a non-trivial community of users outside its developing organization as a matter of choice and that this community contributes, directly or indirectly, to its further development and growth.

Systems of the first kind. These are everyone's favourite systems. They are successful by our measure and by most other measures (sometimes too successful for their developers). They begin life as small, simple, unambitious systems meant to serve only their authors and perhaps their immediate colleagues. They have limited requirements, usually in the research area. But their excellence and simplicity attract others who are willing to contribute to the further development, additional features, or maintenance responsiblities in exchange for being able to use such systems in their own work. They become successful partly because potential users find it simple and easy to adapt them when needed facilities are missing or ill-

conceived. Systems of the first kind rarely evolve according to any coherent plan agreed to between the implementors and clients, but rather by the willingness to contribute facilities and features as needed. Thus it is not surprising that these systems sometimes appear a little haphazard.

Systems of the second kind. These the planned systems. They are cut 'from whole cloth,' designed and implemented as major projects, directed toward objectives defined by negotiation, often (but not always) aimed at new architectures, meant to satisfy major clients, and built according to schedules and budget constraints imposed for business, contract, or other external reasons.

Some, but not all, of the major operating systems sold by computer manufacturers fall into this category. For example, IBM's OS/360 was conceived as a whole system to satisfy a new, broad marketplace and to incorporate many of the technological achievements of the previous five years, and thus it is a system of the second kind. So is MULTICS, which was built primarily at MIT as a 'real' system based on the previous experimental system CTSS. Pilot is a system of the second kind: it was conceived as a successor to the Alto system and intended to support a range of product, development, and research applications within Xerox over a specified number of years on a new machine architecture.

Not all systems of the second kind are successes. Some notable failures include IBM's TSS/360, the Berkeley Computer Corporation system [Lampson], and the Elliot 503 Mark II system [Hoare]. Each of these was conceived as a major system and a fairly ambitious project, but none survived the patience of its sponsors or clients to reach maturity.

Systems of the third kind. These systems borrow much of their supporting software from an existing system but represent a fundamental change in the way of life. The Michigan Terminal System, for example, provides a paging, terminal-oriented, time-sharing system especially suited for university use on the IBM 360/67 and IBM 370 systems. Most of its compilers, run-time support, subroutine libraries, program development tools, etc., were taken and converted directly from OS/360, but its operating system kernel is dramatically different from OS/360 and it supports new applications that OS/360 never could. (Of course, there are also many OS/360 applications that MTS cannot support.) The obvious motiviation for building a system of this kind is to avoid the time and expense of designing, implementing, and maintaining all new supporting software for the operating system when it is desired only to implement an operating system kernel and some basic functions.

Systems of the fourth kind. While the population of systems of the first three kinds is relatively small, there are many systems of the fourth kind. They make major contributions to the art and science of operating systems but either never reach maturity or never gain acceptance outside the developing organization. For many of them, there is never any serious intent to promote them for widespread use, to support a variety of applications, or to be 'complete.' They are primarily laboratory exercises to support research and to teach about operating system structures, or to support other laboratory work. Note that systems of the first kind begin life as systems of the fourth kind but suffer the calamity of success.

Systems of the fifth kind. Finally, the world is full of small, uninteresting systems which do little to enhance the machines they support and which contribute little to the technology or have little impact on the computing community. These systems come in all sizes, shapes, colors, and prices, and I have nothing interesting to say about them.

This taxonomy is helpful in comparing like with like when we talk about operating systems in a context of which work and which do not. For example, it does not make sense to berate the excessive generality of OS/360, which did succeed, in comparison with the more limited objectives and elegant structures of, say, CAL-TSS or Project SUE, both of which failed to become generally usable. It was observed in [Lampson and Sturgis] that there is much more work involved in making an operating system usable by general programmers than just providing a nice kernel. Similarly, when we ask why the Pilot development was so different from that of the Alto system, it is important to bear in mind the fundamental difference in objectives and ground rules for the two systems. The following were explicitly *not* objectives of the Alto system:

> "This system must satisfy the corporate needs for the next 10-15 years in specific areas."
>
> "A (nearly) complete list and specification of the functions and facilities required of the system over the next five years must be provided before design starts."
>
> "This system must incorporate all of the wonderful lessons of operating system technology from the past five years."
>
> "This system is expected to have more than one hundred users or to be installed in more than one hundred locations."

These or similar objectives did apply, however, to Pilot and to most other systems of the second kind. The developers of the Alto system are chagrined to find that they now have to spend considerable time and energy supporting a system which has several thousand users and supports a wide variety of corporate needs. By contrast, in the Pilot development, we were chagrined to discover that in striving to meet these objectives or our schedule, we often found ourselves unable to apply what we felt was the best technical solution to a problem.

The five-to-seven-year rule

The above classification identifies operating systems in terms of how they were developed and their impact. It separates systems of the second kind, which are willed into existence and operation, from those of the first, third, and fourth kinds, which evolve in a less deliberate manner or as part of some other research or project. While I have no recipe for producing successful systems of the latter kinds, I can offer an hypothesis which, if true, will be useful to anyone setting out to build a system of the second kind: *it takes from five to seven years for a system of the second kind to grow from birth to maturity.* For example, in the systems I enumerated above:

> OS/360 was begun in 1963-1964. Despite early availability and vigourous promotion by its manufacturer, it was not until 1968-1969 that it really gained wide acceptance by its users as a valuable, economic tool.
>
> MULTICS development began in about 1965. After an initial flurry of publications about the system, its design, and its goals, the outside world heard very little from MULTICS-land until the early 1970's, when it started to acquire a following outside MIT and was subsequently adopted by Honeywell, the manufacturer of MULTICS hardware.
>
> The first five years of the life of Pilot were chronicled above. Although the previous skepticism by its users is now changing to enthusiasm, there is still much to be done before Pilot fulfills their expectations.

It also appears that it takes from five to seven years for other operating systems produced by major manufacturers or as major system projects to mature. Let us consider what happens during those years.

First, there is the period of planning and design. This is a time of exceptional optimism, of desire to incorporate the past successes and avoid past mistakes, of a determination to 'do it right.' In the Pilot project, for example, an attitude that prevailed was "the Alto system demonstrated a lot about personal computing; now let's build one for real, to support the company's business. And incidentally, we should build it in Mesa, and build it with virtual memory, and re-engineer the Ethernet, and unify the protocols, improve the file system, etc., etc., etc."

Next comes the initial implementation and first release. There are no operational client programs against which to validate it, so it gets little testing. Anyway, some of the promised function was deferred in the interests of meeting the delivery schedule.

Then comes a period of trying to make the system work at all. Those first hardy users have had to pick their way through minefields of bugs and problems, and some may have become discouraged and turned to other alternatives. However, through perseverance, the problems are solved one by one, and eventually the system seems to work passably, at least for its few active users.

But now it is important to make it work well. It is too slow or too big; it supports too few users/clients; or it fails to match the performance of its predecessors. Promised enhancements and/or deferred functions are abandoned and development is concentrated

on very simple matters. During these two phases, client or user participation is essential, although painful. Without the appropriate feedback from others who are trying to use the system for non-trivial reasons (other than its own development), the implementors do not have enough information to guide their work and identify problems in performance, style, or function.

Finally, if the sponsor has not lost patience, there is a period of evolving expectations about the new system. Clients realize that it is not the same as a previous one and that their old models of performance, behaviour, and usage need to be modified to take advantage of the new facilities and the new constraints. Some functions or facilities of the new system may never work as well or as fast as the corresponding ones of the old, and programs converted from the old may appear sluggish in the new environment. This is the beginning of the period of 'community involvement' with the operating system, during which clients learn how to live within the framework defined by that system and how to contribute to its further success and growth.

The five-to-seven-year rule for systems of the second kind is a strong generalization from weak evidence. I know of no analysis which might lead to it as a conclusion. I would even like to see it disproved (and to learn how to disprove it at will). Nevertheless, from my own experience and from observations of the experience of others, it seems to be true, at least most of the time. It seems to apply both to the professional system designers and programmers who populate the industry and to the elite corps of computer scientists from the academic/research community. Even people with impressive credentials fail in their attempts to build systems in less time, and very few succeed. We believed from the beginning that we could do better in the development of Pilot.

Both in casual conversations and detailed analyses of the successes and failures of systems of the second kind, the same terms keep recurring: that the systems are "too ambitious" and/or "too general." Hoare used these terms in his Turing Lecture [Hoare]. I can remember as a graduate student that my colleagues and I would sneer at the manufacturer-supplied operating systems (IBM's OS/360, Univac's Exec VIII, Burroughs' MCP, and all the rest) in exactly the same terms. In almost any cocktail conversation about a system in trouble or one which the users find unsatisfactory, criticism is focused on the generality or amibitousness of the project goals. In my discussions with the original implementors of the kernel of the Alto system, the same terms were used again: "if only we had set more limited goals for Pilot by concentrating on, say, real memory requirements rather than on features, we would have produced a nice, well-performing system in two or three years, just the way they did." However, that begs the question: we *did* concentrate on real memory usage, execution speed, simple structure, and all of the other things that are important in making a successful design. We made task lists of things to do, problems to solve, features to support; we assigned priorities and worked on first things first; we parried requests for yet more features or complexity; we ignored unreasonable constraints imposed externally and let our computer science wisdom prevail. It still took us five years, and our critics at the time still worried about the grandiosity of our system.

I suspect that there is something about the ground rules of projects like Pilot, the Berkeley Computer Corporation system, MULTICS, and other systems of the second kind that makes it difficult or impossible to plan or carry out projects the way we do for systems of the other kinds. (Note that by definition, systems of the other kinds cannot be too ambitious or general: either they succeed on their merits, meaning that they have exactly the right blend of generality and simplicity, or they were never meant to fulfill the kinds of goals that a system of the second kind is.) Part of it is, no doubt, in the way that projects are sponsored. Systems of the first, third, and fourth kinds are usually financed as part of some other project or research.

But systems of the second kind are investments. As such they are subject to the kinds of review of planning, budgeting, scheduling, and scrutiny that the sponsor needs to confirm continuing support (neither the Alto system, MTS, or HYDRA, for example, were ever subject to this kind of review). Furthermore, investments in system development are still so risky these days that most sponsors would rather purchase a satisfactory system, if available, than build one. Thus, by definition, the new system has to be more ambitious and/or more general than its predecessors. If it is necessary to finance a new system, then the sponsors and/or their clients feel entitled to some voice in the facilities, features, style, character, or other attributes of the system. I.e., the system designers do not get to go off alone to build the system of their dreams, and the five-to-seven-year rule prevails.

Summary

I have been a member of the Pilot project since early 1977 and have managed it in its later years. Coming from an academic and research background, I have been somewhat surprised at what it has been possible for us to do and also at what we have not been able to do. I think it is important for people to write about these things occasionally because we too often concentrate on the objects we are creating and not enough on the process of creating them. The world of programming methodology and structured programming is devoted to helping us achieve perfection in the systems we design—an important goal but somewhat at odds with the need to get something done. Pilot had to be delivered and work well enough, despite the fact that we never had time to make it perfect or even as small and as simple as we would have liked. It is helpful to treat the major system as an organism itself, with a life cycle and a personality and characteristics derived from the organizations that build, use, and sponsor it. The successful ones usually outlive the interests and participation of their implementors, and they captivate or even dominate the professional lives and interests of many other people.

I have believed in the five-to-seven-year rule for at least a decade and have not found much evidence against it. Yet this is not proper science, so I challenge graduate students and researchers in the operating system field to conduct systematic studies about how systems are conceived and born and which ones grow, mature, and lead productive lives. This would be a study partly of technology but partly of the sociology and dynamics of system development, and it would teach us how to build better, simpler, less ambitious systems more predictably.

The classification of operating systems into five kinds came about as I tried to compare Pilot with other systems and see where the five-to-seven-year rule applied and where it did not. There is definitely a qualitative difference between the kind of development we carried out and the kind that I have watched or been associated with in universities and research laboratories. Thus it is not surprising that there is a difference in character between the kinds of systems that emerge from these activities. I do not know whether this classification is 'right,' so again I challenge research students to explore the field of operating systems from this point of view, making systematic studies to help us understand how we do better at building them.

Finally, a word of advice to designers, implementors, sponsors, and users: if you are involved with a new, challenging system planned and cut out of whole cloth and meant as a service, not as an experiment, but intended to stretch our horizons and expectations—i.e., a system of the second kind—then have patience. I have not yet seen anyone who has been able to build one as quickly and as well as he thought he could.

Acknowledgements

Approximately forty people in Xerox System Development Department have made contributions to Pilot, all of them valuable. In addition, we gained great benefit from close location to and many conversations with our colleagues at PARC. The number of people deserving acknowledgement is far too great to list here. However, I wish to specially acknowledge Dr. David E. Liddle, who created and sustained SDD and the Pilot project as part of it.

References

[Belady and Lehman]
Belady, L. A., and Lehman, M. M., 'A model of large program development,' *IBM System Journal*, no. 3, 1976.

[Bensoussan *et al*]
Bensoussan, A., Clingen, C. T., and Daley, R. C., 'The MULTICS Virtual Memory: Concepts and Design,' *Communications of the ACM*, vol 15, no 5, May 1972, pp 308-318.

[Deutsch and Taft]
Deutsch, L. P. and Taft, E. A., 'Requirements for an Experimental Programming Environment,' report # CSL-80-10, Xerox Corporation, Palo Alto Research Center, Palo Alto, 1980.

[Dorado]
The Dorado: A High-performance Personal Computer, Three Papers, Technical Report CSL-81-1, Xerox Palo Alto Research Center, Palo Alto, California, January 1981.

[Geschke *et al*]
Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., 'Early Experience with Mesa.' *Communications of the ACM*, vol. 20, no. 8, August 1977

[Hoare]
Hoare, C. A. R., 'The Emperor's Old Clothes,' (1980 ACM Turing Award Lecture), *Communications of the ACM*, vol. 24, no. 2, February 1981.

[Lampson]
Lampson, B. W., 'Dynamic protection structures,' *Proceedings of the AFIPS Fall Joint Computer Conference*, 1969, pp 27-38. (Note: The Berkeley Computer Corporation was a widely publicized venture by a number of respected computer scientists to build a major time-sharing system and utility in 1968-1970. I can find no references to it in the literature except this one, which is mostly about the operating system structure.)

[Lampson and Redell]
Lampson, B. W. and Redell, D. D., 'Experience with Processes and Monitors in Mesa,' *Communications of the ACM*, vol. 23, no. 2, February 1980.

[Lampson and Sturgis]
Lampson, B. W. and Sturgis, H. E., 'Reflections on an Operating System Design,' *Communications of the ACM*, vol. 19, no. 5, May 1976.

[Lauer and Needham]
Lauer, H. C. and Needham, R. M., 'On the Duality of Operating System Structures,' *Proc. Second International Symposium on Operating Systems*, IRIA, Oct. 1978, reprinted in *Operating Systems Review*, vol. 13, no 2, April 1979, pp 3-19.

[Lauer and Satterthwaite]
Lauer, H. C. and Satterthwaite, E. H., 'Impact of Mesa on System Design,' *Proceedings of Fourth International Conference on Software Engineering*, Munich, September 1979, pp 174-182.

[Mitchell *et al*]
Mitchell, J. G., Maybury, W. and Sweet, R., *Mesa Language Manual*, report # CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, 1979.

[Redell *et al*]
Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C. McJones, P. R., Murray, H. G., Purcell, S. C., 'Pilot: An Operating System for a Personal Computer,' *Communications of the ACM*, vol. 23, no. 2, February 1980.

[Ritchie and Thompson]
Ritchie, D. M. and Thompson, K., 'The UNIX Time-Sharing System,' *Communications of the ACM*, vol. 17, no. 7, July 1974.

[Sproull and Lampson]
Sproull, R. F. and Lampson, B. W., 'An open operating system for a single-user machine,' *Proceeding of the Seventh Symposium on Operating System Principles*, Asilomar, December 1979.