THE EVOLUTION OF OPERATING SYSTEMS*

PER BRINCH HANSEN

(2000)

The author looks back on the first half century of operating systems and selects his favorite papers on classic operating systems. These papers span the entire history of the field from the batch processing systems of the 1950s to the distributed systems of the 1990s. Each paper describes an operating system that combines significant ideas in an elegant way. Most of them were written by the pioneers who had the visions and the drive to make them work. The author summarizes each paper and concludes that operating systems are based on a surprisingly small number of ideas of permanent interest.

INTRODUCTION

The year 2000 marks the first half century of computer operating systems. To learn from the pioneers of the field, I have selected *my favorite papers on classic operating systems*. These papers span the entire history of the field from the batch processing systems of the 1950s to the distributed systems of the 1990s. I assume that you already know how operating systems work, but not necessarily how they were invented.

The widespread use of certain operating systems is of no interest to me, since it often has no obvious relationship to the merits (or flaws) of these systems. To paraphrase G. H. Hardy (1969), *Beauty is the first test: there is no permanent place in the world for ugly ideas.*

Let me explain how I made my choices:

*P. Brinch Hansen, The evolution of operating systems. In *Classic Operating Systems: From Batch Processing to Distributed Systems*, P. Brinch Hansen, Ed.

- *Each paper describes an operating system that combines significant ideas in an elegant way.*
- I picked mostly papers written by the pioneers who had the visions and the drive to make them work. I also included a few elegant systems that broke no new ground, but convincingly demonstrated the best ideas known at the time.
- I would have preferred short papers that were a pleasure to read. However, as Peter Medawar (1979) truthfully has said, "Most scientists do *not* know how to write." In some cases, I had to settle for papers in which "clarity has been achieved and the style, if not graceful, is at least not raw and angular."

The evolution of operating systems went through seven *major phases* (Table 1). Six of them significantly changed the ways in which users accessed computers through the open shop, batch processing, multiprogramming, timesharing, personal computing, and distributed systems. In the seventh phase the foundations of concurrent programming were developed and demonstrated in model operating systems.

Table 1 Classic Operating Systems

Major Phases		*Operating Systems*	
I	Open Shop	1	IBM 701 open shop (1954)
II	Batch Processing	2	BKS system (1961)
III	Multiprogramming	3	Atlas supervisor (1961)
		4	B5000 system (1964)
		5	Exec II system (1966)
		6	Egdon system (1966)
IV	Timesharing	7	CTSS (1962)
		8	Multics file system (1965)
		9	Titan file system (1972)
		10	Unix (1974)
V	Concurrent Programming	11	THE system (1968)
		12	RC 4000 system (1969)
		13	Venus system (1972)
		14	Boss 2 system (1975)
		15	Solo system (1976)
		16	Solo program text (1976)
VI	Personal Computing	17	OS 6 (1972)
		18	Alto system (1979)
		19	Pilot system (1980)
		20	Star user interface (1982)
VII	Distributed Systems	21	WFS file server (1979)
		22	Unix United RPC (1982)
		23	Unix United system (1982)
		24	Amoeba system (1990)

I chose 24 papers on *classic operating systems* with reasonable confidence. With so many contenders for a place in operating systems history, you will probably disagree with some of my choices. For each phase, I attempted to include a couple of early *pioneering* systems followed by a few of the later systems. Some of the latter could undoubtedly have been replaced by other equally *representative* systems. Although I left out a few *dinosaurs*, I hope that there are no *missing links*.

The publication dates reveal that the 1960s and 1970s were the vintage years of operating systems; by comparison, the 1980s and 1990s seem to have yielded less. This is to be expected since the early pioneers entered the field *before* the best ideas had been invented.

The selected papers show that operating systems are based on a surprisingly small number of *ideas of permanent interest* (Table 2). The rest strike me as a fairly obvious consequence of the main themes.

Table 2 Fundamental Ideas

	Major Phases	*Technical Innovations*
I	Open Shop	The idea of operating systems
II	Batch Processing	Tape batching First-in, first-out scheduling
III	Multiprogramming	Processor multiplexing Indivisible operations Demand paging Input/output spooling Priority scheduling Remote job entry
IV	Timesharing	Simultaneous user interaction On-line file systems
V	Concurrent Programming	Hierarchical systems Extensible kernels Parallel programming concepts Secure parallel languages
VI	Personal Computing	Graphic user interfaces
VII	Distributed Systems	Remote servers

The following is a brief resume of the selected papers with some background information. Throughout I attempt to balance my own views by quoting both critical and complimentary comments of other researchers.

PART I OPEN SHOP

1 IBM 701 Open Shop

We begin the story of operating systems in 1954 when computers had *no operating systems* but were operated manually by their users:

> *The IBM 701 computer at the General Motors Research Laboratories.*
> George F. Ryckman (1983)

George Ryckman remembered the gross inefficiency of the *open shop* operation of IBM's first computer, the famous 701:

> Each user was allocated a minimum 15-minute slot, of which time he usually spent 10 minutes in setting up the equipment to do his computation ... By the time he got his calculation going, he may have had only 5 minutes or less of actual computation completed—wasting two thirds of his time slot.

The cost of the wasted computer time was $146,000 per month—in 1954 dollars!

John McCarthy (1962) made a similar remark about the TX-0 computer used in open shop mode at MIT. He added:

> If the TX-0 were a much larger computer, and if it were operated in the same manner as at present, the number of users who could be accommodated would still be about the same.

PART II BATCH PROCESSING

Surely, the greatest leap of imagination in the history of operating systems was the idea that computers might be able to schedule their own workload by means of software.

The early operating systems took drastic measures to reduce idle computer time: the users were simply removed from the computer room! They were now asked to prepare their programs and data on punched cards and submit them to a computing center for execution. The open shop had become a *closed shop*.

Now, card readers and line printers were too slow to keep up with fast computers. This bottleneck was removed by using fast tape stations and small satellite computers to perform *batch processing*.

Operators collected decks of punched cards from users and used a satellite computer to input a batch of jobs from punched cards to a magnetic tape. This tape was then mounted on a tape station connected to a main computer. The jobs were now input and run one at a time in their order of appearance on the tape. The running jobs output data on another tape. Finally, the output tape was moved to a satellite computer and printed on a line printer. While the main computer executed a batch of jobs, the satellite computers simultaneously printed a previous output tape and produced the next input tape.

Batch processing was severely limited by the sequential nature of magnetic tapes and early computers. Although tapes could be rewound, they were only efficient when accessed sequentially. And the first computers could only execute one program at a time. It was therefore necessary to run a complete batch of jobs at a time and print the output in *first-come, first-served* order.

To reduce the overhead of tape changing, it was essential to batch many jobs on the same tape. Unfortunately, large batches greatly increased service times from the users' point of view. It would typically take hours (or even a day or two) before you received the output of a single job. If the job involved a program compilation, the only output for that day might be an error message caused by a misplaced semicolon!

The *SHARE operating system* for the IBM 709 was an early batch processing system described by Bratman (1959).[1] Unfortunately, this short paper does not explain the basic idea succinctly, concentrating instead on the finer points of different job types.

2 BKS System

Much to my surprise, it was difficult to find a well-written paper about any early batch processor. Bob Rosin (2000) explained why such papers were rare:

> First, these systems were "obvious" and could be understood in minutes from reading a manual. Second, there were very few different kinds of computers, and the community of system programmers was similarly small. At least in the United States, almost everyone who wanted to know about these systems could and did communicate directly with their authors.

[1] The SHARE system is briefly mentioned at the end of Article 1.

The paper I chose describes the *BKS system* which occupied 2,688 words only out of a 32,768-word memory. In comparison to this system, later operating system designers have mostly failed in their search for simplicity!

> *The BKS system for the Philco-2000.*
> Richard B. Smith (1961)

PART III MULTIPROGRAMMING

In the 1960s large core memories, secondary storage with random access, data channels, and hardware interrupts changed operating systems radically. Interrupts enabled a processor to simulate concurrent execution of multiple programs and control simultaneous input/output operations. This form of concurrency became known as *multiprogramming*.

Christopher Strachey (1959) wrote the first seminal paper on multiprogramming. Fifteen years later, Strachey (1974) wrote to Donald Knuth:

> The paper I wrote called "Time-sharing in Large Fast Computers" was read at the first (pre IFIP) conference in 1960 [sic]. It was mainly about multiprogramming (to avoid waiting for peripherals) ... I did not envisage the sort of console system which is now so confusingly called time-sharing.

Multiprogramming and secondary storage made it possible to build operating systems that handled a continuous stream of input, computation, and output on a single computer using drums (or disks) to hold large buffers. This arrangement was called *spooling*.[2]

Since spooling required no tapes, there was no longer any overhead of tape mounting (unless user programs processed their own data tapes). Large random access buffers made it feasible to use *priority scheduling* of jobs, such as shortest-job-next (instead of first-come, first-served).

Incidentally, time-sharing may have made input spooling obsolete, but many organizations still use output spooling for printers shared by clusters of personal computers.

The term *batch processing* is now often used as a synonym for spooling. This is somewhat misleading since jobs are no longer grouped into batches. In 2000 this form of "batch processing" was still being used to run jobs through the Cray machines at the Pittsburgh Supercomputing Center.[3]

[2] *Spooling* is an acronym for "Simultaneous Peripheral Operation On-Line."

[3] See the Web site at http://www.psc.edu/machines/cray/j90/access/batch.html.

3 Atlas Supervisor

The use of multiprogramming for spooling was pioneered on the *Atlas* computer at Manchester University in the early 1960s:

> *The Atlas supervisor.*
> Tom Kilburn, R. Bruce Payne and David J. Howarth (1961)

Tom Kilburn felt that "No other single paper on the Atlas System would be a better choice" (Rosen 1967). This amazing paper explains completely new ideas in readable prose without the use of a single figure!

Atlas also introduced the concept of *demand paging* between a core memory of 16 K and a drum of 96 K words:

> The core store is divided into 512 word "pages"; this is also the size of the fixed blocks on drums and magnetic tapes. The core store and drum store are addressed identically, and drum transfers are performed automatically.
>
> A program addresses the combined "one-level store" and the supervisor transfers blocks of information between the core and drum store as required; the physical location of each block of information is not specified by the program, but is controlled by the supervisor.

Finally, Atlas was the first system to exploit *supervisor calls* known as "extracodes":

> Extracode routines form simple extensions of the basic order code, and also provide specific entry to supervisor routines.

The concepts of spooling, demand paging, and supervisor calls have influenced operating systems to this day. The Atlas supervisor has been called "the first recognisable modern operating system" (Lavington 1980). It is, I believe, the most significant breakthrough in the history of operating systems.

The virtual machine described in most published papers on Atlas is the one that runs user programs. The chief designer of the supervisor, David Howarth (1972a), pointed out that this virtual machine "differs in many important respects from the 'virtual machine' used by the supervisor itself." These differences complicated the design and maintenance of the system.

The later RC 4000 multiprogramming system had the same weakness (Brinch Hansen 1973).

By 1960 high-level programming languages, such as Fortran, Algol 60 and Cobol, were already being used for user programming. However, operating systems, such as the Atlas supervisor, were still programmed in machine language which was both difficult to understand and error-prone.

4 B5000 Master Control Program

The *Burroughs B5000* computer had *stack instructions* for efficient execution of sequential programs written in Algol 60 (Barton 1961). For this purpose the B5000 was truly a revolutionary architecture. The Burroughs group published only a handful of papers about the B5000 system, including

> *Operating system for the B5000.*
> Clark Oliphint (1964)

Admittedly, this brief paper does not do justice to the significant accomplishments of this pioneering effort. Organick (1973) and McKeag (1976a) provide an abundance of detailed information.

Burroughs used its own variants of Algol to program the *B5000 Master Control Program*, which supported both *multiprogramming* and *multiprocessing* of user programs.

The system used *virtual memory* with automatic transfers of data and program segments between primary and secondary storage (MacKenzie 1965). A typical system could run on the order of 10 user jobs at a time. About once a day, *thrashing* would occur. This was not detected by the system. The operator was expected to notice any serious degradation of performance and restart the system (McKeag 1976a).

Unfortunately the programming languages of the early 1960s offered no support for concurrent programming of operating systems. High-level languages for concurrent programming were only invented in the 1970s.

At the time the only option open to Burroughs was to adopt an extremely dangerous short-cut: The B5000 operating system (and its successors) were written in *extended Algol* that permitted systems programs to access the whole memory as an array of (unprotected) machine words.[4] This programming trick effectively turned extended Algol into an assembly language with

[4]It was sometimes referred to as "Burroughs overextended Algol."

an algorithmic notation. The dangers of using such an implementation language were very real.

Roche (1972) made the following comment about a B5500 installation in which the user was permitted to program in extended Algol:

> This allows the use of stream procedures, a means of addressing, without checks, an absolute offset from his data area. Mis-use and abuse of these facilities by ill-informed or over-ambitious users could, and often did, wreck the system.

Organick (1973) pointed out that the termination of a task in the B6700 operating system might cause its offspring tasks to lose their stack space! The possibility of a program being able to delete part of its own stack is, of course, completely at variance with our normal expectations of high-level languages, such as Fortran, Algol, or Pascal.

According to Rosin (1987), "High-level languages were used exclusively for both customer programming and systems programming" of the B5000. Similar claims would be made for later operating systems programmed in intermediate-level languages, including Multics (Corbató 1965), OS 6 (Stoy 1972), and Unix (Ritchie 1974). However, in each case, system programmers had extended sequential programming languages with unsafe features for low-level programming.

There is no doubt about the practical advantages of being able to program an operating system in a language that is at least partly high-level. However, a programming notation that includes machine language features is, per definition, *not* a *high-level* language.

Since nobody could expect Burroughs to use concepts that had not yet been invented, the above criticism does not in any way diminish the contribution of a bold experiment: the first tentative step towards writing operating systems in a high-level language.

5 Exec II System

The operation of early batch processing systems as closed shops set a precedence that continued after the invention of multiprogramming. The only system that boldly challenged the prevailing wisdom was the *Exec II* operating system for the Univac 1107 computer at Case Western Reserve University:

> *Description of a high capacity, fast turnaround university computing center.*
> William C. Lynch (1966)

Bill Lynch writes that

> The [Case Computing] Center employs an open-shop type philosophy that appears to be unique among large scale installations. This philosophy leads to turnaround times which are better by an order of magnitude than those commonly being obtained with comparable scale equipment.

Exec II was designed to run one job at a time using two fast drums for input/output spooling. The system was connected to several card reader/line printer groups. When a user inserted a deck in any reader, the cards were immediately input. The user would then remove her cards and proceed to the line printer where the output of the job would appear shortly.

85% of the jobs required less than a minute of computer time. A student was often able to run a small job, repunch a few cards, and run the job again in less than five minutes. The system typically ran 800 jobs a day with a processor utilization of 90%. Less than 5% of the jobs used magnetic tapes. Users were also responsible for mounting and identifying their own tapes.

Occasionally, the phenomenal success of Exec II was limited by its policy of selecting the shortest job and running it to completion (or time limit):

> when a long running program is once started, no other main program is processed until the long running job is finished. Fortunately this does not happen often but when it does, it ruins the turnaround time. It appears to be desirable to have ... an allocation philosophy which would not allow the entire machine to be clogged with one run, but would allow short jobs to pass the longer ones. Such a philosophy, implemented with conventional multiprogramming techniques, should remove this difficulty.

> The scheduling algorithm currently being used leaves something to be desired. It selects jobs (within classes) strictly on the basis of shortest time limit. No account is taken of waiting time. As a result, a user with a longer run can be completely frozen out by a group of users with shorter runs.

The system also supported *remote job entry* through modems and telephone lines. Exec II came remarkably close to realizing the main advantages of time-sharing (which was still in the future): remote access to a shared computer with fast response at reasonable cost.

Did I forget to mention that Exec II did all of that in a memory of 64 K words?

Exec II demonstrated that the most important ingredient of radically new ideas is often the rare intellectual ability to look at existing technology from a new point of view. (That would also be true of the first timesharing systems.)

6 Egdon System

The Egdon system deserves to be recognized as a classic operating system:

> *The Egdon system for the KDF9.*
> David Burns, E. Neville Hawkins, D. Robin Judd and John L. Venn (1966)

It ran on a KDF computer with 32 K words of core memory, eight tape units and a disk of 4 M words. The disk was mainly used to hold library routines, system programs and work space for the current user program.

The system combined traditional tape batching with spooling of tape input/output. The system automatically switched between two input tapes. Jobs were copied from a card reader onto one of the tapes. When that tape was full, the system rewound it and executed one job at a time. At the same time, the system started filling the second input tape. In a slightly more complicated way, the system switched between a third tape that received output from the running program and a fourth one that was being printed.

The Egdon system was completed on time in 15 months with a total effort of 20 person-years. The authors attribute their sense of urgency to the existence of clear objectives from the start and a stiff penalty clause for late delivery. This "meant that a clear definition was arrived at quickly and changes were kept to a minimum."

PART IV TIMESHARING

John McCarthy proposed the original idea of timesharing at MIT in an unpublished memorandum dated January 1, 1959:

> I want to propose an operating system for [the IBM 709] that will substantially reduce the time required to get a problem solved on the machine ... The only way quick response can be provided at bearable cost is by time-sharing. That is, the computer must attend to other customers while one customer is reacting to some output.

> I think the proposal points to the way all computers will be operated in the future, and we have a chance to pioneer a big step forward in the way computers are used.

In the spring of 1961 he explained his visionary thinking further (McCarthy 1962):

> By a time-sharing computer system I shall mean one that interacts with many simultaneous users through a number of remote consoles. Such a system will look to each user like a large private computer...When the user wants service, he simply starts typing in a message requesting the service. The computer is always ready to pay attention to any key that he may strike.
>
> Because programs may ... do only relatively short pieces of work between human interactions, it is uneconomical to have to shuttle them back and forth continually to and from secondary storage. Therefore, there is a requirement for a large primary memory ... The final requirement is for secondary storage large enough to maintain the users' files so that users need not have separate card or tape input-output units.

It would be difficult to summarize the essence of timesharing more concisely. But to really appreciate McCarthy's achievement, we need to remind ourselves that when he outlined his vision nobody had ever seen a timesharing system. A truly remarkable and revolutionary breakthrough in computing!

7 CTSS

Fernando Corbató at MIT is generally credited with the first demonstration of timesharing:

> *An experimental time-sharing system.*
> Fernando Corbató, Marjorie Merwin-Daggett and Robert C. Daley (1962)

A quarter of a century later, Rosin and Lee (1992) interviewed Corbató about this system, known as *CTSS*:

> By November 1961 we were able to demonstrate a really crude prototype of the system [on the IBM 709]. What we had done was [that]

> we had wedged out 5K words of the user address space and inserted a little operating system that was going to manage the four typewriters. We did not have any disk storage, so we took advantage of the fact that it was a large machine and we had a lot of tape drives. We assigned one tape drive per typewriter.
>
> The paper said we were running on the [IBM] 7090, but we in fact had not got it running yet.

Corbató agreed that

> the person who deserves the most credit for having focussed on the vision of timesharing is John McCarthy ... [He] wrote a very important memo where he outlined the idea of trying to develop a timesharing system.

In September 1962 McCarthy working with Bolt Beranek and Newman demonstrated a well-engineered small timesharing system on a PDP 1 computer with a swapping drum (McCarthy 1963). However, by then the prototype of CTSS running on inadequate hardware had already claimed priority as the *first* demonstration of timesharing.

In the summer of 1963 CTSS was still in the final stages of being tested on a more appropriate IBM 7090 computer equipped with a disk (Wilkes 1985). Eventually this version of CTSS became recognized as the first large-scale timesharing system to be offered to a wide and varied group of users (Crisman 1965).

8 Multics File System

In 1964 MIT started the design of a much larger timesharing system named *Multics*. According to Corbató (1965):

> The overall design goal of the Multics system is to create a computing system which is capable of comprehensively meeting almost all of the present and near-future requirements of a large computer service installation.

By the fall of 1969 Multics was available for general use at MIT. The same year, Bell Labs withdrew from the project (Ritchie 1984):

> To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier.

Multics was never widely used outside MIT. In hindsight, this huge system was an overambitious dead end in the history of operating systems. It is a prime example of the *second-system effect*—the temptation to follow a simple, first system with a much more complicated second effort (Brooks 1975).

However, CTSS and Multics made at least one lasting contribution to operating system technology by introducing the first *hierarchical file systems*, which gave all users instant access to both private and shared files:

> *A general-purpose file system for secondary storage.*
> Robert C. Daley and Peter G. Neumann (1965)

Note that this paper was a *proposal* only published several years before the completion of Multics.

9 Titan File System

The *Titan system* was developed and used at Cambridge University (Wilson 1976). It supported timesharing from 26 terminals simultaneously and was noteworthy for its simple and reliable file system:

> *File integrity in a disc-based multi-access system.*
> A. G. Fraser (1972)

A 128 M byte disk held about 10,000 files belonging to some 700 users. It was the first file system to keep passwords in scrambled form to prevent unauthorized retrieval and misuse of them. Users were able to list the actions that they wished to authorize for each file (*execute, read, update,* and *delete*.) The system automatically made copies of files on magnetic tape as an insurance against hardware or software errors.

According to A. G. Fraser (Discussion 1972):

> The file system was designed in 1966 and brought into service in March 1967. At that time there were very few file systems designed for on-line use available, the most influential at the time being the Multics proposal.

J. Warne added that "This is a very thorough paper, so precise in detail that it is almost a guide to implementation" (Discussion 1972). Fraser's paper makes it clear that it is a nontrivial problem to ensure the integrity of user files in a timesharing system.

File integrity continues to be of vital importance in *distributed systems* with shared file servers.

10 Unix

In 1969 Dennis Ritchie and Ken Thompson at Bell Labs began trying to find an alternative to Multics. By the end of 1971, their *Unix* system was able to support three users on a PDP 11 minicomputer. Few people outside Bell Labs knew of its existence until 1973 when the Unix kernel was rewritten in the *C* language. Nothing was published about Unix until 1974:

> *The Unix time-sharing system.*
> Dennis M. Ritchie and Ken Thompson (1974)

Unix appeared at the right time (Slater 1987):

> The advent of the smaller computers, the minis—especially the PDP-11—had spawned a whole new group of computer users who were disappointed with existing operating software. They were ready for Unix. Most of Unix was not new, but rather what Ritchie calls "a good engineering application of ideas that had been around in some form and [were now] made convenient to use."

By the mid-1980s Unix had become the leading standard for timesharing systems (Aho 1984):

> In the commercial world there are 100,000 Unix systems in operation ... Virtually every major university throughout the world now uses the Unix system.

A superior tool like Unix often feels so natural that there is no incentive for programmers to look for a better one. Still, after three decades, it can be argued that the widespread acceptance of Unix has become an obstacle to further progress. Stonebraker (1981), for example, describes the problems that Unix creates for database systems.

PART V CONCURRENT PROGRAMMING

By the mid-1960s operating systems had already reached a level of complexity that was beyond human comprehension. In looking back Bill Lynch (1972) observed that:

> Several problems remained unsolved within the Exec II operating system and had to be avoided by one *ad hoc* means or another. The problem of deadlocks was not at all understood in 1962 when the system was designed. As a result several annoying deadlocks were programmed into the system.

From the mid-1960s to the mid-1970s computer scientists developed a conceptual basis that would make operating systems more understandable. This pioneering effort led to the discovery of *fundamental principles of concurrent programming*. The power of these ideas was demonstrated in a handful of influential *model operating systems*.

11 THE Multiprogramming System

The conceptual innovation began with Edsger Dijkstra's famous *THE system*:

> *The structure of the THE multiprogramming system.*
> Edsger W. Dijkstra (1968a)

This was a spooling system that compiled and executed a stream of Algol 60 programs with paper tape input and printer output. It used software-implemented demand paging between a 512 K-word drum and a 32 K-word memory. There were five user processes and 10 input/output processes, one for each peripheral device. The system used *semaphores* for process synchronization and communication.

This short paper concentrates on Dijkstra's most startling claim:

> We have found that it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and its implementation can admit exhaustive testing. The only errors that showed up during testing were trivial coding errors ... the resulting system is guaranteed to be flawless.

In Brinch Hansen (1979) I wrote:

> Dijkstra's multiprogramming system also illustrated the conceptual clarity of *hierarchical structure*. His system consisted of several program layers which gradually transform the physical machine into a more pleasant abstract machine that simulates several processes which share a large, homogeneous store and several virtual devices. These program layers can be designed and studied one at a time.

The system was described in more detail by Habermann (1967), Dijkstra (1968b, 1971), Bron (1972) and McKeag (1976b).

Software managers continue to believe that software design is based on a magical discipline, called "software engineering," which can be mastered by average programmers. Dijkstra explained that the truth of the matter is simply that

> the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

In my opinion, the continued neglect of this unpopular truth explains the appalling failure of most software which continues to be inflicted on computer users to this day.

12 RC 4000 Multiprogramming System

In 1974 Alan Shaw wrote:

> There exist many approaches to multiprogramming system design, but we are aware of only two that are *systematic* and *manageable* and at the same time have been *validated* by producing real working operating systems. These are the hierarchical abstract machine approach developed by Dijkstra (1968a) and the nucleus methods of Brinch Hansen (1969) ... The nucleus and basic multiprogramming system for the RC 4000 is one of the most elegant existing systems.

The *RC 4000 multiprogramming system* was not a complete operating system, but a small *kernel* upon which operating systems for different purposes could be built in an orderly manner:

> *RC 4000 Software: Multiprogramming System.*
> Per Brinch Hansen (1969)

The kernel provided the basic mechanisms for creating a *tree of parallel processes* that communicated by messages. It was designed for the RC 4000 computer manufactured by Regnecentralen in Denmark. Work on the system began in the fall of 1967, and a well-documented reliable version was running in the spring of 1969.

Before the RC 4000 multiprogramming system was programmed, I described the design philosophy which drastically generalized the concept of an operating system (Brinch Hansen 1968):

> The system has no built-in assumptions about program scheduling and resource allocation; it allows any program to initiate other programs in a hierarchal manner.[5] Thus, the system provides a general frame[work] for different scheduling strategies, such as batch processing, multiple console conversation, real-time scheduling, etc.

In retrospect, this radical idea was probably the most important contribution of the RC 4000 system to operating system technology. If the kernel concept seems obvious today, it is only because it has passed into the general stock of knowledge about system design. It is now commonly referred to as the principle of *separation of mechanism and policy* (Wulf 1974).

The RC 4000 system was also noteworthy for its *message communication*. Every communication consisted of an exchange of a message and an answer between two processes. This protocol was inspired by an early decision to treat peripheral devices as processes, which receive input/output commands as messages and return acknowledgements as answers. In distributed systems, this form of communication is now known as *remote procedure calls*.

The system also supported *nondeterministic communication* which enabled processes to inspect and receive messages in arbitrary (instead of first-come, first-served) order. This flexibility is necessary to program a process that implements priority scheduling of a shared resource. In hindsight, such a process was equivalent to the "secretary" outlined by Dijkstra (1975). In RC 4000 terminology it was known as a *conversational process*.

Initially the RC 4000 computer had only an extremely *basic operating system* running on top of the kernel. According to Lauesen (1975):

[5] Here I obviously meant "processes" rather than "programs."

> The RC 4000 software was extremely reliable. In a university environment, the system typically ran under the simple operating system for three months without crashes ... The crashes present were possibly due to transient hardware errors.

When the RC 4000 system was finished I described it in a 5-page journal paper (Brinch Hansen 1970). I then used this paper as an outline of the 160-page system manual (Brinch Hansen 1969) by expanding each section of the paper. Article 12 is a reprint of the most important part of the original manual, which has been out of print for decades.[6]

13 Venus System

The Venus system was a small timesharing system serving five or six users at a time:

> *The design of the Venus operating system.*
> Barbara H. Liskov (1972)

Although it broke no new ground, the Venus system was another convincing demonstration of Dijkstra's concepts of *semaphores* and *layers of abstraction*.

14 Boss 2 System

The *Boss 2 system* was an intellectual and engineering achievement of the highest order:

> *A large semaphore based operating system.*
> Søren Lauesen (1975)

It was an ambitious operating system that ran on top of an extended version of the RC 4000 kernel. According to its chief designer, Søren Lauesen:

> Boss 2 is a general purpose operating system offering the following types of service simultaneously: batch jobs, remote job entry, time sharing (conversational jobs), jobs generated internally by other jobs, process control jobs.

[6] My operating systems book (Brinch Hansen 1973) included a slightly different version of the original manual supplemented with abstract Pascal algorithms.

The system used over a hundred parallel activities, one for every peripheral device and job process. These activities were implemented as *coroutines* within a single system process. The coroutines communicated by means of semaphores and message queues, which potentially were accessible to all coroutines. These message queues were called "queue semaphores" to distinguish them from the message queues in the RC 4000 kernel.

Dijkstra (1968a) and Habermann (1967) were able to prove by induction that the THE system was *deadlock-free*. Lauesen used a similar argument to prove that the coroutines of Boss 2 eventually would process any request for service.

Boss 2 was implemented and tested by four to six people over a period of two years:

> During the six busiest hours, the cpu-utilization is 40–50 pct used by jobs, 10–20 pct by the operating system and the monitor.[7] The average operating system overhead per job is 3 sec.
>
> During the first year of operation, the system typically ran for weeks without crashes. Today it seems to be error free.

15 Solo System

Concurrent Pascal was the first high-level language for concurrent programming (Brinch Hansen 1975). Since synchronization errors can be extremely difficult to locate by program testing, the language was designed to permit the detection of many of these obscure errors by means of compilation checks. The language used the *scope rules* of *processes* and *monitors* to enforce security from race conditions (Brinch Hansen 1973, Hoare 1974).

By January 1975 Concurrent Pascal was running on a PDP 11/45 minicomputer with a removable disk pack. The portable compiler (written in Pascal) generated *platform-independent concurrent code*, which was executed by a small kernel written in assembly language.

The first operating system written in Concurrent Pascal was the *portable Solo* system which was running in May 1975:

> *The Solo operating system: a Concurrent Pascal program.*
> Per Brinch Hansen (1976a)

[7] The RC 4000 kernel was also known as the "monitor."

It was a single-user operating system for the development of Pascal programs. Every user disk was organized as a single-level file system. The heart of Solo was a job process that compiled and ran programs stored on the disk. Two additional processes performed input/output spooling simultaneously.

The Solo system demonstrated that it is possible to write small operating systems in a secure programming language without machine-dependent features. The programming tricks of assembly language were impossible in Concurrent Pascal: there were no typeless memory words, registers, and addresses in the language. The programmer was not even aware of the existence of physical processors and interrupts. *The language was so secure that concurrent processes ran without any form of memory protection.*[8]

16 Solo Program Text

Solo was the first major example of a modular concurrent program implemented in terms of abstract data types (classes, monitors and processes) with *compile-time checking of access rights*. The most significant contribution of Solo was undoubtedly that the program text was short enough to be published in its entirety in a computer journal:

> *The Solo operating system: processes, monitors and classes.*
> Per Brinch Hansen (1976b)

Harlan Mills had this to say about the Solo program text (Maddux 1979):

> Here, an entire operating system is visible, with every line of program open to scrutiny. There is no hidden mystery, and after studying such extensive examples, the reader feels that he could tackle similar jobs and that he could change the system at will. Never before have we seen an operating system shown in such detail and in a manner so amenable to modification.

PART VI PERSONAL COMPUTING

In the 1970s microprocessors and semiconductor memories made it feasible to build powerful personal computers. Reduced hardware cost eventually allowed people to own such computers. Xerox PARC was the leader in the

[8] Twenty years later, the designers of the *Java* language resurrected the idea of platform-independent parallel programming (Gosling 1996). Unfortunately they replaced the secure monitor concept of Concurrent Pascal with inferior insecure ideas (Brinch Hansen 1999).

development of much of the technology which is now taken for granted: bit-mapped displays, the mouse, laser printers and the Ethernet (Hiltzik 1999).

In Brinch Hansen (1982) I made two predictions about the future of software for personal computing:

> For a brief period, personal computers have offered programmers a chance to build small software systems of outstanding quality using the best available programming languages and design methods...The simple operating procedures and small stores of personal computers make it both possible and essential to limit the complexity of software.
>
> The recent development of the complicated programming language Ada combined with new microprocessors with large stores will soon make the development of incomprehensible, unreliable software inevitable even for personal computers.

Both predictions turned out to be true (although Ada was not to blame).

17 OS 6

The *OS 6 system* was a simple single-user system developed at Oxford University for a Modular One computer with 32 K of core memory and a 1 M word disk:

> *OS 6—an experimental operating system for a small computer: input/output and filing system.*
> Joe E. Stoy and Christopher Strachey (1972)

The system ran one program at a time without multiprogramming. The paper describes the implementation of the file system and input/output streams in some detail.

The operating system and user programs were written in the typeless language *BCPL*, which permitted unrestricted manipulation of bits and addresses (Richards 1969). BCPL was the precursor of the C language (Kernighan 1978).

18 Alto System

The Alto was the first personal computer developed by Xerox PARC. Initially it had 64 K words of memory and a 2.5 M-byte removable disk pack.

It also had a bit-mapped display, a mouse and an Ethernet interface. Over a thousand Altos were eventually built (Smith 1982).

The *Alto operating system* was developed from 1973 to 1976 (Lampson 1988):

> *An open operating system for a single-user machine.*
> Butler W. Lampson and Robert F. Sproul (1979)

This paper describes the use of known techniques in a small, single-user operating system. The authors acknowledge that "The streams are copied wholesale from Stoy and Strachey's OS 6 system, as are many aspects of the file system." A notable feature of the Alto system was that applications could select the system components they needed and omit the rest. The most revolutionary aspect of the system, its graphic user interface, was application-dependent and was *not* part of the operating system (Lampson 2000).

In some ways, the Alto system was even simpler than *Solo*. It was a strictly sequential single-user system. Apart from keyboard input, there was no concurrent input/output. The system executed only one process at a time.

It did, however, have an extremely robust file system that could be reconstructed in about one minute from "whatever fragmented state it may have fallen into."

A "world-swap" mechanism enabled a running program to replace itself with any other program (or an already preempted program). Swapped programs communicated through files with standard names. This slow form of context switching was used to simulate coroutines, among other things for a printer server that alternated strictly between input and printing of files received from the local network.

The Alto system was written almost entirely in BCPL. The use of a low-level implementation language provided many opportunities for obscure errors (Swinehart 1985):

> In the Alto system, it was possible to free the memory occupied by unneeded higher-level layers for other uses; inadvertent upward calls had disastrous results.

19 Pilot System

Pilot was another single-user operating system from Xerox:

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 (1980)

It was written in the high-level language Mesa (Lampson 1980). From Concurrent Pascal and Solo, Mesa and Pilot borrowed the idea of writing a modular operating system in a concurrent programming language as a collection of processes and monitors with compile-time checking as the only form of memory protection.

Mesa relaxed the most severe restriction of Concurrent Pascal by supporting a variable number of user processes. However, the number of system processes remained fixed. Mesa inherited some of BCPL's problems with invalid pointers, but was otherwise fairly secure (Swinehart 1985).

Pilot adapted several features of the Alto system, including its graphic user interface and streams for reliable network communication. Pilot supported a "flat" file system. As in the Alto system, redundant data stored in each page permitted recovery of files and directories after system failure. A file could only be accessed by mapping its pages temporarily to a region of virtual memory. This unusual mechanism was removed in the subsequent Cedar system (Swinehart 1985).

Ten years after the completion of the Alto system, operating systems for personal computing were already quite large. Pilot was a Mesa program of 24,000 lines. It was succeeded by the much larger *Cedar* system (Swinehart 1985).

Solo, Pilot, Cedar and a handful of other systems demonstrated that operating systems can be written in secure high-level languages. However, most operating system designers have abandoned secure languages in favor of the low-level language C.

20 Star User Interface

Graphic user interfaces, pioneered by Doug Englebart (1968), Alan Kay (1977) and others, had been used in various experimental Alto systems (Lampson 1988). The *Xerox Star* was the first commercial computer with a *mouse* and *windows* interface. It was based on the Alto, but ran three times as fast and had 512 K bytes of memory:

The Star user interface: an overview.
David C. Smith, Charles Irby, Ralph Kimball and Eric Harslem (1982)

This wonderful paper was written when these ideas were still unfamiliar to most computer users. Before writing any software, the Star designers spent two years combining the different Alto interfaces into a single, uniform interface. Their work was guided by a brilliant vision of an *electronic office*:

> We decided to create electronic counterparts to the objects in an office: paper, folders, file cabinets, mail boxes, calculators, and so on—an electronic metaphor for the physical office. We hoped that this would make the electronic world seem more familiar and require less training.
>
> Star documents are represented, not as file names on a disk, but as pictures on the display screen. They may be selected by pointing to them with the mouse and clicking one of the mouse buttons ... When opened, documents are always rendered on the display exactly as they print on paper.

These concepts are now so familiar to every computer user in the world that it is difficult to appreciate just how revolutionary they were at the time. I view graphic interfaces as one of the most important innovations in operating system technology.

The *Macintosh* system was a direct descendant of the Star system (Poole 1984, Hiltzik 1999). In the 1990s the mouse and screen windows turned the *Internet* into a global communication medium.

PART VII DISTRIBUTED SYSTEMS

In the late 1970s Xerox PARC was already using Alto computers on Ethernets as *servers* providing printing and file services (Lampson 1988). In the 1980s universities also developed experimental systems for distributed personal computing. It is difficult to evaluate the significance of this recent work:

- By 1980 the major concepts of operating systems had already been discovered.
- Many distributed systems were built on top of the old time-sharing system *Unix*, which was designed for central rather than distributed computing (Pike 1995).
- In most distributed systems, process communication was based on a complicated, unreliable programming technique, known as *remote procedure calls*.

- Only a handful of distributed systems, including Locus (Popek 1981) and the Apollo Domain (Leach 1983), were developed into commercial products.
- There seems to be no consensus in the literature about the fundamental contributions and relative merits of these systems.

Under these circumstances, the best I could do was to select a handful of *readable* papers that I hope are *representative* of early and more recent distributed systems.

21 WFS File Server

The *WFS* system was the first *file server* that ran on an Alto computer:

> *WFS: a simple shared file system for a distributed environment.*
> Daniel Swinehart, Gene McDaniel and David R. Boggs (1979)

The WFS system behaved like a remote disk providing random access to individual pages. To perform a disk operation, a client sent a *request* packet to the WFS host, which completed the operation before returning a *response* packet to the sender. The Ethernet software did not guarantee reliable delivery of every packet. However, since the server attempted to reply after every operation, the absence of a reply implied that a request had failed. It usually sufficed to retransmit a request after a time-out period.

There was no directory structure within the system. Clients had to provide their own file naming and directories. Any host had full access to all WFS files. The lack of file security imposed further responsibility on the clients.

In spite of its limitations, WFS was an admirable example of utter simplicity. (One of the authors implemented it in BCPL in less than two months.)

The idea of controlling peripheral devices by means of request and response messages goes back to the RC 4000 system. In distributed systems, it has become the universal method of implementing remote procedure calls.

22 Unix United RPC

Remote procedure calls (RPC) were proposed as a *programming style* by James White (1976) and as a *programming language concept* by me (Brinch

Hansen 1978). Since then, system designers have turned it into an unreliable mechanism of surprising complexity.

In their present form, remote procedure calls are an attempt to use *unreliable message passing* to invoke procedures through local area networks. Many complications arise because system designers attempt to trade reliability for speed by accepting the premise that users are prepared to accept unreliable systems provided they are fast. This doubtful assumption has been used to justify distributed systems in which user programs must cope with lost and duplicate messages.

Much ingenuity has been spent attempting to limit the possible ways in which remote procedure calls can fail. To make extraneous complexity more palatable, the various *failure modes* are referred to as different forms of "semantics" (Tay 1990). Thus we have the intriguing concepts of "at-most-once" and "at-least-once" semantics. My personal favorite is the "maybe" semantics of a channel that gives no guarantee whatsoever that it will deliver any message. We are back where we started in the 1950s when unreliable computers supported "maybe" memory (Ryckman 1983).

Tay (1990) admits that "Currently, there are [*sic*] no agreed definition on the semantics of RPC." Leach (1983) goes one step further and advocates that "each remote operation implements a protocol tailored to its need." Since it can be both *system-dependent* and *application-dependent*, a remote procedure call is no longer an abstract concept.

From the extensive literature on remote procedure calls, I have chosen a well-written paper that clearly explains the various complications of the idea:

> *The design of a reliable remote procedure call mechanism.*
> Santosh Shrivastava and Fabio Panzieri (1982)

The authors describe the implementation used in the *Unix United* system. They correctly point out that:

> At a superficial level it would seem that to design a program that provides a remote procedure call abstraction would be a straightforward exercise. Surprisingly, this is not so. We have found the problem of the design of the RPC to be rather intricate.

Lost and duplicate messages may be unavoidable in the presence of hardware failures. But they should be handled *below* the user level. As an

example, the *Pilot* system included a "network stream" protocol by which clients could communicate reliably between any two network addresses (Redell 1980).

23 Unix United System

By adding another software layer on top of the Unix kernel, the University of Newcastle was able to make five PDP11 computers act like a single Unix system, called *Unix United*. This was achieved without modifying standard Unix or any user programs:

> *The Newcastle Connection or Unixes of the World unite.*
> David R. Brownbridge, Lindsay F. Marshall and Brian Randell (1982)

Unix United combined the local file systems into a global file system with a common root directory. This *distributed file system* made it possible for any user to access remote directories and files, regardless of which systems they were stored on. In this homogeneous system, every computer was both a user machine and a file server providing access to local files. The most heavily used services were file transfers, line printing, network mail, and file dumping on magnetic tape.

The idea of making separate systems act like a single system seems simple and "obvious"—as elegant design always does. The authors wisely remind us that

> The additional problems and opportunities that face the designer of homogeneous distributed systems should not be allowed to obscure the continued relevance of much established practice regarding the design of multiprogramming systems.

Unix United was a predecessor of the *Sun Network File System* (Sandberg 1985).

24 Amoeba System

Amoeba is an ambitious distributed system developed by computer scientists in the Netherlands:

> *Experiences with the Amoeba distributed operating system.*
> Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren,
> Gregory J. Sharp, Sape J. Mullender, Jack Jansen and
> Guido van Rossum (1990)

An Amoeba system consists of diskless *single-user workstations* and a pool of *single-board processors* connected by a *local area network*. *Servers* provide directory, file, and replication services. *Gateways* link Amoeba systems to wide area networks.

A *microkernel* handles low-level memory allocation and input/output, process and thread scheduling, as well as remote procedure calls. All other services are provided by system processes. Like the RC 4000 multiprogramming system, Amoeba creates a dynamic tree of processes. Each process is a cluster of non-preemptible threads that communicate by means of shared memory and semaphores.

The system is programmed as a collection of *objects*, each of which implements a set of operations. Access rights, called *capabilities*, provide a uniform mechanism for naming, accessing and protecting objects (Dennis 1966). Each object is managed by a *server process* that responds to *remote procedure calls* from user processes (using "at-most-once" semantics).

The file system is reported to be twice as fast as the Sun Network File System (Sandberg 1985). Since files can be created, but not changed, it is practical to store them contiguously on disks. The system uses fault-tolerant *broadcasting* to replicate directories and files on multiple disks.

Nothing is said about the size of the system. Amoeba has been used for parallel scientific computing. It was also used in a project involving connecting sites in several European countries.

This ends our journey through half a century of operating systems development. The first 50 years of operating systems led to the discovery of fundamental concepts of hierarchical software design, concurrent programming, graphic user interfaces, file systems, personal computing, and distributed systems.

The history of operating systems illustrates an eternal truth about human nature: *we just can't resist the temptation to do the impossible.* This is as true today as it was 30 years ago, when David Howarth (1972b) wrote:

> Our problem is that we never do the same thing again. We get a lot of experience on our first simple system, and then when it comes to doing the same thing again with a better designed hardware, with all the tools we know we need, we try and produce something which is ten times more complicated and fall into exactly the same trap. We do

not stabilise on something nice and simple and say "let's do it again, but do it very well this time."

Acknowledgements

I thank Jonathan Greenfield, Butler Lampson, Mike McKeag, Peter O'Hearn and Bob Rosin for their helpful comments on this essay.

References

1. A. V. Aho 1984. Foreword. *Bell Laboratories Technical Journal 63*, 8, Part 2 (October), 1573–1576.
2. R. S. Barton 1961. A new approach to the functional design of a digital computer. *Joint Computer Conference 19*, 393–396.
3. H. Bratman and I. V. Boldt, Jr. 1959. The SHARE 709 system: supervisory control. *Journal of the ACM 6*, 2 (April), 152–155.
4. P. Brinch Hansen 1968. *The Structure of the RC 4000 Monitor*. Regnecentralen, Copenhagen, Denmark (February).
5. P. Brinch Hansen 1969. *RC 4000 Software: Multiprogramming System*. Regnecentralen, Copenhagen, Denmark, (April). *Article 12*.
6. P. Brinch Hansen 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–241, 250.
7. P. Brinch Hansen 1973. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ.
8. P. Brinch Hansen 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207.
9. P. Brinch Hansen, 1976a. The Solo operating system: a Concurrent Pascal program. *Software—Practice and Experience 6*, 2 (April–June), 141–149. *Article 15*.
10. P. Brinch Hansen, 1976b. The Solo operating system: processes, monitors and classes. *Software—Practice and Experience 6*, 2 (April–June), 165–200. *Article 16*.
11. P. Brinch Hansen 1978. Distributed Processes: a concurrent programming concept. *Communications of the ACM 21*, 11 (November), 934–941.
12. P. Brinch Hansen 1979. A keynote address on concurrent programming. *Computer 12* , 5 (May), 50–56.
13. P. Brinch Hansen 1982. *Programming a Personal Computer*. Prentice-Hall, Englewood Cliffs, NJ.
14. P. Brinch Hansen 1993. Monitors and Concurrent Pascal: a personal history. *SIGPLAN Notices 28*, 3 (March), 1–35.
15. P. Brinch Hansen 1999. Java's insecure parallelism. *SIGPLAN Notices 34*, 4 (April), 38–45.

16. C. Bron 1972. Allocation of virtual store in the THE multiprogramming system. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott Eds., Academic Press, New York, 168–184.

17. F. P. Brooks, Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, MA.

18. D. R. Brownbridge, L. F. Marshall and B. Randell 1982. The Newcastle Connection or Unixes of the World Unite! *Software—Practice and Experience 12*, 12 (December), 1147–1162. *Article 23.*

19. D. Burns, E. N. Hawkins, D. R. Judd and J. L. Venn 1966. The Egdon system for the KDF9. *The Computer Journal 8*, 4 (January), 297–302. *Article 6.*

20. F. J. Corbató, M. Merwin-Daggett and R. C. Daley 1962. An experimental timesharing system. *Spring Joint Computer Conference 21*, 335–344.

21. F. J. Corbató and V. A. Vyssotsky 1965. Introduction and overview of the Multics system. *Fall Joint Computer Conference 27*, 185–196.

22. P. A. Crisman Ed. 1965. *The Compatible Time-Sharing System: A Programmer's Guide.* Second Edition, The MIT Press, Cambridge, MA.

23. R. C. Daley and P. G. Neumann 1965. A general-purpose file system for secondary storage. *Fall Joint Computer Conference 27*, 213–229. *Article 8.*

24. J. B. Dennis and E. C. van Horn 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (March), 143–155.

25. E. W. Dijkstra 1968a. The structure of the THE multiprogramming system. *Communications of the ACM 11*, 5 (May), 341–346. *Article 11.*

26. E. W. Dijkstra 1968b. Cooperating sequential processes. In *Programming Languages*, F. Genuys Ed., Academic Press, New York, 43–112.

27. E. W. Dijkstra 1971. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 2, 115–138.

28. D. C. Englebart and W. K. English 1968. A research center for augmenting human intellect. *Fall Joint Computer Conference 33*, 395–410.

29. A. G. Fraser 1972. File integrity in a disc-based multi-access system. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott Eds., Academic Press, New York, 227–248. *Article 9.*

30. J. Gosling, B. Joy and G. Steele 1996. *The Java Language Specification.* Addison-Wesley, Reading, MA.

31. A. N. Habermann 1967. On the harmonious cooperation of abstract machines. Ph.D. thesis. Technological University, Eindhoven, The Netherlands.

32. G. H. Hardy 1969. *A Mathematician's Apology.* Foreword by C. P. Snow. Cambridge University Press, New York.

33. M. Hiltzik 1999. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age.* Harper Business, New York.

34. C. A. R. Hoare 1974. Monitors: an operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.

35. D. J. Howarth 1972a. A re-appraisal of certain design features of the Atlas I supervisory system. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott Eds., Academic Press, New York, 371–377.

36. D. J. Howarth 1972b. Quoted in *Studies in Operating Systems*, R. M. McKeag and R. Wilson Eds., Academic Press, New York, 390.

37. A. C. Kay and A. Goldberg 1977. Personal dynamic media. *IEEE Computer 10*, 3 (March), 31–41.

38. B. W. Kernighan and D. M. Richie 1978. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.

39. T. Kilburn, R. B. Payne and D. J. Howarth 1961. The Atlas supervisor. *National Computer Conference 20*, 279–294. *Article 3*.

40. B. W. Lampson and R. F. Sproull 1979. An open operating system for a single-user machine. *Operating Systems Review 13*, 5 (November), 98–105. *Article 18*.

41. B. W. Lampson and D. D. Redell 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117.

42. B. W. Lampson 1988. Personal distributed computing: The Alto and Ethernet software. In *A History of Personal Workstations*, A. Goldberg Ed., Addison-Wesley, Reading, MA, 291–344.

43. B. W. Lampson 2000. Personal communication, March 20.

44. S. Lauesen 1975. A large semaphore based operating system. *Communications of the ACM 18*, 7 (July), 377–389. *Article 14*.

45. S. Lavington 1980. *Early British Computers*. Digital Press, Bedford, MA.

46. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf 1983. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications 1*, 5, 842–856.

47. J. A. N. Lee 1992. Claims to the term "time-sharing." *IEEE Annals of the History of Computing 14*, 1, 16–17.

48. B. H. Liskov 1972. The design of the Venus operating system. *Communications of the ACM 15*, 3 (March), 144–149.

49. W. C. Lynch 1966. Description of a high capacity fast turnaround university computing center. *Communications of the ACM 9*, 2 (February), 117–123. *Article 5*.

50. W. C. Lynch 1972. An operating system designed for the computer utility environment. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott Eds., Academic Press, New York, 341–350.

51. R. A. Maddux and H. D. Mills 1979. Review of "The Architecture of Concurrent Programs." *IEEE Computer 12*, (May), 102–103.

52. J. McCarthy 1959. A time-sharing operator program for our projected IBM 709. Unpublished memorandum to Professor P. M. Morse, MIT, January 1. Reprinted in *IEEE Annals of the History of Computing 14*, 1, 1992, 20–23.

53. J. McCarthy 1962. Time-sharing computer systems. In *Computers and the World of the Future*, M. Greenberger Ed., The MIT Press, Cambridge, MA, 221–248.

54. J. McCarthy, S. Boilen, E. Fredkin and J. C. R. Licklider 1963. A time-sharing debugging system for a small computer. *Spring Joint Computer Conference 23*, 51–57.

55. R. M. McKeag 1976a. Burroughs B5500 Master Control Program. In *Studies in Operating Systems*, R. M. McKeag and R. Wilson Eds., Academic Press, New York, 1–66.

56. R. M. McKeag 1976b. THE multiprogramming system. In *Studies in Operating Systems*, R. M. McKeag and R. Wilson Eds., Academic Press, New York, 145–184.

57. F. B. MacKenzie 1965. Automated secondary storage management. *Datamation 11*, 11 (November), 24–28.

58. P. B. Medawar 1979. *Advice to a Young Scientist*. Harper & Row, New York.

59. C. Oliphint 1964. Operating system for the B 5000. *Datamation 10*, 5 (May), 42–54. *Article 4*.

60. E. I. Organick 1973. *Computer System Organization: The B5700/B6700 Series*. Academic Press, New York.

61. R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom 1995. *Plan 9 from Bell Labs*. Lucent Technologies.

62. L. Poole 1984. A tour of the Mac desktop. *Macworld 1*, (May–June), 19–26.

63. G. Popek, B. Walter, J. Chow, D. Edwards, C. Kline, G. Rudison and G. Thiel 1981. Locus: a network transparent, high reliability distributed system. *ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 169–177.

64. D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell 1980. Pilot: an operating system for a personal computer. *Communications of the ACM 23*, 2 (February), 81–92. *Article 19*.

65. M. Richards 1969. BCPL: a tool for compiler writing and system programming. *Spring Joint Computer Conference 34*, 557–566..

66. D. M. Ritchie and K. Thompson 1974. The Unix time-sharing system. *Communications of the ACM 17*, 7 (July), 365–375. *Article 10*.

67. D. M. Ritchie 1984. The evolution of the Unix time-sharing system. *Bell Laboratories Technical Journal 63*, 8, Part 2 (October), 1577–1593.

68. D. J. Roche 1972. Burroughs B5500 MCP and time-sharing MCP. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott Eds., Academic Press, New York, 307–320.

69. S. Rosen Ed. 1967. *Programming Systems and Languages*. McGraw-Hill, New York.

70. R. F. Rosin Ed. 1987. Prologue: the Burroughs B 5000. *Annals of the History of Computing 9*, 1, 6–7.

71. R. F. Rosin and J. A. N. Lee Eds. 1992. The CTSS interviews. *Annals of the History of Computing 14*, 1, 33–51.

72. R. F. Rosin 2000. Personal communication, March 20.

73. G. F. Ryckman 1983. The IBM 701 computer at the General Motors Research Laboratories. *IEEE Annals of the History of Computing 5*, 2 (April), 210–212. *Article 1.*

74. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon 1985. Design and implementation of the Sun Network Filesystem. *Usenix Conference*, (June), 119–130.

75. A. C. Shaw 1974. *The Logical Design of Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ.

76. S. K. Shrivastava and F. Panzieri 1982. The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers 31*, 7 (July), 692–697. *Article 22.*

77. R. Slater 1987. *Portraits in Silicon.* The MIT Press, Cambridge, MA, 273–283.

78. D. C. Smith, C. Irby, R. Kimball and Eric Harslem 1982. The Star user interface: an overview. *National Computer Conference*, 515–528. *Article 20.*

79. R. B. Smith 1961. The BKS system for the Philco-2000. *Communications of the ACM 4*, 2 (February), 104 and 109. *Article 2.*

80. M. Stonebraker 1981. Operating system support for database management. *Communications of the ACM 24*, 7 (July), 412–418.

81. J. E. Stoy and C. Strachey 1972. OS6—an experimental operating system for a small computer. *The Computer Journal 15*, 2 & 3, 117–124 & 195–203. *Article 17.*

82. C. Strachey 1959. Time sharing in large fast computers. *Information Processing*, (June), UNESCO, 336–341.

83. C. Strachey 1974. Letter to Donald Knuth, May 1. Quoted in Lee (1992).

84. D. Swinehart, G. McDaniel and D. R. Boggs 1979. WFS: a simple shared file system for a distributed environment. *ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, (December), 9–17. *Article 21.*

85. D. C. Swinehart, P. T. Zellweger and R. B. Hagmann 1985. The structure of Cedar. *SIGPLAN Notices 20*, 7 (July), 230–244.

86. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen and G. van Rossum 1990. Experiences with the Amoeba distributed operating system, *Communications of the ACM 33*, 12 (December), 46–63.

87. B. H. Tay and A. L. Ananda 1990. A survey of remote procedure calls. *Operating Systems Review 24*, 3 (July), 68–79.

88. J. E. White 1976. A high-level framework for network-based resource sharing. *National Computer Conference*, (June), 561–570.

89. M. V. Wilkes 1985. *Memoirs of a Computer Pioneer.* The MIT Press, Cambridge, MA.

90. R. Wilson 1976. The Titan supervisor. In *Studies in Operating Systems*, R. M. McKeag and R. Wilson Eds., Academic Press, New York, 185–263.

91. W. A. Wulf, E. S. Cohen, W. M. Corwin, A. K. Jones, R. Levin, C. Pierson and F. J. Pollack 1974. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM 17*, 6 (June), 337–345.