Programming Systems and Languages 1965–1975

Saul Rosen
Purdue University

In spite of impressive gains by PL/I, Fortran and Cobol remain the languages in which most of the world's production programs are written and will remain so into the foreseeable future. There is a great deal of theoretical interest in Algol 68 and in extensible languages, but so far at least they have had little practical impact. Problem-oriented languages may very well become the most important language development area in the next five to ten years. In the operating system area all major computer manufacturers set out to produce very ambitious multiprogramming systems, and they all ran into similar problems. A number of university projects, though not directly comparable to those of the manufacturers, have contributed greatly to a better understanding of operating system principles. Important trends include the increased interest in the development of system measurement and evaluation techniques, and increased use of microprogramming for some programming system functions.

Key Words and Phrases: languages, operating systems, programming systems, multiprogramming, history

CR Categories: 1.2, 4.22, 4.32

Author's address: Purdue University, Computing Center, Lafayette, IN 47907.

Introduction

In a paper "Programming Systems and Languages, a Historical Survey" [28], which was written in 1963, I tried to present a brief history of the development of computer software up to that time. I wrote a short addendum to that paper [29] in 1966. Now, about six years later, on the occasion of the 25th Anniversary of ACM, it seems appropriate to approach the subject again; to attempt to bring the history up to date, and in line with the general aim of this anniversary issue, to attempt to peer a short way into the future.

The earlier papers could attempt to cover all significant language and system developments since the field was young, and it was possible to be personally involved, or at least directly aware of all of them. In more recent years it has become impossible for any one person to keep up with all developments, and even if it were possible it would take a set of books rather than one short article to discuss them all. This paper must therefore be, even more than the earlier ones were, an essay from the point of view of one observer. The language and system developments that are covered are only the ones that the author was aware of and the ones that he considered most important and most influential.

Programming Languages

Perhaps the most striking fact about programming languages in the past ten years has been the continued overwhelming acceptance of FORTRAN and COBOL. In 1972 and on into the foreseeable future FORTRAN and COBOL are the languages in which most of the world's serious production programs are written.

When a language has achieved a certain level of acceptance it becomes increasingly attractive to go on

using that language. Relatively efficient compilers exist on a number of different computers. Libraries of routines have been developed that can be used or adapted as needed. There are large numbers of programmers who are familiar with the language. For most conventional scientific and data processing applications there is relatively little to be gained by deviating from the normality represented by languages like FORTRAN and COBOL.

There are of course many programming situations in which valid reasons exist for deviating from the standard languages. Unusual space or timing requirements might dictate the use of assembly language or the use of one of the many special system programming languages that have been developed [1]. There are situations in which the use of another language has become established, for example JOVIAL in the case of sections of the military establishment. There are also many special purpose languages which provide major advantages and conveniences in the areas in which they are appropriate.

There were some people in IBM and in the SHARE organization who thought that PL/I would gradually supplant and make obsolete languages like FORTRAN, COBOL, and ALGOL. PL/I contains most of the features of all of those languages, and provides many useful features that are not present in any of them. There were and still are those who blame the preeminence of FORTRAN over ALGOL on IBM's support of FORTRAN and its lack of enthusiasm for ALGOL. With IBM's very enthusiastic backing and its multiple, large implementation teams it looked as if PL/I was bound to succeed. And it did succeed to some extent. PL/I is now quite widely used, and its use is increasing. Instead of replacing FORTRAN and COBOL, it seems on the way toward joining them as one of the standard production languages. It works especially well for those installations which have large programs that involve both computing and data processing. Even if the computing and data processing load occurs in separate jobs, there are some real advantages to being able to use a single language for all types of computing.

Even for IBM, the introduction of a new major language has been a long, hard grind. The first series of PL/I compilers for the 360 were slow and clumsy. The basic language specifications were frozen before any experience had been gained in writing in the language or in writing compilers for the language, and features that appeared useful and desirable turned out to be confusing and difficult to implement.

There has been some debate as to whether one can reasonably talk about the efficiency of a programming language. In most situations it is the compiler and not the language that should be called efficient or inefficient. Yet it does seem clear that the efficiency of a compiler may depend on the definition of the language to be compiled. FORTRAN is essentially more efficient in some ways than PL/I or ALGOL. There are fewer things that have to be checked at compile time, fewer complex operations that have to be postponed to run time. Even with the recent efficient PL/I compilers on the 360, the class of problems that can be handled well by FORTRAN can still be handled more efficiently in FORTRAN than in PL/I. While similar efficiency considerations may not be valid when comparing PL/I to COBOL, it has been argued [33] that COBOL is a more natural and suitable language for typical data processing applications.

So far PL/I has been almost exclusively an IBM language. PL/I compilers do exist on machines by other manufacturers, but most of them are early, incomplete, and inefficient versions. An exception is Honeywell, which inherited a rather good PL/I compiler for its 600/6000 series computers when it took over the General Electric computer division.

The original descriptions of PL/I were quite informal, and the PL/I effort was subject to some criticism on this score. To produce a formal description of a language of the magnitude and complexity of PL/I was a formidable task. This task was undertaken by a group at the IBM research laboratory in Vienna, and the resulting document was so thick that it was humorously referred to as the Vienna telephone directory. Their work contains a number of interesting contributions in the area of language definition and description that are described in considerable detail in [23].

Algol 60 and Algol 68

ALGOL 60 has retained its importance as a publication language for numerical analysis, and in several of its extended versions it has been used also as a publication language for some of the more theoretical developments in computer software. The most interesting of the recent extensions is the language PASCAL [35]. In Europe, as the large IBM and Control Data computers become popular, the use of ALGOL as a production language faded, and FORTRAN became an international standard. The larger Burroughs computers, the 5000, 6000, and 7000 series, whose hardware design was very strongly influenced by ALGOL are still widely programmed in the Burroughs version of ALGOL. However, they are not widely used for scientific computation, and in recent years even Burroughs has yielded, at least to the extent of providing FORTRAN processors on their large computers.

At an early stage in the ALGOL development the decision had been made to break the effort into two parts. The first had as its aim the design of a language that would embody those concepts which were clearly understood and in which it was possible to achieve almost universal agreement. The result of this effort was ALGOL 60, presented to the world in the elegant document prepared by Peter Naur. The second effort, which was not implemented until the completion of the first, involved a study of advanced concepts in programming with the aim of providing a language, or perhaps a

series of languages of greater power and generality and usefulness than the original ALGOL 60.

A formal ALGOL working committee was set up under the auspices of the International Federation for Information Processing (IFIP), with an informal periodical, the Algol Bulletin. There was also considerable activity in this area outside of the formal committee structure.

The ALGOL working committee, after much discussion and dissension, produced a formal document describing a new language, ALGOL 68. One of the principles of design of the new language was to permit user extensibility. ALGOL 60 permitted a limited number of data types: real, integer, Boolean, and array. Concepts as simple as double precision or complex arithmetic were foreign to the letter of the ALGOL 60 definition even though they were added to some implementations. ALGOL 68 allows the programmer to define data types (modes) and data structures and also to define operators that apply to these defined modes.

The formal report on ALGOL 68 [34] is a document that is 140 pages long. These 140 pages are almost all formalism, with little exposition. The language description is inelegant, to the point of being unpleasant to read and difficult to understand. There have since been a number of attempts to present some of the principles of ALGOL 68 in readable form, of which one of the best and most accessible to date is [6].

A number of implementations of ALGOL 68 have been started and some may already be in use. It seems safe to predict that ALGOL 68 will not be widely used as a general purpose programming language in the same sense as FORTRAN or even ALGOL 60 have been used. The importance of ALGOL 68 will lie in its contributions to the theoretical development of programming language concepts, and it will be some time before these can be adequately evaluated.

Extensible Languages

Extensible languages are in a way the descendants of the compiler-compiler and compiler generator efforts of earlier years. An excellent summary of those areas is contained in [17]. Many language designers find it attractive to think in terms of a core language in which the only things that are defined a priori are a minimal set of primitives and a very general mechanism by means of which the language can be extended.

ALGOL 68 has already been mentioned as an extensible language, but in this area it is only one among many, and not necessarily the most important. At an extensible languages symposium [2] held in 1969, six extensible languages were discussed in addition to ALGOL 68. They included major language definition projects such as GPL [18], IMP [19], BASEL, and PPL. A recent survey, [8], mentions several other efforts in this area that qualify as language research and development projects of major importance.

These efforts have been of great theoretical interest, but so far at least, their impact on the practical side of programming has been small. Most computer users seem to be able to do all of the language development that they want to do within the framework of conventional subroutine definition and macro definition facilities provided in the popular general purpose languages and assembly systems.

T. E. Cheatman states in [8] that "extensible languages are just now about to emerge in a way that might have a significant impact."

Basic and APL

Two languages, BASIC and APL, have become popular in connection with the expansion of the use of on-line interactive computing. These languages represent two contrasting philosophies of computer language design.

BASIC was developed at Dartmouth in an effort to provide a language that was as simple and natural and easy to learn as possible. It was like a beginners subset of FORTRAN. It caught on because of its simplicity and also because of the effective time-sharing system built around it at Dartmouth and effectively marketed by General Electric. Most competing time-sharing services developed their own BASIC processors, and BASIC soon became the language of discourse for small time-sharing users.

The APL language was developed over many years by Kenneth Iverson at the IBM research laboratories—a development which culminated in the publication of his book [20] in 1962. The language and the book achieved considerable notice, mostly in academic circles. It seemed useful as a publication language for describing hardware and software computing algorithms but, at least originally, was not considered important as a practical programming language. In 1966 Iverson and his colleagues at IBM designed and installed an elegant time-sharing system on an IBM 360/50 based on the use of APL as a programming language. APL differs from most so-called higher level languages in that it attempts to emulate and exploit the conciseness and elegance of mathematical notation in the expression of algorithms. It provides many operators on scalars and vectors and matrices, and some of the operators do complicated things. The resulting conciseness is attractive to users of slow terminals, since powerful algorithms can be expressed in a few lines. Some people find this unattractive, claiming that the language is cryptic and confusing, but some, especially sophisticated programmers, love it.

Since the APL language and the processor and time-sharing system that supported it did not arise through normal IBM software support channels, it was ignored by IBM, but spread rapidly through word of mouth publicity and informal distribution until customer demand forced IBM to recognize its existence and set up more formal channels of distribution.

Special Purpose Languages

The discussion so far has only been in terms of general purpose precedure oriented languages. Many other more specalized languages have either become or remained important. In the area of string handling, SNOBOL 4 is now widely used. There are a number of simulation languages, SIMSCRIPT, GPSS, and SIMULA that have been implemented on many machines. There are well-known specialized languages in computer assisted instruction (COURSEWRITER, PLANIT), in graphics, in engineering design (ECAP, ICES), and in many other areas. Jean Sammet in her classic volume on programming languages [32] devotes almost 100 pages to languages of this type, including a comprehensive bibliography of developments through 1967.

Languages that help in the statement of computing problems in specialized areas may represent the most important computer language developments of the next five to ten years. By their very nature these languages belong to the specialized areas in which they apply, and are not often of major interest to computer scientists. Translator writing systems and extensible languages have been proposed and sometimes used as tools for the implementation of such languages. Some of the most useful have been written in FORTRAN and represent relatively straightforward extensions of FORTRAN.

Operating Systems

The "third generation" of large scale computing systems is characterized by multiprogramming operating systems based on large capacity disk and drum storage devices. In the new generation of computing systems that were delivered in the mid and late sixties, it became possible to provide adequate implementations of system concepts that had been proposed and tried with only moderate success on earlier second generation computers. These concepts include the idea of an automatic scheduling system in which jobs can be loaded continuously from many sources into queues in disk storage, from which they can be scheduled for execution according to their priority and their resource requirements. They include the concept of an on-line file system that provides safe permanent storage for programs and data files, and that provides the utility routines and backup facilities that are necessary for the successful operation of such systems. They include concepts like conversational remote job entry, and the varieties of interactive computing that are usually included within the connotation of the phrase time-sharing.

All of the major computer manufacturers set out to build major multiprogramming systems for their new equipment, and they all ran into serious difficulties and delays. The problems were essentially the same for all of them. There was too much system code that had to be resident in central memory and too much computer time used in system overhead functions. Attempts to reduce system overhead usually had the effect of increasing the amount of resident system code, thereby reducing the amount available for user jobs. The obvious remedy was to require larger and larger central memories; but core memory was expensive, and the size of core required in order to run the operating system efficiently might price the computer out of its market.

Peripheral storage access was slow, and the performance of the system was closely tied to the efficiency with which peripheral storage could be used. It is almost impossible to optimize access to peripheral storage in general systems of the type projected. Special high performance drums or fixed head disks for system residence can help, as can multiple high speed channels into main memory, but these can be expensive options in many computing systems. Early versions of most systems, and also some later versions, were disk access limited and used central processor and other system resources at only a small fraction of their capacity.

Systems had to be debugged and improved, and the process of change introduced new bugs and new problems. The systems became immensely complicated as new features were added and as more sophisticated algorithms were introduced in attempts to improve performance. The early days of most production systems were a nightmare of system crashes and temporary fixes.

Nobody really knew how to go about producing and debugging these new operating systems. IBM's approach was to use hundreds and even thousands of programmers in a "human wave" attack on the problem. Other less affluent manufacturers brought themselves to the verge of bankruptcy trying to imitate them. Academic critics pointed to their own in house systems that had been done in a few man years of effort, and urged the computer manufacturers to adopt their (often simplistic) approaches.

The complex, new multiprogramming systems needed a higher level of reliability than the earlier uniprogramming systems. In the earlier systems a system error, either hardware or software, would affect only the job that was then being run, and recovery usually required the rerunning of just one job. In the new systems a system crash could affect dozens of jobs in various stages of completion. Elaborate recovery procedures are needed to permit system recovery from errors with minimum loss of files already created and work already completed. Recovery procedures of this type are among the most difficult system programs to install, to debug, and to maintain.

For many of the reasons discussed above it was possible for an elaborate multiprogramming system to provide poorer performance on a computing system than a much simpler uniprogramming system on the same equipment. Thus several years after the initial release of EXEC 8 for the UNIVAC 1108 most customers were still using the uniprogramming EXEC 2 system which had

been developed earlier for the 1107 and which had provided the initial software support for the 1108. I do not mean to suggest that UNIVAC's performance was worse than that of other manufacturers. Their customers were perhaps luckier than those of some other manufacturers as they had an earlier system to fall back on while the new system was brought up to an acceptable level of performance. Most of them did eventually switch to EXEC 8 to be able to take advantage of the many features that are provided by modern operating system software.

The performance of the operating systems on the major computers improved gradually as new versions were released that partially solved some of the problems of earlier systems. It became apparent, at least to me, that reasonable operating systems could evolve from the clumsy inefficient structures that had been started with little analysis or understanding years earlier. Core memory has become cheaper and users have become resigned to the fact that to run a big system requires large central memory. Peripheral storage has improved and continues to improve in speed and accessibility. The mating of communication devices to computers has developed so far that to have large or medium scale computers at the center of impressive communication networks is relatively routine.

There are many operating systems on many different kinds of computers. I shall comment briefly on a few of those that, in my opinion, have been most important in introducing new concepts and in total impact on the computer field.

The Atlas System and Virtual Memory

The Atlas hardware and software system was one of the most interesting and forward looking of the second generation systems. The Atlas introduced the concept of the "single level storage system" [21] implemented by hardware paging. It was the first large scale implementation of the type of system that has come to be called a virtual memory system. Although it must have seemed ambitious to its designers, and so it was for its time, the Atlas was actually planned, designed, and built on a far too modest scale to be anything but a prototype for such systems in the future.

Paging, implemented by hardware address translation, is now a feature of many different computers, including models designed by just about every major computer manufacturer. The special problems of operating systems on paging machines have attracted much interest, which has produced a large literature in this area. The reader is referred to [11] for a detailed discussion of virtual memory and for an extensive bibliography.

The Burroughs Systems

The Burroughs 5000 system was the first, and is still perhaps the only, computer whose design was based on a number of well-defined software objectives. The aim of the design was to provide an appropriate compile time and run time environment for the new ALGOL 60 language, and features were included to assist in the handling of block structure and dynamic storage allocation, and to provide for the automatic interpretation of pure procedure segments encoded into Polish (parenthesis-free) strings. These features were extended and refined in the more recent Burroughs 6000 and 7000 series machines.

When the expression virtual memory became popular, it turned out that the Burroughs 5000 series already had a virtual memory system. As other "new" features appeared in operating systems on more widely used machines (e.g. OS 360), Burroughs programmers and Burroughs users proclaimed that these were things they had been doing for years. It may have been the first modern multiprocessor multiprogramming system [24], but its direct influence on most other operating systems was limited because the hardware architecture remained unique to the Burroughs line and because there were few customer installations during the early years in which the system was being developed.

From the beginning the Burroughs systems have been characterized by the use of ALGOL-like languages as system programming languages. There is no language processor that corresponds to the assembler on other systems, and higher level languages are the only languages used by system and problem programmers. The hardware organization of Burroughs computers makes this approach relatively efficient and quite effective.

The CDC 6000 Series

Control Data Corporation started building computers in 1958 and soon gained a reputation as a company that emphasized and delivered hardware performance and that kept costs and prices down by deemphasizing expensive but unessential activities like software development.

Its own success and the trend of developments in the computer industry led to a gradual change in attitude and an ever-increasing commitment in the area of computer software. This change in attitude was in progress during the time when CDC was developing its impressive 6000 series computer.

The first 6600, delivered in 1964, heralded the beginning of a new generation of computing systems. The 6000 series computers are powerful multiprocessor systems with one or two central processors sharing a large main memory with ten peripheral processors. Each peripheral processor also has its own private core mem-

ory, and among them they drive 12 high speed channels which can support a large variety of local and remote peripheral devices.

This equipment needed a sophisticated multiprogramming operating system, and a system was put together rapidly by the computer design group at the CDC Chippewa labs, hence the name Chippewa system. In striking contrast to the Burroughs approach, the Chippewa system was not only not written in a higher level language, but major parts of it were written in raw octal machine code.

The hardware designers had good insight into how an operating system should be structured for the equipment they had designed, and even though the original implementation lacked elegance, the basic concepts have survived. When the first 6600's were being delivered, the relatively new software division set out to produce a different and a more sophisticated operating system SIPROS; but the performance of SIPROS was never adequate, and after a few frustrating years they backed off and went to an enhanced Chippewa system which has evolved into the current SCOPE systems.

A group associated with the manufacturing plants in Minnesota built even more directly on the Chippewa structure and developed a system first called MACE and then renamed as KRONOS. Although KRONOS is being marketed mainly as a time-sharing system, it is probably the first major operating system that uses a preempt-resume scheduling strategy in an environment that is mostly dedicated to batch processing [4].

Control Data's early history of limited and inadequate software support has served to obscure the fact that the multiprogramming multiprocessor systems on the 6000 series computers are probably the most successful systems of their kind, and perhaps more than other better advertised systems, they provide considerable insight into the structure and problems of such systems.

IBM's System 360 and Its Operating Systems

The most important software systems of the period 1965–1970 were the systems developed by IBM for its system 360. This is true not because of the intrinsic merit of the systems themselves, but rather because of IBM's position of dominance in the computer industry. It is true in spite of the inelegant and often clumsy design of OS 360, and in spite of the inefficiencies and unreliability of the early versions that have been only partially alleviated by the more recent ones.

The documentation for the 360 operating systems is voluminous, almost to the point of being overwhelming. There are literally thousands of documents, and the techniques of computer assisted text preparation and publication make it possible to produce new documents and new versions of old documents at an alarming rate. A good brief introduction to OS 360 can be found in [25].

It is hard to point to any really new concepts or ideas that were introduced in OS 360. Its contribution is rather one of integration and synthesis. This was, perhaps, the first example of a really large modular system based on a few general concepts: reentrant code, queuing of control blocks, uniform and consistent handling of interrupts, etc. The designers of OS 360 introduced a great deal of new and useful terminology, much of which has become and will remain a standard for the whole industry.

Some of the most interesting and attractive concepts in OS lost some of their attractiveness in the actual implementations. One example is JCL, the Job Control Language. JCL provides a flexible and versatile control card language. The designers were right in wanting to make the many features of JCL available to the user, but the way in which they made them available in a language that is cryptic, confusing, and difficult to write and debug has made JCL one of the swear words of third generation systems. Another example in the area of data management is the indexed sequential access method. This represented a new level of service to the user. It made automatic the allocation and reallocation of storage, the setting up and modification of directories, the handling of overflow conditions and other features that would otherwise require difficult specific detailed coding. Unfortunately the user who took advantage of the availability of features of this type often found that the resulting inefficiency of computer use might offset the advantages offered by the powerful general capabilities of the system. This is of course true in most large systems. It stands out in OS because of the large number of general capabilities that have been designed into the system.

The original concept of OS 360 was to provide support for all 360 systems except for the very smallest ones which would use the very primitive BPS (Basic Programming Support) and BOS (Basic Operating System) startup systems.

It soon became apparent that OS was going to be the system for large 360's with at least 256K bytes of core, and that something else would be needed if the smaller systems were to run at all efficiently. This led to the parallel development at IBM of the Disc Operating System (DOS) which was started early in 1965, and which, according to its implementors, was released several months before the initial release of OS.

The early OS releases were quite primitive sequential systems that could not support concurrent input, output, and computing. A group of IBM support programmers at the Houston Space Flight Center developed a modified OS system called HASP (Houston Automatic Spooling Priority System) that became popular. Later multiprogramming versions of OS, MFT (Multiprogramming with a Fixed number of Tasks), and MVT (Multiprogramming with a Variable number of Tasks) do permit multiple reader and writer programs to operate simultaneously with one or more user programs. However, the facilities

for scheduling and resource allocation still seem inadequate at most large installations, and a more capable HASP2 system, which was introduced toward the end of 1968, is widely used. HASP has a rather ambivalent relationship to OS since the HASP program runs as one of the programs in the OS multiprogramming environment, but it is a specially privileged program that controls system input and output and that makes all of the important job-scheduling decisions.

In order to include reasonably sophisticated scheduling techniques in an operating system it is necessary to be able to preempt jobs, to remove them from primary storage (core) to secondary storage (drum or disk), and to later restore them to primary memory and resume their operation. All large third generation computers with the exception only of the IBM 360 systems provide at least one hardware relocation register which permits jobs loaded into main memory to be location independent. They can be preempted, moved to secondary storage, and later reloaded into a different area of primary memory and resumed.

Under the system 360 operating systems, if a job is moved out to secondary storage it can be resumed only if it is reloaded into the area from which it was dumped. This is practical only in special cases, and rollout or preempt-resume scheduling strategies are therefore not practical on the 360. This severely limits the storage management that can be used on the 360.

It was expected that the 370 series would correct this design flaw, but the 370 as originally released hardly differed from the 360. It seems to be an open secret that relocation hardware exists in the IBM 370 but is temporarily disabled, presumably pending the development of appropriate software.

The 360 Model 67 Operating Systems

IBM introduced the 360 model 67 to provide a product competitive with the GE 645 in the time-sharing market [30]. The model 67 is an atypical 360, which can run the standard 360 operating systems but which also has paging and segmentation hardware and so can support a sophisticated virtual memory operating system. The ambitious time-sharing system TSS 67 was announced in the spring of 1965 with the initial release scheduled for the fall of 1967. In the spring of 1966, I wrote in [29] that "all previous experience inclines this observer to be pessimistic regarding on-time delivery and initial performance." The early performance of the system was actually worse than even a pessimistic observer might have predicted.

The early performance was predicted with some accuracy in a simulation study by Nielsen [26], who was then a graduate student at Stanford University.

The TSS 67 effort was probably second only to OS 360 in terms of manpower and money expended on an operating system. It has been estimated that over 2,000 man years of work went into TSS. Eight successive versions were produced, almost all of which included the redesign of major components of the system. The effort led to interesting developments in the area of command language, interactive debugging, a table driven scheduler, and storage management strategies. Performance was much better in later versions than in the earlier ones. Some of the performance gains were achieved by adding more hardware, especially core memory. It turned out that paging was less effective in reducing the core memory requirements than had been expected. With adequate core memory and relatively few users, TSS was a usable, though expensive time-sharing system.

In the fall of 1970, Doherty [16] described efforts made at IBM's research center to improve the performance of TSS on their own 360/67. In that paper he points out that prior to these changes, i.e. using the standard TSS software as of the beginning of 1970, "we could support fewer than 15 simultaneous users on Release 4 of TSS/360 without LCS, and responses to trivial requests were in the 10–30 second range."

Although TSS was the standard supported 360/67 software, there were other systems developed and used. One of these was the Michigan Terminal System, developed at the University of Michigan and used there and at a number of other universities.

Another was CP67 which grew out of a project at the IBM Cambridge Research Center and became quite popular among users of the 360/67. CP67 introduced the concept of a "virtual machine." A typical time-sharing system attempts to make the user believe that he has the whole computer with its operating system all to himself. CP67 lets the user specify a system 360 hardware configuration and a controlling software system. Thus one user may be running OS 360 on a (virtual) machine with large core and many disk units while another is running DOS on another (virtual) machine with small core and very limited peripherals. Most users run the Cambridge Monitor System on a (virtual) 360/67. The system runs relatively efficiently when CMS is used, but much less efficiently if it is simulating a number of different 360 systems.

Multics

In the mid 1960's time sharing was the central concept of most operating system research projects. This emphasis was reinforced by the funding agencies of the United States government, especially by ARPA, the Advanced Research Projects Agency of the Department of Defense. The most important project supported by ARPA in this area was Project MAC at MIT.

The problems in the design of small dedicated time sharing systems had been solved on the prototype second generation systems, CTSS at MIT, the original BASIC System at Dartmouth, and others. Project MAC set out to solve the next generation of design problems for large

time sharing systems in MULTICS (Multiplexed Information and Computing Service) on the GE 645 computer, many of whose hardware features were designed to satisfy MULTICS requirements.

The aim of the MULTICS project was not merely to provide the conventional services that can be provided adequately by most time sharing systems—its goals were far more ambitious. They were based on a comprehensive philosophy of program design and structure, and on a concept of how computers should be used for research and scientific computation. It is a philosophy which had its analogues and its advocates in the earlier computer generations. The work of Holt and Turanski and their colleagues on the GP and GPX systems for UNIVAC I and UNIVAC II is most directly relevant.

Their approach envisions essentially all of the important achievements and activities in computer program design and development embodied in one large on-line programming library. The library consists of modules of programs and data and commentary organized within a retrieval structure that makes all modules available to authorized users. The library represents a living, growing structure. Research and program development groups in all installations using the system contribute to its growth.

The earlier GP systems were attractive to a small class of programmers—mostly programmers who were interested in the design and development of large programs and programming systems. The programming systems were complicated and system overhead was high. The same seems to be true of the MULTICS system.

The philosophy of systems like GP and MULTICS is most attractive in terms of a large community of users. Ideally all major universities and research laboratories should be included. MULTICS envisioned direct interconnection among user installations and looked forward to the large scale computer utility which would become the "depository of the data base and information processing procedures of the community."

The computing community was not ready for this kind of thing in 1965, and it may never really be ready for it. Bell Telephone Laboratories went along, but the more practically oriented university computing centers did not. Bell finally dropped out in 1970.

The MULTICS system is now running at MIT and at one or two other installations. It may yet be more widely used if Honeywell actually produces a rumored successor to the 645. Even though it has had limited practical success so far, MULTICS has had a tremendous amount of influence in the area of operating system software. Most of the important concepts in the design of virtual memory systems were developed by people connected with the MULTICS project. They produced a series of publications in which many programming system problems are analyzed and discussed. Some of the most important are [10, 13, 5]. Many of the problems are universal problems, and the discussions will contribute to everyone's better understanding of the principles involved.

The ARPA Network

In recent years ARPA has shifted its major emphasis in the area of computing to the building of a large network tying together many different computers with a great variety of operating systems. Most of the software problems arise from the complete lack of uniformity or standardization in the systems and in their control languages.

So long as there are great differences in hardware architecture it is not going to be possible to achieve even the degree of uniformity that has been achieved in the language area. The ARPA network may very well be a prototype of the computing systems of the 1980's, and its influence may be an important factor in a move toward standardization of operating system language and structure.

Dijkstra and His Followers

A one-page note on "Concurrent Programming Control" that appeared in Communications in September of 1965 was the first significant contribution to what has become an interesting and important approach in the area of operating systems. The paper was by E. W. Dijkstra [14]. See also Knuth [22]. It was a very useful contribution in that it provides a solution to a basic interlock problem, but it is more important for the approach that it exemplifies. There is a simple statement of the problem, a proposed solution, and then a proof of the fact that the solution does indeed solve the problem. At the First Symposium on Operating System Principles held at Gatlinburg, Tennessee, in October 1967, Dijkstra [15] presented a paper in which he very briefly described the THE operating system that he and his colleagues at Eindhoven had designed.

The following is a brief quotation from his paper: "We have found that it is possible to define 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 . . . At the time this was written the testing had not yet been completed, but the resulting system is guaranteed to be flawless."

Dijkstra's work has had a great deal of impact, especially in academic and research circles. A number of operating systems for relatively small computing systems have been built according to the principles enunciated by Dijkstra and his followers, and a number of others have been started. See, for example, Brinch Hansen [7].

It is not clear to what extent the concepts of formal structure and formal proof will apply to larger more general purpose operating systems. The modular hierarchical structure upon which the method is based may lead to intolerable inefficiencies in large systems. The formal proofs may turn out to be extremely difficult

mathematical exercises. Even if their methods are not universally adopted, Dijkstra and his colleagues have had a great and salutary effect on system designs. They have thought deeply about problems of interlock and interference and deadlock, and have clarified and solved many problems in these areas. They have helped bring clear thinking into system design.

Hardware Developments

With its model 85, IBM introduced a storage management concept based on a small "cache" or "buffer" memorythat uses hardware to produce a single level storage system. This attractive concept has been extended to all of the larger members of the IBM 370 family and will probably be used in many future systems [9].

For now the buffer memory concept deals only with two levels of very high speed memory. Typically the "slow" memory is magnetic core memory with a 2 microsecond cycle, and the fast memory is an integrated circuit memory with access time less than 100 nanoseconds. It does not yet affect disk and drum storage where access times are measured in milliseconds.

The most usual and most obvious prediction of the next major "breakthrough" in the computer field is that it will be the development of inexpensive random access memory, with access times in microseconds to replace the direct access drums and disks at equal or possibly even lower cost per bit. It is of course impossible to predict if this will happen by 1975. I am surprised that it hasn't happened yet.

When it does happen, it will have a very major effect on system software. It seems probable that in those future systems hardware management of multilevel storage as typified by the large 370 systems will play a much more important role than the software management techniques that are typical of today's virtual memory systems.

Microprogramming

It has become increasingly popular to use microprograms in a read-only memory in place of hard wired sequencing circuits in computer design. Microprograms can be used to perform functions that are usually considered to be software functions. It has been suggested that much of what we now call the operating system will soon be done by microprogramming rather than by software methods.

Several years ago [31] I used the term hardware programming rather than microprogramming to emphasize the fact that it is indeed programming and is often programming of the most detailed kind. When the microprogramming is done in a read-write memory (e.g. IBM's writeable control storage), the similarity with software programming becomes even more apparent.

Software in microprogramming seems bound to become more important in the future. The advances in this area have been slower and less dramatic than some of the enthusiasts for microprogramming predicted, but they have been important nonetheless. Recently delivered peripheral controllers, for example, contain as much sequencing hardware (implemented by microprogramming) as would be used in a large scale general purpose computer of a slightly earlier generation.

Conclusion

One of the encouraging developments in the field is the great current interest in system measurement, evaluation, and modeling [3]. Designers of hardware and software are more conscious now than they have ever been before of the necessity for incorporating measurement and evaluation techniques into systems at an early stage in their design. Many new hardware and software designs are incorporating features that will permit gathering of statistics and monitoring of performance at a level that has usually been prohibitively difficult or expensive in earlier systems.

There is much to criticize in the area of operating systems and computer software in general. Yet anyone who compares the situation now with that of five or ten years ago must be impressed by the tremendous progress that has been made. We try to do too much with our systems; we try to make them run reliably on equipment that cannot support them adequately. We do not always succeed, but over the years there has been a great increase in all of the things that count: in productivity, in reliability, in flexibility. We now do as a matter of course things that would have been considered almost impossibly difficult not very many years ago. We have achieved a much better understanding, both theoretical and practical, of the problems of operating system software. We are in a much better position to face the software problems that will be posed by future systems than we were just a few years ago. If ultra complex machines like the CDC STAR and the ILLIAC IV are any indication of things to come, the problems may be very difficult indeed, and it will be many years before the problems of operating system design and implementation can be dismissed as routine.

References

Only a few direct references are listed here. Many of these references contain much more complete bibliographies in the specific areas that they cover. See especially [12 and 32].

1. ACM Proc. of a SIGPLAN symposium on languages for systems implementation. Purdue U., Oct. 1971. Sigplan Notices 6, 9 (Oct. 1971).
2. ACM Proc. of the extensible languages symposium. Boston, May 1969. Sigplan Notices 4, 8 (Aug. 1969).

3.
ACM Proc. of SIGOPS workshop on system performance evaluation. Harvard U., Apr. 1971.
4.
Abell, V.A., Rosen, S., and Wagner, R.E. Scheduling in a general purpose operating system. Proc. AFIPS 1970 FJCC, Vol. 37, AFIPS Press, Montvale, N.J., pp. 89–96.
5.
Bensoussan, A., Clingen, C.T., and Daley, R.C. The MULTICS virtual memory. Proc. second symposium on operating system principles, 1969, pp. 30–42.
6.
Branquart, P., Lewi, J., Sintzoff, M., and Wodon, P.L. The composition of semantics in Algol 68. *Comm. ACM* 14, 11 (Nov. 1971), 697–708.
7.
Brinch Hansen, P. (Ed.). RC4000 software multiprogramming system. A/S Regnecentralen, Copenhagen, 1971.
8.
Cheatham, T.E. The recent evolution of programming languages. Proc. IFIP Cong. 1971, North Holland Pub. Co., Amsterdam, pp. I118–I134.
9.
Conti, C.J. Concepts for buffer storage. *Comput. Group News 2*, 8 (Mar. 1969), 9–13.
10.
Daley, R.C., and Dennis, J.B. Virtual memory, processes, and sharing in MULTICS. *Comm. ACM 11*, 5 (May 1968), 306–312.
11.
Denning, P.J. Virtual memory. *Computing Surveys 2*, 3 (Sept. 1970), 153–189.
12.
Denning, P.J. Third generation computer systems. *Computing Surveys 3*, 4 (Dec. 1971), 175–216.
13.
Dennis, J.B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602. Reprinted in [27].
14.
Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM 8*, 9 (Sept. 1965), 569.
15.
Dijkstra, E.W. The structure of "THE" multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
16.
Doherty, W.J. Scheduling TSS for responsiveness. Proc. AFIPS 1970 FJCC, Vol 37, AFIPS Press, Montvale, N.J., pp. 97–117.
17.
Feldman, J., and Gries, D. Translator writing systems. *Comm. ACM 11*, 2 (Feb. 1968), 77–113.
18.
Garwick, J.V. GPL, a truly general purpose language. *Comm. ACM 11*, 9 (Sept. 1968), 634–638.
19.
Irons, E.T. Experience with an extensible language. *Comm. ACM 13*, 1 (Jan. 1970), 31–40.
20.
Iverson, K. *A Programming Language*. Wiley, New York, 1962.
21.
Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H. One level storage system. *IRE Trans. Electronic Comput. EC-11* (Apr. 1962), 223–235.
22.
Knuth, D.E. Additional comments on a problem in concurrent programming control. Letter in *Comm. ACM 9*, 5 (May 1966), 321–322.
23.
Lucas, P., and Walk, K. On the formal description of PL/1. *Ann. Rev. Automatic Programming 6*, 3 (1969), 105–182.
24.
McKeag, R.M. Burroughs B5500 master control program. Report in series: Investigation of Operating System Techniques. Dept. Comput. Science, Queens U. of Belfast, 1971.
25.
Mealy, G.H., Witt, B.I., and Clark, W.A. The functional structure of OS/360. *IBM Systems J. 5*, 1 (1966), 2–51.
26.
Nielsen, N.R. Simulation of time sharing systems. *Comm. ACM 10*, 7 (July 1967), 397–412.
27.
Rosen, S. (Ed.). *Programming Systems and Languages*. McGraw-Hill, New York, 1967.
28.
Rosen, S. Programming systems and languages. A historical survey. Proc. AFIPS 1964 SJCC, Vol. 25, Spartan Books, N.Y., pp. 1–15. Reprinted in [27].
29.
Rosen, S. *Programming Systems and Languages*. Some recent developments. In [27], pp. 23–26.
30.
Rosen, S. Electronic computers, a historical survey. *Computing Surveys 1*, 1 (Mar. 1969), 7–36.
31.
Rosen, S. Hardware design reflecting software requirements. Proc. AFIPS 1968 FJCC Vol. 33, Pt. 2, AFIPS Press, Montvale, N.J., pp. 1443–1449.
32.
Sammet, J.E. *Programming Languages: History and Fundamentals*. Prentice-Hall, Englewood Cliffs, N.J., 1969.
33.
Sammet, J.E. Problems in, and a pragmatic approach to, programming language measurement. Proc. 1971 AFIPS FJCC, Vol. 39, AFIPS Press, Montvale, N.J., pp. 243–251.
34.
van Wijngaarden, A. (Ed.). Report on the algorithmic language ALGOL 68. *Numerische Mathematik 14* (1969), 79–218.
35.
Wirth, N. The programming language Pascal. *Acta Informatica 1* (1971), 35–63.