Some threads in the development of early operating systems

by GEORGE H. MEALY
Scituate, Massachusetts

ABSTRACT

This paper discusses many of the important themes in design of the early operating systems and their libraries. Historically, the library preceded the development of the early batch system but many of its utility functions gradually migrated to the system proper. Important ideas concerning system structure which were further elaborated in the sixties are briefly discussed.
INTRODUCTION

The broadest definition of an operating system (OS) is an environment for developing and running programs. The motivation for invention of the earliest batch systems was economic: computers were a scarce resource, thus it was important to minimize idle time and make running time as productive as possible. The early designs concentrated on the process of sequencing jobs and keeping the machine running.

The Times

During the early fifties, the field was still small enough for most of us to become personally acquainted. Few meetings required parallel sessions, and attendees were more or less conversant with both hardware and software. As the decade wore on, specialization necessarily increased.

By 1960, programmers were rapidly becoming a scarce resource; a typical installation was spending roughly the same amount of money for software development as for hardware. Programming, and especially system programming, was then coming to be regarded as a professional activity. In the early fifties, by contrast, few of us viewed ourselves as being programmers first and foremost. Typically, we had taken up programming in order to get our primary job done; many of us never entirely returned to our previous discipline.

The rapid advances in hardware technology and architecture forced programmers to become more and more inventive. In 1950, we must remember, the first commercially developed computers had yet to be delivered. At Bell Telephone Laboratories Inc. (BTL), the main computing resources in 1950 were the Bell Model 5 relay calculator and the GPAC analog computer. Within the decade, BTL went from these to the IBM Card Programmed Calculator, the IBM 650, rental of time on the IBM 701, and finally the IBM 704. Each change of hardware caused an upheaval in our way of doing business. By the end of the decade, large open shops had developed. FORTRAN was in widespread use, but machine language programming had far from disappeared.

Installations were dissimilar, even within the airframe industry. North American Aviation used an operating system quite early. The RAND Corporation did not use an operating system until 1960. Only one RAND closed-shop programmer then used FORTRAN, and the open-shop programmers were forced to use JOHNNIAC rather than the IBM 704. (The theory, I was told, was that the open-shop programmers might waste precious machine time.)

Design of the early systems was profoundly influenced by the fact that main storage was miniscule by today's standards. The IBM 704, for instance, at first had only 8192 36 bit words of core storage and at most 16384 words of drum storage. Cards were the primary input medium. While tape reliability was a major concern throughout the fifties, after the introduction of magnetic core storage main storage reliability had come more or less under control.

Scope

This report cannot pretend to be a comprehensive treatment of operating systems or of their libraries. To a great extent it reflects the author's personal involvement with the field. (I do not know, for instance, whether systems were developed for UNIVAC I, the RAYDAC, or the early 1100 series machines. Again, Holt and Turanski are said to have made important early contributions to the organization of the library, but I did not know them during that period.) A dispassionate historian would attempt to write a very different paper. But, since few of the systems of the fifties were described in the literature, such an historian would have great difficulty in researching the subject. There was no single line of development, even among systems designed for the IBM binary computers, and the wheel was probably reinvented many times over. Description of even a single system is a complex task, especially since the ideas of many people are involved; I have chosen to examine a number of threads in the tapestry, for it is far from clear how to describe the tapestry as a whole.

The gradual development of the online system library is in many ways a more interesting story than that of the batch system. Many functions now classed as OS functions were first embodied as utility subroutines and programs. The early assemblers, interpreters, and compilers (classed here as utility programs) were developed entirely outside of any OS context and in the almost complete absence of any theoretical guidance. Today, the library is an integral part of the OS—to the extent, for instance, that many programmers identify the UNIX system with its library rather than with its nucleus and shells.

During the fifties, a number of systems were developed which had no recognizable relation (we thought) to batch systems or to each other; in hindsight, I have little hesitation in calling them operating systems. Examples are the SAGE system (the first multiple-access computer?) and the BTL electronic switching systems (ESS.) There was a strong family resemblance, on the other hand, between the batch systems of the fifties and their structure was quite simple.

Most of the organizational ideas which became important in the following decade were present in rudimentary form during the fifties. This report ends by briefly tracing their roots in the fifties.
THE SHARE ORGANIZATION

SHARE, the first computer user group, was formed about the time of announcement of the IBM 704 in 1955. “SHARE” was not an acronym, but was aptly rendered as the “Society to Help Avoid Redundancy of Effort.” Its founders were among the organizations then using the IBM 701 and in desperate need of additional computing power. Their most immediate problem in conversion from the 701 to the 704, which was not upward compatible, was the development of basic math routines, utility subroutines, and utility programs. Only a few then existed, such as the assembler developed by IBM (NY AP1) which was shortly abandoned by SHARE in favor of that developed by Roy Nutt of United Aircraft (UA SAP).

A crucial event was IBM’s acceptance of the idea that it should reproduce and distribute at no charge routines and programs submitted to the SHARE Library. This was not unalloyed benevolence, for existence of the library became an important sales tool for IBM.

Cooperative development of operating systems started immediately, in the form of the General Motors and North American system for the 704, really a side effect of the formation of SHARE although not an official SHARE effort. (As late as 1959, barely half of the SHARE installations believed that an operating system was a good thing.) During the first three SHARE meetings in 1956, there had been discussion of possible improvements to the design of the 704. This resulted in announcement of the IBM 709 and in the formation of the 709 System Committee at SHARE IV in January, 1957. The system in question was SOS.

An important function of SHARE (and later user groups) has been informal education. This has taken place during committee meetings, formal presentations, the informal evening sessions in the SHARE Suite (which was well stocked), and in the SHARE Secretary Distribution—mailings to the member installations.

THE ONLINE LIBRARY

In 1955, the library was a set of file cabinets containing card decks. The decks were, variously, math and utility subroutines and utility programs. Typically, the programmer wanting to make a computer run submitted a card tray or one or more drawers of cards, any data tapes required, and a detailed set of operator instructions. To assemble and test a program, for instance, the card deck started with the binary form of the first pass of the assembler followed by the source program—a process called “linkage loading.”

The concept of PUBLIC and EXTERNAL symbols was introduced in compiler modifications made by Monte Minami (NY AP1) which was shortly abandoned by SHARE in favor of that developed by Roy Nutt of United Aircraft (UA SAP).

Since the FORTRAN relocatable object format followed the SHARE standard, UA SAP could be used to assemble routines to be loaded with compiled routines. More important was the fact that this allowed compiled programs to communicate with OS common data and routines. So, although the early versions of FORTRAN did their best (we thought) to defeat any integration into an OS, it was possible to ameliorate the effects of FORTRAN’s inexperience.

LANGUAGE PROCESSORS

The earliest language processors were interpreters. The design of the IBM Card Programmed Calculator precluded any other approach: the interpreter consisted of the hardware and suitable plugboard wiring—in effect, it was microprogrammed. At that time, interpreters were often, as on the IBM 650 before development of the SOAP assembler, written in decimal machine code.

While it was generally recognized that interpreters did not make for efficient use of the machine (by a factor of roughly 100 or worse), compilation was in its infancy, and many of us believed that compilers could not, even in principle, match the efficiency of good assembled code. But, interpreters were ubiquitous during the early fifties, and many if them survived through the early sixties, having been moved from one machine to another in the meantime, principally to avoid reprogramming applications.

A quite different activity on the language front was the invention of the list processing languages IPL (Newell, Shaw, and Simon of Carnegie Tech and the RAND Corporation) and LISP (McCarthy’s group at MIT). The ideas of pointer data and the pushdown stack introduced by list processing swiftly became indispensable tools in compiler and OS design.
Of the advances in assembler design, the invention of macro substitution was the most important. I do not know where and when it originated, but the idea was a component of the development of SOS, and a macro preprocessor for UA SAP was developed by Irwin Greenwald of RAND about 1957. In 1958-59, macros were developed much further by Eastwood and McIlroy at BTL and implemented in BE SAP, descended from UA SAP. Many of their ideas survive in current assemblers. (During the sixties, a number of unreconstructed machine language programmers seriously proposed discarding compilers in favor of macro-assemblers!)

The one theoretical advance of the fifties which had any immediate effect was the invention of Backus-Naur form, based on the work of the logician E. L. Post during the mid-thirties. Finite-state automata, discovered in the early fifties and based on the work of A. M. Turing, became important in the design of compilers only in the sixties. Chomsky's discovery of phrase-structured grammars about 1955 was also based on Post's ideas, but again did not influence compiler design until much later. While some of us felt intuitively that the grammar must be the guide to compilation of a syntactically correct program, the first syntax-directed compiler was not developed until about 1960 by Ned Irons.

COMMAND LANGUAGES

The idea of an OS command language developed rather slowly. The earliest systems used control cards in a fixed field format, and the processing pattern within a batch job was pretty much restricted to variations of that described earlier. Later, the assembler source format was adopted, the operation code field being used to call the assembler or compiler from the library, load a user program, specify breakpoint dumps, and the like. Despite the Byzantine semantics of the later OS/360 job control language, its syntax remained that of the assembler.

The MIT Compatible Timesharing System (CTSS) first allowed the operation field to be the filename of an arbitrary program, a development which preceded the widespread use of mass storage and thus online storage of user programs and data files. At this stage of OS evolution, the command processor became (or should have become) just another program. The way was now clear for development of command processors which used different linguistic styles but could coexist on the same OS. The Bourne and Berkeley shells for UNIX are examples.

SYMBOLIC MODIFICATION

Deplorable as it was, the practice of patching object programs was widespread. That there was a net saving in machine time is doubtful. Many programs ended up being such a welter of patches that it was difficult, if not impossible, to update the sources correctly. The original FORTRAN compiler and its runtime input-output library were cases in point: In order to embed the runtime routines into BESYS, I was forced many times to iterate the process of using Arthur Samuel's dissembler on the relocatable decks, commenting the resulting source code, and reassembling. The effort was worthwhile quite apart from what I learned while trying to understand Roy Nutt's code, for it was then easy to include new features in FORMAT statements and even to allow FORMAT statements to be read at runtime.

At the first meeting of the SHARE 709 System Committee in 1957, Chuck Baker of RAND presented a proposal which he called the "Alpha System." His idea was that the source program should be encoded into a form which allowed rapid loading but also retained the information necessary to reconstruct the source program. The loading process would collate in modifications written in the source language. If desired, an updated compressed (SQUOZE) deck and listing could be obtained. The symbol table was then to be used to aid in formatting dump output.

I can attest to the fact that the difference this made in the life of a machine (or assembler) language programmer was extraordinary. Unfortunately, this did not help the FORTRAN programmer. A version of the assembly pass of 709 FORTRAN which would produce a SQUOZE deck was coded, but never debugged. The basic problem here was that the idea of linkage loading did not come early enough to have any impact on the design of SOS. In the event, while we eventually got FORTRAN working within SOS, it was used in conjunction with the assembler written by Dave Ferguson of UCLA.

DEBUGGING TOOLS

Machine designs often incorporate features intended to assist in diagnosing machine and program bugs. The SAGE system, for instance, had an extremely elaborate diagnostic console. The Lincoln Laboratories utility control program (UCP) was designed to simulate the actions of an operator using that console and was directed by control cards; UCP was the immediate ancestor of BESYS. But, debugging through use of the machine console was very expensive and thus of limited utility. Post-mortem dumps were often uninformative, and a program trace usually unsurveyable.

The idea of the breakpoint, rather than post mortem, dump was to get a picture of the console state and selected portions of storage at points specified by the user. While one could insert code at these points to output information, two severe difficulties existed in practice: The breakpoint code was frequently buggy, and the cost of reassembly was high. In SOS, breakpoints were specified by inserting debugging macros, thus effectively eliminating these difficulties.

The notion of a breakpoint dump was not new at the time, but combining it with the idea of symbolic modification and macros was. Marty Belsky and his group at the IBM service bureau in New York City invented a different approach, described next, that was implemented on the IBM 704 in a matter of a few months.

The design of the 704 had included a feature designed to facilitate program tracing. When the trapping mode was enabled, any successful jump instruction sent control to location 1, depositing the address of the trapped instruction at location 0. A special instruction, trap transfer, would never be trapped. As far as I know, the trapping mode was never used for its intended purpose, but it could be used for obscure forms of subroutine and coroutine linkage.

In the case of NY SNAP, a table containing one four word
entry per breakpoint was built and an instruction in each entry was swapped with the instruction in the program at the breakpoint. Dumps were written in binary to a scratch tape and converted following the end of the program test. This scheme is still in use, although dump output conversion is now done on the spot.

Roy Nutt used the trapping mode for communication between FORTRAN compiled code and his runtime routines which interpreted [sic] FORMAT statements.

THE SIXTIES

The early batch systems considered only one job at a time, and the order of execution of the jobs was fixed once a batch started. Job scheduling could not become a system function until the advent of plentiful random access storage, nor could a system designer contemplate switching control between members of a set of partially executed jobs. The latter, given a suitable adjustment of terminology (read "job" as "unit of processing," "task," or "process") was the typical processing pattern in the real-time systems and became that of the operating systems of the sixties and later.

Apart from the presence of mass storage, this development benefited from a number of earlier hardware and programming advances. These are summarized below.

Multiple Access

SAGE, under development in the fifties, had a number of consoles of different types. The most numerous displayed radar plots and other data related to missile tracks. The console operator gave direction to his part of the enterprise by keying in data. The SAGE program was responsible for monitoring all system inputs (radar, communications lines, and consoles) by polling each class of inputs on a regular basis and yielding control to an appropriate subprogram whenever it noted a change of status. The ESS program was not unlike the SAGE program in that its overall control was based on polling inputs. Batch systems, by contrast, had only one input source (apart from switches on the operator's console) to worry about.

Hardware Interrupts, Multiprogramming, and Multiprocessing

The practice of polling elevated what should ideally have been a low level concern—noting the presence of new input data and other status changes—to a high level concern in system design. The introduction of hardware interrupt systems by STRETCH corrected this situation and, in principle at least, reduced system overhead and response time.

Input-output channels allowed input-output operations to overlap central processing unit (CPU) operation. On the output side, this was relatively easy to accommodate in OS design, even when the interrupts were not used. On the input side, "anticipatory buffering," as the practice came to be called, was a more complex proposition. The design of the SOS buffering subsystem was predicated on the presence of these hardware facilities. Ironically, the official version of SOS was never run with the interrupts enabled.

The term multiprogramming derived from the view that a computer runs programs and an interrupt diverts its attention from one program to another. Multiprocessing referred to a computer configuration with multiple CPUs, whether or not they shared main storage. Both terms are more suggestive than exact and have meant different things to different people. Regarding the former term, the principal focus in designing a real-time system has been the data rather than the program; use of the term multiprogramming would be inappropriate, quite apart from the use by the early systems of polling rather than hardware interrupts. Regarding the latter term, CPUs have been joined within single hardware systems by other types of processors, such as array processors, numeric processors, and input-output channels; by only a slight abuse of terminology one could call STRETCH, the IBM 709, and their suite, multiprocessing systems. The GAMMA 60 was, to use Doug McIlroy's word, multi-everything.

By late 1963 it was clear that scheduling and dispatching of whatever units of processing was already an important system function. Whatever one called the unit of processing in any context, it must be accompanied by a sort of a job ticket which identified among other things the program, the data, and the type of processor to be used. Scheduling and dispatching then arranged these in queues and from time to time assigned each processor to a given unit of processing. The terms task (OS/360) and process (MULTICS) were neutral with respect to program and data. Hence my coined term multitasking. Multitasking was not a new invention but a revisionist view of the early system organizations.

Dynamic Relocation and Paging

Linkage loading of relocatable object programs had made it unnecessary for programs to directly reference absolute locations in storage or to be assembled at absolute origins. However, once the program was loaded, it was bound to absolute storage addresses and could not be moved. Even before the advent of timesharing, there were situations in which it was desirable to write a partially executed program out to a backing store and then read it later into a different location in main storage. Further, a buggy program had to be prevented from destroying its near or distant neighbors.

The solution to these problems embodied in the STRETCH hardware, and later built into the augmented 704 on which CTSS was first implemented, was to provide machine registers to dynamically relocate addresses in the program and to bound the range of allowable memory references, on the pain of a memory protection interrupt. A quite different and more elegant solution was provided by the British ATLAS computer's paging hardware. An additional benefit of the ATLAS design was provision of a much larger (virtual) memory than that implied by the width of the machine's memory address registers.

ACKNOWLEDGEMENTS

Bernie Galler, Doug McIlroy, Owen Mock, George Ryckman, and Kei Shimizu have jogged my memory and suggested changes of emphasis, for which I thank them.