On the use of generalized executive system software

by WILLIAM GORMAN

Computer Sciences Corporation
Silver Spring, Maryland

INTRODUCTION

The characteristic of third generation computing systems that most distinguishes them from previous ones is that they are designed to perform multiprogramming. The purpose of multiprogramming is cost-effective utilization of computer hardware, which is achieved by reducing the CPU time otherwise lost waiting for completion of I/O or operator action. An operating system is necessary to achieve multiprogramming: to schedule jobs, allocate resources, and perform services such as I/O.

Since these systems must be very generalized in order to accommodate the vast spectrum of potential application program requirements, they require some specific information from the user if they are to perform effectively. To supply the needed information and intelligently (efficiently) use the system, then, the user must have some understanding of the operating system's function as related to his particular needs.

A third generation computing system has so much generality that to use or understand it one must wade through stacks of manuals that seem neither clear nor convenient. Problems plague users, who get caught in a juggling act with executive system control language. The unwary become hopelessly involved, generating endless control card changes, with attendant debugging problems and loss of valuable personnel time. Other users have gotten almost mystic about their job control language (JCL), taking an "it works don't touch it" attitude. With direction such as this, even competent organizations can become very inefficient, using far more hardware, software, and human resources than are actually needed for the work at hand.

Before we surrender and send out an appeal for the "save money" salesmen, let's examine the purposes of executive system software and determine if the application of a little horse-sense doesn't go a long way toward solving our dilemma. Randall1 notes in his excellent paper on operating systems that the quite spectacular improvements that are almost always made by tuning services "are more an indication of the lamentable state of the original system, and the lack of understanding of the installation staff, than of any great conceptual sophistication in the tools and techniques that these companies use." Clearly, the key to effective use is understanding of the operating system and of its interface between a job and the hardware available to perform that job.

It is the purpose of this paper to suggest ways to effectively use a third generation operating system. Most of the examples used will be from the most generalized and complicated of them all—IBM OS/360. Our examination of operating systems will begin with the typical hardware resources of a computing plant and the OS response to those resources. A brief overview of the user tools supplied by the operating system will then be presented, followed by discussions on bugs and debugging and other problems of performance.

Our conclusion will cover the most valuable and cantankerous resource of all—human. Lack of space prevents a complete tutorial, but it is the author's hope that many questions and ideas will be raised in the reader's mind. Perhaps a thoughtful user may see ways to regain effective use of his computing facility, or as Herb Bright says, "Learn to beat OS to its knees."

HARDWARE RESOURCES

CPU time

The first and most valuable resource we shall examine is that of computer time itself. Emerson has said, "Economy is not in saving lumps of coal but in using the time whilst it burns." So it is with computer time. Most large computer shops run their computers 24 hours a day, yet typically their central processing units are doing useful work for far too small a percentage of that time.

Cantrell and Ellison1 note that "The second by second performance of a multiprogrammed system is always limited by the speed of the processor or an I/O channel or by a path through several of these devices used in series.... If some limiting resource is not saturated, there must be a performance limiting critical path through some series of resources whose total utilization adds up to 100%." To achieve the theoretical potential of a computing system, we must manipulate it so as to increase the percentage of resource use. Analysis of the bottlenecks that cause idle time generally reveals that resource needs of companion runs are in conflicting demands in such a manner as to gain greater use of the CPU.
There are three states of CPU time: wait, system, and active. System wait time is time when the CPU is idle. System time is that time spent in supervisory routines, 1/O and other interrupt processing, and error handling—most of which is considered overhead. Active time is the time spent executing program problems. Any reduction of system or wait time makes more time available for problems, thus contributing to greater efficiency.

I/O channel time

The channel handles the transfer of information between main storage and the I/O devices and provides for concurrency of I/O and CPU operation with only a minimum of interference to the CPU. Whenever I/O activity overloads a CPU, idle time can result because the CPU might be forced to wait for completion of I/O activity in order to have data to process. Such cases might be an indication of poor job mix.

Problems also result from the frequency and duration of I/O activity. When data is moved in many small bursts, competition for channels and devices can markedly slow the progress of the operating system.

Main storage

Main storage, or memory, is probably the most expensive and limiting resource in a computing system, besides the CPU itself. Many programs use huge amounts of memory—often more than is available. Since the consumption of memory by programmers seems, like Parkinson's Law, to rise with availability, it is doubtful that is initiated.

Certain portions of the operating system must reside permanently in main memory in order to execute; but the basic system is too large, with many portions too infrequently used, to make it all resident. Memory not used by the system then serves as a pool of storage from which the system assigns a partition or region to each job step as it is initiated.

One memory scheme used by the 360 breaks memory up into fixed-length parts or partitions, and the user program is allocated the smallest available partition that will accommodate it. Another 360 scheme has the system allocate (de-allocate) memory at execution time in the specific amounts requested. This method is more complicated, with more overhead, but it permits a greater variation in the number and size of jobs being executed.

The most common abuse of memory that I have observed is over-allocation, or more simply the request for greater amounts of memory than are used. Fragmentation, a particularly frustrating problem, results from the requirement that memory be allocated to user jobs in single continuous chunks. As jobs of varying size are given memory, the memory assignments are at first contiguous to one another. When a job finishes, the space it occupied is freed and can be assigned to another job or jobs. However, if subsequent jobs require less than the full amount vacated, small pieces or fragments of unused memory occur and must wait until jobs contiguous to them are ended and can be combined back into usable size. As a result, when we wish to execute programs with large storage needs, the operator often must intervene and delay the initiation of other jobs until enough jobs terminate to create the necessary space. Thus, our CPU can become partially idle by virtue of our need to assemble memory into a single contiguous piece large enough to start our job.

Direct-access storage

Direct-access storage is that medium (drum, disk, or data cell) where data can be stored and retrieved without human intervention. Modern computing demands could not be met without direct-access storage, and operating systems could never reach their full potential without it.

The operating system uses direct-access to store system load modules and routines for use upon demand. Control information about jobs waiting to be processed, jobs in process, and job output waiting to be printed or punched is stored on direct-access devices by the operating system. The system also provides facilities whereby user programs have access to temporary storage to hold intermediate data.

Magnetic tapes

Data can be recorded on magnetic tape in so many different forms that we frequently sacrifice efficiency through lack of understanding. We often encounter difficulty with I/O errors not because of bad tapes, but rather due to incorrect identification to the operating system of recording format and such trivial things as record size. Further errors can develop from contradictions between our program's description and the JCL description of the same data.

We generally inform the operating system of the recording format, etc., through JCL parameters. The system provides many services in the handling of tapes, one of the more important ones being the ability to identify data sets on tape by comparing JCL parameters with labels written as separate files in front of the data being identified. In my diagnostic work, I have identified more I/O errors as due to bad JCL and wrong tape mounts than as legitimate I/O errors. Due to the perishable nature of tape, provision for backup must also be made.

Unit-record devices

Printers, card readers, and punches all fit into this category. The operating system rarely reads or writes user data directly from user programs to these units. Normally, data input from a card reader or output to a punch or
printer is stored as an intermediate file on direct-access devices, so that the system can schedule the use of these relatively slow devices independently of the programs using them. High volume and slow speed can occasionally cause system degradation.

SOFTWARE TOOLS

Many of the tools of the operating system are independently developed segments or modules collected into libraries for use by the system and the user. Additional libraries are created to contain installation-developed routines, programs, and utilities.

Utilities

Supplied with our operating system are numerous service programs or utilities for performing frequently used operations such as sorting, copying, editing, or manipulating programs and data. Among the services supplied are programs to update and list source files, print or punch all or selected parts of data sets, and compare sets of data.

Generally these programs are easy to use once learned, are control card driven, and have "...the priceless ingredient of really good software, abrasion against challenging users." They are generally stable from operating system release to release.

User-written utilities

This brings us to the subject of user-written utilities and programs. A search that I personally made at one installation uncovered over 20 user-written utilities, all occupying valuable disk space and all performing the same function—updating program source files. The only reason for this that I could discover was that the user was unable or unwilling to understand the utility already available. Despite what I termed waste, the writers, to a man, thought their approach sensible.

Many useful utilities have been user-written, and often copies can be secured from the writers at no charge or low charge.

Programming languages and subroutine libraries

Programming languages have been developed to reduce the time, training, expense, and manpower required to design and code efficient problem programs. Essentially they translate human-readable code into machine-readable instructions, thus speeding up the programming process. With these language translators come subroutine libraries of pretested code to handle functions such as deriving the square root of a number of editing sterling currency values.

It is beyond the scope of this paper to discuss language selection, but one note seems in order. When only one or two programmers in an installation are using a complicated higher-level language, those users can encounter serious debugging problems for which no help is available from fellow programmers, due to a lack of expertise in that language.

Input/output control systems

The IOCS portion of the system automatically synchronizes I/O operations with the programs requesting them, provides built-in automatic error handling, and is further extended by system schemes to handle queues of I/O requests from many totally unrelated programs. The system also permits the user to change his output medium with only a simple change to JCL. Users need to write I/O code at the device level only when introducing unique, special-purpose hardware to a system.

Linkers and loaders

The 360 linkage editor combines program segments that were compiled or assembled separately into a single program ready to be loaded. We can therefore make changes without recompiling an entire program. The linkage editor also permits us to create a program too large for available hardware by breaking it into segments that can be executed and then overlaid by other segments yet to be executed.

The loader handles minor linkage tasks and physically loads into main storage the programs we wish to execute.

JCL AS A GLUE

Operating system job control languages (JCL) have been established to enable us to bypass the operator and define precisely to the system the work we wish to perform. JCL reminds me of glue: used properly, it's effective; used poorly, it's a mess.

I look upon the differences between the major operating systems as trade-offs between simplicity and flexibility. UNIVAC and most of the others have opted for simplicity, while IBM has stressed flexibility. For example, UNIVAC uses a very simple control language with single-letter keys that identify the limited range of options permitted via control card. IBM, on the other hand, allows extreme flexibility with literally dozens of changes permitted at the control card level—a not very simple situation.

I consider 360 JCL to be another language—quite flexible, but unfortunately a little too complicated for the average user. To execute a job on the 360 we need three basic JCL cards: a job card to identify our job and mark its beginning, an execute card to identify the specific program we wish to execute, and data definition (DD) cards to define our data sets and the I/O facilities needed to handle them.

When we supply information about a data set via JCL rather than program code, it becomes easier to change
parameters such as block size, type of I/O device used, etc., than it would be with other control languages, simply because no recompilation is required as is frequently so with the other approaches. However, due to the complexity of the process we can unknowingly make mistakes. For example, to create printed output under 360 OS we need only code SYSOUT=A on the DD card describing the data set. Since printed output is usually stored on intermediate disk files, a block size is needed; but unless block size is specified, the output may end up unblocked. Also we might not be able to estimate the volume of printed output that would be generated before our job fails for lack of space allocated to handle the printed output.

Numerous and extremely troublesome problems are generated when our use of JCL is uninformed or haphazard. The large number of JCL parameters required to properly execute a job introduces error possibilities due to sheer volume and an inability to remember every detail required by a large process. Even proficient JCL users may require several trial runs to iron out bugs, while uninformed users frequently give up and instead borrow JCL that allegedly works, even though that JCL may not really match their needs. It therefore becomes imperative that we devise ways to help users assume their proper responsibilities and get away from JCL as much as possible.

IBM assists by making provisions for cataloged libraries of JCL called procedures. To invoke a procedure, a user need supply only a job card and an execute card for each procedure we wish to execute. Within a procedure, necessary details can be coded in symbolic form, with the procedure equating our symbols to proper JCL values. Any value so defined can be changed merely by indicating the symbol and its new value on the execute card invoking the procedure. We can also add or override DD cards and DD card parameters by supplying additional cards containing values to be changed or added.

BUCKS AND DEBUGGING

Diagnosing bugs

Diagnosing bugs in user programs requires a clear understanding of the relationship between system services and problem programs.

Bugs call to mind complicated dumps and endless traces, I/O errors that aren't I/O errors at all, and other frustrating experiences. Certain higher-level languages include debugging facilities, trace facilities, and other diagnostic capabilities that can further complicate the diagnostic process whenever they are unable to properly field and identify an error.

Our problem, then, in debugging is the rapid reduction of bugs to their simplest terms, so that proper corrections can be easily and simply made. Here we see the need for informed diagnosticians.

Worthwhile procedures for debugging

Often there is more than one path to a problem solution. We should avoid the trial-and-error, pick-and-choose methods because they are expensive and generally unproductive.

Here is a quick overview of my formula for diagnosing abnormal terminations ("ABEND's").

First, examine the operating system's reported cause for the ABEND. Try to get a clear understanding of why the operating system thought an error occurred. If any point is not clear, consult the appropriate reference manuals. Research the ABEND description until it is understood.

Before progressing, ask these questions: Can I in general identify the instructions subject to this error? Can I recognize invalid address values that would cause this error? If either answer is yes, proceed; if no, dig some more.

Next, examine the instruction address register portion of the program status word (PSW) which reveals the address of the next instruction to be executed. Check the preceding instruction to see if it was the one that failed. If this process does not locate the failing instruction, perhaps the PSW address was set as the result of a branch.

Check each register at entry to ABEND. Do they look valid or do they look like data or instructions? Are a reasonable percentage of them addresses within our region of memory?

If register conventions are observed, tracing backwards from the error point might reveal where a program went awry. The beauty of higher-level languages is that they consistently follow some sort of register use convention. Once these are learned, debugging becomes simpler.

The process just described continues point by point backwards from the failure to the last properly executed code, attempting to relate the progress of the machine instructions back to the original language statements. If this process fails, attempt to start your search at the last known good instruction executed and work forward.

The same kind of process is followed with I/O errors and errors in general: first identifying the exact nature of the error which the system believes to have occurred; next identifying via register conventions pertinent items such as the data set in error—going as far back into the machine level code as necessary to isolate the error type.

I have a whole string of questions that I ask myself when debugging and it seems that I'm forced to dream up new ones constantly. Let me sum up my approach with four statements:

• Get a clear understanding of the nature of the error.
• Ask yourself questions that bring you backwards from failure point to the execution of valid code.
• If this yields nothing, try to approach the error forward, working from the last known valid execution of code.
PERFORMANCE PROBLEMS

The improvement of system performance and the elimination of bottlenecks has attracted wide attention of late, perhaps because economy and good business practice dictate that it be so. Unfortunately, no cookbook approach yet exists, and it remains up to us to discover one for ourselves. The tools are legion \( ^{5,6} \) and are sometimes quite expensive and difficult to interpret. The tools include accounting data, failure statistics, operator shift reports, various types of specially developed system interrogation reports, simulations, and hardware and software monitors. The availability of tools and the level of sophistication of systems personnel may dictate whether these tools are handled in-house or contracted out on a consulting basis.

Our first step is to outline each system resource to determine its fit into the overall system scheme and how our use may affect that fit. A manager might begin this process with a sort of time and motion study, eliminating as many handling problems associated with introduction of jobs to the computer as possible and smoothing the work flow between users, schedulers, messengers, operators, etc., and the computing system itself. The worst bottleneck might, in fact, be the one that prevents a tape or a card deck from being where needed when needed. Assuming that these things have been accomplished, we then poll the operators and the users for their impressions of how well the system serves their needs, at which time we might be forced to reexamine our work procedures.

Our next step properly belongs to systems programmers. Their study should concentrate on those aspects of the system that consume the greatest asset of all—time. There are many techniques and packages available that measure system activity, and these should be put to work. Since the system contains the most heavily used code, our systems programmer places the most active system modules in main memory, intermediate-activity modules in the fastest available secondary storage hardware such as drums, and lower-activity modules in larger-volume, lower-speed (and lower traffic-density) hardware, perhaps large disks or tape. The direct-access addresses of the most frequently fetched routines ought to be resident to eliminate searches for them, and where possible these data sets should reside on the devices with the fastest access speeds. System data sets on direct-access devices with movable arms should have the most active of these routines stored closest to their directories, so that seek arm travel is kept to a minimum. Educated trial and error is necessary before reasonable balance in these areas can be achieved.

Next we study each of the online storage facilities and their use. Questions as to adequacy, reliability, method of backup, and recovery ought to be asked. Criteria for the allocation of direct-access space should be established based upon criticality of use, volume and frequency of use, and cost-effectiveness when compared with available alternatives.

Follow this with the determination and elimination of unused facilities which should be identifiable through the tools previously mentioned.

After our system examination comes an examination of user programs and processes. In this part of our improvement cycle we first look at production programs, starting with the heaviest users of computer time and resources. Many times we find them in an unfinished state, with improvements possible through the elimination of unnecessary steps. An important item to observe is the use of unblocked records or the operation of production programs from source rather than from object or executable load modules. Production programs that require the movement of data from card to disk or tape to disk preparation to use should be brought to bear upon production programs and other user processes.

Production programs should be examined to determine where the most CPU time is spent in executing and, consequently, what code could be improved to yield the best results.

With OS 360, users can reduce wait time, system time, channel time, and device time by assembling records into blocks set as close as possible to addressable size. This reduces the number of times the I/O routines are invoked, as well as the number of channel and device requests and the seek time expended. With blocking, fewer I/O operations are needed and our programs spend less time in a nonexecutable state waiting on I/O completion. We gain additionally because blocking permits greater density on the storage devices. Many users are unaware of the fact that gaps exist between every record written on a track of a direct-access device. For example, the track capacity on an IBM 2314 disk is 7294 characters. If 80-byte card images are written as one block of 7280 characters, only one write is required to store 91 records on a track; yet if these records are written unblocked, only 40 records will fit on a track because of the inter-record gaps, and 40 writes are invoked to fill that track.

Using large block sizes and multiple buffers introduces additional costs in terms of increased memory required for program execution. Obviously, we should balance these somewhat conflicting demands.

A frustrating problem encountered by users of some systems is that of proper allocation of direct-access space. Since printed or punched output is temporarily stored on...
occupy that space long after it is no longer used, simply the user's program. Subsequent tapes when earlier tapes are completed. Further, users should free tape drives not being used in subsequent steps.

Proper tape handling is mainly a user problem. Blocking should be employed for efficiency's sake. Data should be blocked to the largest sizes possible, consistent with memory availability, to reduce the amount of tape required to contain a data set and the average I/O transfer time consumed per record. Use of the highest densities provides for faster data transfer and surprisingly greater accuracy because of the built-in error recovery available with high density techniques.

To protect tapes from inadvertent destruction the use of standard-label tapes is encouraged as a site-imposed standard. This permits the operating system to verify that the tape mounted by the operators is the one requested by his JCL. Jobs can abnormally terminate if insufficient space is allocated to data sets being created or updated. Over-allocation is wasteful and reduces space available for other jobs, as well as permitting excessive output to be created without detection. The space required should if possible be obtained in one contiguous chunk, as less CPU time and access time are used than if data is recorded in several pieces scattered across a unit.

Another problem is locating files or even individual records within files. The system provides catalogs to point to the unit upon which a specific data set resides, but improper use or nonuse of these catalogs or of suitable substitutes can prevent a job from executing, due to an inability to identify where the data set resides. The use of a catalog introduces the problem of searching the catalogs for the individual data set's identity; and if these catalogs are excessively long, useful time (both CPU and I/O) can be lost, since every request for a data set not specifically identified as to unit results in a search of the system catalogs.

Because of the changing nature of user requirements, a data set occupying permanently allocated space might occupy that space long after it is no longer used, simply because we are unaware of the fact. Techniques exist to monitor space, but users can easily cheat them.

Proper estimation and allocation of direct-access space needs is a must, as is the release of unused or temporary space as soon as its usefulness has ceased. At nearly any installation one can find unused data sets needlessly tying up valuable space and sometimes forcing the system to fragment space requests due to the volume of space so wasted.

Proper tape handling is mainly a user problem. Blocking should be employed for efficiency's sake. Data should be blocked to the largest sizes possible, consistent with memory availability, to reduce the amount of tape required to contain a data set and the average I/O transfer time consumed per record. Use of the highest densities provides for faster data transfer and surprisingly greater accuracy because of the built-in error recovery available with high density techniques.

To protect tapes from inadvertent destruction the use of standard-label tapes is encouraged as a site-imposed standard. This permits the operating system to verify that the tape mounted by the operators is the one requested by the user's program.

When processing multiple volume files, two tape drives should be allocated, if available, to permit a program to continue processing rather than wait for the mounting of subsequent tapes when earlier tapes are completed. Further, users should free tape drives not being used in subsequent steps.

Systems programmers usually provide for blocking of card input and certain output types through default values and control card procedure libraries. Users ought not to unblock this I/O.

Careful reduction of the volume of printed data to that actually needed by the ultimate recipient serves both the user and the system by reducing output volume. A typical high speed printer can consume some 35 tons of paper a year, and I can't even estimate the average consumption of cards for a punch unit. Perhaps, like me, you flinch when you observe the waste of paper at a typical computer site. To further reduce the waste of paper, users are well advised to go to microfilm where these facilities exist, particularly for dictionary type output.

It is amazing how many users are still dependent on punched cards in this generation. Processing large volumes of cards requires many extra I/O operations and machine cycles that could be avoided by having these data sets on tape or disk. True, to update a card deck one only needs to physically change the cards involved; but the use of proper update procedures with tape or disk is a far more efficient and accurate use of computer and human time.

This brings us to the subject of software tool usage. The most frequent complaints associated with vendor-supplied utilities are that they are difficult to use and that the documentation is unclear. This is probably due to their generality. Another difficulty is finding out what is available.

To answer the difficulty-of-use problem, we might point out that it is simpler and more productive to try to understand the utilities available than to write and debug new ones. A small installation cannot afford the luxury of user-developed utilities when existing ones will do the job. Often it is better to search for what one is sure must exist than to create it. Still another avenue to investigate would be whether other installations might have the required service routine developed by their users.

Since available utilities are usually more or less unknown to installation users, consideration might be given to assigning a programmer the responsibility of determining the scope of the utilities available and how to use them. This information could then be passed on to fellow programmers. In fact, a good training course would justify its costs by eliminating unnecessary programming and enabling installations programmers to select and use utilities that perform trivial tasks quickly. With vendor-supplied utilities, the first use or two might appear difficult, but with use comes facility.

The use of programming languages ought not be an ego trip. Programmers should be forced to practice "egoless programming" and to follow recognized practices. Efficiency dictates that programs be written in modular form and that they be straightforward, well documented, and without cute programming gimmicks or tricks, lest the next release of the operating system render the program nonexecutable. Programmers themselves should realize that they cannot escape later responsibility for the programs "tricky" programs, as long as they work for the same employer.

Our evaluation process then continues with research into how new programs and problem program systems are
tested and developed. It is the writer's experience that only rarely has much consideration been given to the efficiency of program development and testing. Frequently the heaviest consumers of computer resources are programmers with trial-and-error methods of debugging and complex or "clever" coding. Again, understanding the system can yield us an insight into relative inefficiencies of program development and how they might be overcome.

Once we have attempted everything within our means, we might then consider outside performance improvement or consulting services.

A final suggestion on resource consumption is a point regarding cost allocation. If a resource does not cost the user, he or she is not likely to try hard to conserve it. Proper allocation of cost in relation to resource use is both a form of control and an attempt to influence users to adjust their demands to the level most beneficial to the system. In short, users ought not to be given a free ride.

A CASE FOR SPECIALISTS

Merging the many parts of a third generation computing system into an effective problem-solving instrument requires that we inform the system of a myriad of details. Efficiency and economy in their most basic forms dictate that we simplify our work as much as possible. Some way must be devised to supply to the system the detail it needs without typing up some 40 percent (as has actually happened—I cannot bear to cite a reference) of programmer time with JCL and associated trivia. What we are striving for is a lessening of the applications programmer's need to know operating system details.

As already noted, the first step is to have knowledgeable system programmers supply as many efficient procedures and other aids as possible and to have them generate a responsive system. Also, those same systems people can generate libraries of control language to cover all the regular production runs (and as many of the development and other auxiliary processes as are practical after sufficient study).

I propose, however, that we go one step further in this area.

Chief programmer team

One exciting concept to emerge recently has been that of the Chief Programmer Team. Significantly increased programmer productivity and decreased system integration difficulties have been demonstrated by the creation of a functional team of specialists, led by a chief programmer applying known techniques into a unified methodology. Managers of programming teams would do well to study this concept. Properly managed, this process has the programmer developing programs full-time, instead of programming part-time and debugging JCL part-time.

Programmer assistance

Our examination of the use of third generation systems has continually pointed out the need for including sufficiently knowledgeable people in the process of system use. A focal point for users needing assistance should be created. This is usually done anyway informally, as users search out fellow programmers and systems people who might have answers to their problems. I propose that experienced, knowledgeable, and systems oriented people be organized into a team to answer user questions and to provide diagnostic assistance. This same group could aid in developing standards, optimizing program code, and teaching courses tailored to user needs. My own experience in a programmer assistance center has shown that such services greatly increases the productivity of a DP installation's personnel.

Diagnostic services

A cadre of good diagnostic programmers should be used to assist programmers who are unable to isolate their program bugs or who need assistance with utilities, JCL, or any other aspect of the operating system. Such a group should keep a catalog of the problems encountered for handy future reference and as a way for determining personnel training needs or system enhancements. The group could aid in locating and correcting software errors by creating small kernels or test programs designed solely to re-create the error. Through the use of such kernels, various error solutions could be tested without disturbing the users main program or process.

Program optimization service

These same personnel might also be charged with research into and development of simple and efficient programming techniques. These techniques could then be implemented in the optimization of the most heavily used programs or systems. Once we identify our largest program consumers of computer time and their most heavily used routines, we can find it cost-effective to thoroughly go over such code, replacing the routines identified with more efficient ones.

Known programming inefficiencies and their correction might be identified by an occasional thorough review of the computer-generated output. For example, the entire output of a weekend might be reviewed in search of poor use of I/O processing. Runs with poor system usage might then be earmarked, and the programmers responsible, notified and given suggestions for possible improvements. The most frequently encountered poor practices might then be subjects of a tutorial bulletin or training session for programmers.

Conventions and standards

"Standards" is a dirty word to many people, but when large numbers of programming personnel are found to be
employing poor practices, our systems group would be charged with developing optimum alternatives. Effective streamlining of frequently used facilities could be accomplished through the publication of tested techniques and standards or conventions. With standards for guidance, the user has a yardstick to determine if his utilization of system resources meets a minimum acceptable level. Prior to the design of new problem program systems, these standards would play an important role in ensuring that optimum use of available facilities was achieved.

“Structured Programming”10,11 and other developing techniques for making the programming practice manageable should be researched and used as the basis for developing usable installation standards.

Instructors

The accumulation of expertise within a single group would be almost wasteful if this knowledge were not disseminated among the users of the system. A logical extension to our assistance group, then, would be the assignment of instructors who would conduct tutorial seminars and develop tailored training courses and user manuals.

SUMMARY

I have attempted in this paper to give you my views on coping with a highly generalized operating system. I have found that complexity is the main problem facing users of third generation systems. My formula suggests that we insulate the general user/programmer from OS detail to the maximum extent possible, and that we provide the user community with a technically competent consultants group to assist in fielding problems with which the user need not be concerned.

ACKNOWLEDGMENTS

I wish to extend my sincere thanks and appreciation to Messrs. Evmenios Damon and Chesley Looney of the National Aeronautics and Space Administration, Goddard Space Flight Center; Carl Pfeiffer and Ian Bennett of Computer Sciences Corporation; Dave Morgan of Boole and Babbage, Inc.; and John Coffey of the National Bureau of Standards. In particular, I wish to thank Herb Bright of Computation Planning, Inc., for his many hours of discussion and words of encouragement.

REFERENCES