ACM

Symposium on Operating System Principles

October 1-4, 1967 Gatlinburg, Tennessee

A PHILOSOPHY FOR COMPUTER SHARING

Edgar T. Irons
Institute for Defense Analyses
Communications Research Division
Princeton, New Jersey

Association for Computing Machinery 211 East 43 Street New York, N.Y. 10017

A PHILOSOPHY FOR COMPUTER SHARING

Edgar T. Irons

Institute for Defense Analyses
Communications Research Division
Princeton, New Jersey

The remarks which follow concern four design objectives for a shared computer system, and their relation to the selection of computing equipment and the design and programming of a time-sharing program at IDA-CRD during the last year. The principle objective was to bring as much computing power to our users as possible, consistent with budgetary bounds, and to provide as intimate a connection to the computer for each user as possible. A second objective was to provide a computing environment which allows each user the maximum freedom and control over the machine possible within balancing of our third objective, namely, maximum reliability of the system. Our fourth objective was one which we choose for any program of a lasting nature; that the programs be as simple as possible to still achieve their purpose. Adhering to this maxim usually, although not always, reduces the time involved in realizing the programs; and in addition, holds promise for increasing their reliability and in any event reduces maintenance problems if only by reducing the possible number of wrong symbols.

Summarizing our objectives for the computing system and time-sharing programs, we wished to achieve, in order of importance,

1. Maximum computing power and availability to the individual,
2. Maximum flexibility compatible with
3. Maximum reliability, and
4. Minimum complexity.

To fix ideas, we shall describe briefly the equipment chosen, and some aspects of the software implemented, and then address in more detail the four items listed above.

Hardware

The computer chosen was a Control Data 6600 computer with 65,000 words of high speed core storage, and 500,000 words of extended core storage, 16,000,000 words of disc storage, and one cathode-ray-tube entry-display station for each user. The computer was chosen because it seemed clearly superior in computational power to any other available, and was by our best judgement able to provide the most computation per dollar. The display stations were chosen as the principle means of communicating with the computer because they offer the closest connection between a user and the computer of any devices which are inexpensive enough to install in quantity. Although they cannot point plot, and are rather limited in character repertoire (64 character font), devices with significantly more capability cost ten to twenty times as much and were clearly too expensive for quantity installation.

The Time Sharing Software

The principal feature of the time sharing system which resulted from the above motives is that the privileged part of the system, that is, the part which has the power to destroy more than one person's property, has been kept to the absolute minimum necessary to effect time sharing and keep users from destroying each other. The general layout of the system and user programs is depicted in Figure 1. The computer memory has a small section devoted to "the system," a somewhat larger space allotted to hold a directory of all files. The rest of the memory is devoted to "user programs." The system has access to the entire memory and all input-output equipment, as it indeed must have in order to be able to perform its functions. Each of the user programs, however, has access only to its own memory. Generally speaking, we expect that the number of programs in core will be approximately equal to the number of active users and hence to the number of entry-display stations. The programs are more or less permanent in that they come into existence when the user logs in and remain until he logs out, which in many cases may be never. The amount of storage devoted to each user program, however, varies frequently according to the demands of the program. In addition, the system may, if it needs

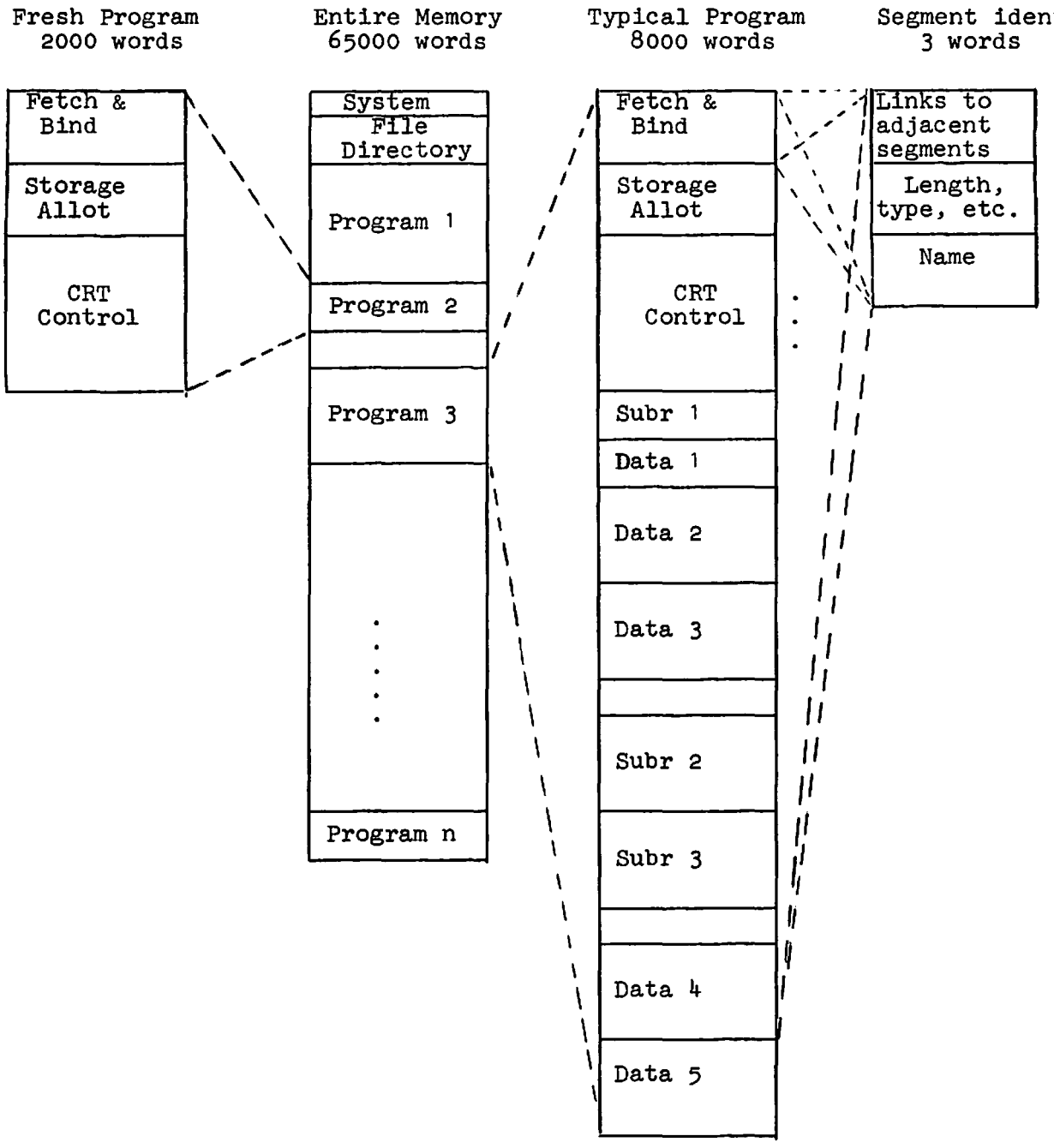
Fresh Program
2000 words
Entire Memory
65000 words
Typical Program
8000 words
Segment ident
3 words
Fetch & Bind
Storage Allot
CRT Control
System File Directory
Program 1
Program 2
Program 3
Program n
Fetch & Bind
Storage Allot
CRT Control
Subr 1
Data 1
Data 2
Data 3
Subr 2
Subr 3
Data 4
Data 5
Links to adjacent segments
Length, type, etc.
Name

Figure 1

space for another program, save the majority of a program on a backup storage device; and retain only a small part of the program which identifies it and gives information useful for scheduling, etc.

Things the System Does

The functions performed by the system are the following:

1. Scheduling programs for execution.
2. Responding to an error (program stops, time limit, etc.) by returning to the program at an address supplied for the purpose, or if none has been supplied, replacing the contents of the program by a standard "fresh program" and the executing at the error entrance.
3. User requested actions as follows:

a. Input-output, or more precisely, information transfer. In each case, N words of information is transferred from medium MA, address AA to medium MB, address AB, where N, MA, AA, MB, AB are supplied by the user but checked by the system. Media include internal core storage, external core storage, disc, entry-display stations, and other input-output devices such as printer, tape, plotter, card reader, etc.

b. Look up a file name and authorize its use as an input or output medium.

c. Create a new file and authorize its use as an input or output medium.

d. Delete a file.

e. Change the directory entry for a file (e.g., its name or, if possible, its length).

f. Allot more (or less) space to a program.

g. Establish a test for some condition (the time, input-output progress, status of some external equipment, etc.) and an address to go to in the program when the test succeeds.

h. Wait until a test (as in (g)) passes.

i. Create a new program or destroy an existing program.

Things the System Does Not do

We now list some functions which might be thought of as system functions; but which, in this case, are treated as user operations in that the programs performing them are in the users program area.

1. Data formating in input-output operations

The system functions which perform input-output (or more properly transfers of information) are designed to reflect as completely and accurately as possible the hardware device with which they communicate. Hence, for example, if a user calls for a card to disc transfer, the information transferred to the disc is a direct copy of what comes from the card reader hardware, packed together. The data is not altered. Where it is the case that data formats for different devices do not match, for example in natural record length or character codes, it is necessary to transfer the data from one device into the computer (probably dia the disc), reformat it and then transfer the reformatted data to the second device. The system has no prejudices about data formats, character codes, etc.

2. Double buffering

The information transfers effected by the system are not double buffered in the case where one of the media is core storage. Requesting an information transfer, however, does not complete the transfer before returning; but only starts it, and since in the 6600, the peripheral processors execute all the input-output operations, a program may continue to execute while input-output is in progress. So the user program may (and through standard subroutines we expect nearly always will) care for the buffering, and in fact, by measuring wait times can even adjust buffer sizes to the optimum for effective buffering.

3. Fetching and binding of subroutines

All management of the users allotted storage area is in charge of the users programs. In the beginning, (or after an unrecoverable error) the user is given a standard "fresh program" by the system. This fresh program will contain perhaps a thousand words of program having the capability of rudimentary communication with an entry-display station, of managing storage allocation in the users program area, and of binding (reading from external storage and relocating) programs. Hence, the fresh program is a sort of minimal control program with facilities for fetching more sophisticated programs. Generally, we will expect that the storage allocation and binding programs will remain in the users core during its lifetime, however, this is not required.

4. Interpreting instructions or command languages

The system recognizes no external requests and places no interpretation of information transmitted to the users program. The user programs perform such

functions, probably through standard subroutines.

5. Error Analysis

Errors resulting from improper system calls, time limits, illegal instruction executions, etc., are recognized by the system; but the resulting information is passed back to the user programs as a number, and further interpretation or subsequent action is left to the user programs.

Philosophy Behind the System

We now consider in more detail the design objectives outlined earlier. While we hope that the IDA-CRD computing system implements these objectives in a satisfactory way, we would not suggest that we have given the only way to achieve our aims; indeed, even a somewhat different measure of importance meted out between the four could require a rather different approach. However, while we shall discuss the design criteria as generally as possible, we shall also draw on the IDA-CRD system for examples, and to illustrate some specific directions suggested by the criteria.

Computing Power and Availability

In selecting computing hardware and an overall design for operating software, we feel that operating in a time shared environment is the obvious choice for bringing the most computer closest to the user. In addition, of the various devices available for communicating with a computer, we feel that the Cathode Ray Tube entry-display stations are clearly superior to other devices available because they allow a substantially higher rate of information transfer in an interactive environment. With these criteria as a basis for hardware selection and software design, one has yet a few difficult choices.

Computation power, paging hardware, and extended core storage are the three aspects of currently available computer systems which occupied most of our attention at IDA-CRD in selecting hardware. Unfortunately, no one computing system excells (or indeed contends) in all three of these areas so one must try to assign some figure of merit to each. Extended core storage, especially when highly leaved to allow very fast transfer rates, tends to balance against paging hardware in that the former increases transfer rates between primary and secondary storage while the latter generally reduces the amount of information required to be transferred. The combination of superior computation power and the availability of high performance extended core storage and disc file systems convinced us that the CDC-6600 was the best hardware for the system at IDA-CRD.

Flexibility

We firmly believe that a program which has such a substantial influence over the everyday use of the computing facilities as does the operating system must be capable of evolving. It is simply impossible to state in advance of use of a system which features will be good and which bad.

In addition the way in which users exercise the operating system changes substantially with time, so that what was once a useful feature may become an obsolete or even detrimental one, and facilities which were once not required may become essential. Consequently, providing a means for the system to evolve gracefully, that is without frequent system crashes, was a major concern in the design of our new system.

Our experience has also shown that there is almost no rule imposed by an operating system for which one cannot at some time find good justification for breaking. Thus another important objective was to design the system so that most of the standard modes of operation can be overridden when required by one user without damage to the others.

The following are some specific areas of flexibility which we consider particularly important.

1. The user should have command of as many facilities built into the hardware as possible, for example, of input-output equipment. It is necessary for the system to control access to such equipment so that two users do not intermingle their output line by line on the printer for example, and is probably advisable to be a bit careful about allowing any programmer to do arbitrary things with all the equipment. However, there is no reason why one user should not be allowed to use a printer on line if there is no other demand for it (e.g., at night), or why one should insist on automatic page headings to prohibit printing continuously on a ribbon of paper. Items of this nature should be invoked as the normal mode of operation but should not be irrevocable. Another aspect of the hardware which might at first sight appear impossible to give a user access to is the memory protect feature on most modern machines, or in some, the paging

hardware. It is true that one cannot offer the user complete control over these features, but it may very well be possible to offer him control over a very useful part of them. For example, in the 6600 it is possible to allow the user access to the memory protect mechanism in the following way: Allow the user to shrink or expand the protect bounds for his program arbitrarily as long as they do not extend outside his programs region. Such an option allows the user to program sub-systems which can protect themselves from the whims of untested programs. In a paged machine, it might be very profitable indeed to allow the user to manipulate the pages (under checks made by the system) in some cases, particularly since the program can very often predict future requirements — something that is usually much more difficult for the system to do.

2. The user should be able, if he chooses, to exercise the privilege of taking his own action to various errors which the system may detect. In the usual case, standard sub-routines can be supplied to display information about the error and what caused it. It should be possible in every case for the error to be bypassed, that is for the erring program to be allowed to continue to run if there is even any remote possibility that this will be useful. Even if there is no possibility for the program to be able to continue; e.g., it has been badly damaged, all of the programs memory and its last state should be maintained for interrogation until the user releases them. It would be possible for the system to deal with errors within these rules; but we feel that it is both easier and more natural to allow the user programs (or standard sub-routines in the user area) to determine error reactions. Such a philosophy allows the user to escape from standard reactions, as his desire many times dictates, or even to change the standard reactions for himself. Using this method of coping with errors, requires that the system be able to recover from the situation in which the error recovery mechanism has an error so that a loop is formed trying to recover. Such loops can be prevented by detecting them and reloading into the programmers area a "fresh program" which is known to have a correct error recovery mechanism. In such a case, the memory area required by the fresh program should first be saved so that the original erring program can be reconstructed for further execution or interrogation.

3. Scheduling flexibility. We believe that the scheduling facilities for a time sharing system should have great flexibility built in and lament that we have not been able to see a better way to gain the flexibility without requiring a system program modification to effect it. Our aim is to have the scheduling mechansim so arranged that the current users of the machine could, if they wished, depart from the standard mechanism completely and choose their own. Such facility would be particularly useful in periods where only a few users were present, or some programs had predictable and peculiar characteristics which made the standard algorithm less than optimum.

In addition, it would provide an excellent laboratory for experimenting with new scheduling algorithms, and a means for modifying a standard one to meet the inevitable changes in demand pattern on the system. We propose that such facilities could be implemented safely by providing a basic scheduler within the dangerous part of the system which uses a simple, but foolproof algorithm such as round-robin, but which can be instructed to call upon a user program to supply a sophisticated schedule. The basic algorithm might apply some crude tests of reasonability to the schedule supplied and revert to its own algorithm if it rejected the user supplied schedule. Certainly it would be required to have a safe means of shutting off the user supplied schedule in the event it goes berserk: a hardware switch could be used for that purpose.

Whether or not a mechanism of this flexibility is used for scheduling the system should allow, indeed encourage, programs to supply as much information as possible which might be useful for scheduling purposes. It is the case that if the actions of all programs could be predicted completely, it would be possible to achieve an optimum scheduling mechanism under most criteria. The unpredictability of a program's future course, or inaccuracies in the predicted course are the demons which make fair scheduling difficult. One should encourage then, user programs to supply information which may be at hand such as a lower or upper bound, or both, for the expected wait time for an input, or a continuous estimate of running time required before the next output. Supplying such information should not be required of the customer, for this would surely be an intolerable burden in an interactive environment; but we should encourage him to supply as much information as he can without overworking himself.

4. Completely dynamic storage allocation should be available within a program. In machines with paging hardware, some of this requirement is solved by that hardware. In machines with a base register, like the 6600, the problem is solved for inter-program allocation in that programs may be moved around relative to one another and be still executable by modifying the base registers for the programs. In other machines, and with individual programs on base register machines, however, no hardware mechanism is available for allowing data items and programs to be moved relative to one another.

However, it is possible to provide an economically reasonable means to allow data items to be moved at execution time by keeping a list of all references to these data items and modifying each reference to an item when the item is moved. We have observed that with suitable encoding, the space required to retain references is on the order of one percent of the space occupied by a program. Keeping enough information around to be able to move programs themselves we suspect would also be economically reasonable, but we have not as yet taken this step. In any event, allowing the moving of data items allows them to grow or shrink in size more or less arbitrarily; and we believe the additional cost in space required is well worth this facility. Note that no deterioration in execution time is required since addresses referring to data items are themselves modified so that no indirect addressing is required. In addition, if it becomes known that a data item is not going to be moved, the additional information being kept in order to be able to move it could be discarded.

5. Control over establishment and release of program segments (sub-routines) or data segments should remain in the hands of the user because he can predict, often with great accuracy; what segments will be needed when. He should, in such cases, be allowed to specify when they are to be fetched or released. In a paged environment, where segments are shared by several programs, the situation becomes more complicated; however, allowing the user to advise the system of future demands clearly permits better storage management. In a non-paged devoted copy environment the user has the option of fetching or establishing what he wants as computation proceeds. We envision this facility being very useful in, for example, fetching pieces of a compiler as the compilation proceeds, allowing the first phases of the compilation to begin with only a small part of the compiler present; and, if the compilation does not stop for errors, fetching the rest of the compiler as the compilation proceeds. In addition, compiler organization which places mechanisms for different language features in independently fetchable pieces can allow simple programs to be compiled by fetching only a small part of the whole compiler.

6. Control over the "system." In the IDA system, the user has almost complete control over the "system" where we here use the word in the larger sense of the collection of programs (now both those in the system area and user area), principally because most of the functions normally thought of as system functions are programs in the users area; and hence, subject to modification or replacement by the users individually. This element of flexibility is valuable for users who wish to depart from normal conventions because of special needs. It is indispensable in a system which is expected to grow and change with time. Such flexibility is particularly important, we believe, in areas closest to the programmer such as the mechanisms available to him for modifying files, entering programs, or program segments, calling for compilation and execution of programs. Having excellent control over these areas should allow orderly development of better programming aids.

Reliability

We shall consider two aspects of reliability; namely, prevention and cure of accidents. On the prevention side, we consider:

1. System design. Generally speaking, the principal design feature of the IDA system for prevention of "system crashes" is to attempt to isolate the effect of the accidents which inevitably happen. By completely isolating user programs from each other and by including as much of the programming as possible in the user area, rather than in the system area, we have attempted to make the effect of most accidents reach only one user. Most of the bugs in system programs are of the kind that arise from unusual uses of the programs (the rest are eliminated before one would call a system operable). For a "system" program which is in the users area, the bugs show up and are dangerous only to the one person who is doing the unusual; and one would expect that when such a bug showed its face that it would be possible to fix it or at least warn the rest of the community before any one else is effected. At any rate, the rest of the users are not wiped out by a system crash. While

having dedicated copies of system programs (the ones in the users area) is the means of achieving the isolation of the effect of system bugs, it is clearly possible to achieve at least a measure of this isolation in other schemes, and we believe this principle to be a very valuable one.

2. Programming techniques. Most techniques for writing reliable programs are concerned with care in programming and thoroughness in testing. We offer two additional suggestions. The first is that more suitable languages than machine language should be used for programming. While most of the computing community has accepted the value of using higher level languages for some time, it is only recently that systems programmers have begun to use them. The increased ease of programming and debugging accrued from the use of such languages is of course, important; but the increased reliability of the resultant programs is of the greatest value here, where bad programs have such far reaching, and often such expensive effects. Although something must be said against attempting to install a compiler and a time-sharing system at the same time, the author feels most strongly that his use of a programmer oriented language in this area is not only justified but will prove itself many times over in enhancing the maintenance, reliability and expandability of the system.

A second principle of reliable programming is to sacrifice complexity for reliability. It often happens in programming, systems or otherwise, that one is confronted with a choice between two or several designs for some generally formulated feature of the overall program. For example, there are many techniques available for various aspects of directory management for a file system. The programming for some of these may be substantially more complex than for others, and worse, it is sometimes not apparent when a file system is being designed which is the more complex to program, to say nothing of how much more complex. In addition, there are tradeoffs between complexity and user facilities involved in these decisions. Allowing the user a little more facility in some area, may complicate the programming substantially. We suspect that many features of complex systems turn out to be used perhaps .01% of the time in the life of the system and then only instead of slightly more awkward alternatives. It is of course extremely difficult to predict which features may be the most valuable and which the least; but we feel that careful thought about possible alternatives for features in systems which require complex programming to implement them is well worth the effort. In other words, continually ask, "is there a simpler way?"

3. The cure. When the system does crash, what then? Of course the system never crashes completely, that is, so that nothing at all is left. But, when it does crash, what remains and what has to be done to recover the rest? The answer is that in every system some sort of backup mechanism is available; in many systems several levels of backup are provided, each one being successively more reliable, but further removed from the state of the machine at crash time. Many questions arise concerning what backup devices should be used, how frequently the information on them should be updated, and so on. Two items of experience, however, stand out in the authors mind: They concern local identification, and some remarks about the lone programmer.

On the subject of local identification, we cite experience with the last system at IDA which used disc files for its primary external storage and tape copies of the disc files to back up the discs. Most of the difficulties with this approach stemmed not from loss of data but from loss of data identification. It never occurred in three years that any significant amount of data stored on the disc files was destroyed, but it happened much more frequently that directories were lost; and if you can't find the data, it doesn't do much good to have it around. On top of that, tape backups of the discs had no local identification, but were merely copies of the discs. Hence, a dropped or bad tape record could cause havoc in trying to recover disc contents. Our suggestion is that disc files are probably very reliable backup devices in themselves; in fact, in our experience we feel that the discs are considerably more reliable than tape from the hardware standpoint. Furthermore, in most systems, even if a malicious bug set out to erase the entire discs, it would take long enough to do so that the gnawing would be noticed long before much damage was done. However, directories can be volatile. Hence, the principle we suggest here is to intersperse all data stored with some recognizable identification in such a way that the identification takes up perhaps one or two percent of the storage space; that is as much interspersed identification as possible without an outstanding economic burden. Some careful attention to redundancy in such identification would certainly be useful.

On the other point of recovery techniques, it is instructive, we think, to observe a lone programmer using a computer with no time sharing or automatic operating system but with programs and data etc., in some quantity which are stored in the system. That is, observe the user of a time sharing system with no competition, and no backup mechanism. Let us assume that the programmer is repairing self bootstrapped compiler (a very dangerous thing to do). His action for safety will probably be to make safe copies of a part of his program which is to be replaced by a new part on a medium (like discs) which is on-line, but fairly dependable, every time a new item is to be tested. If he is a bit cautious, he will tend to make a safer copy of things every now and then perhaps on tape. The actions he takes really follow a very simple equation: balance the time it takes to make a safe copy with the time it would take to recover were the safe copy not made. To extract something useful from this behavior pattern for the organization of a time sharing system, we suggest that perhaps the most effective mechanism and the most satisfactory one to all is to provide the same possibilities for the users as exist for the lone programmer, namely, let him choose the time and medium on which backups are to be made. For example, provide two buttons on the entry display stations which provide for a backup copy to be made, of a file, one to make a copy on disc, one to make a copy on tapes reserved for the purpose. We suggest that the user will find this not only not burdensome, since it takes only one button push each time; but also more reassuring since he knows what is safe and how safe it is. Furthermore, the user knows what is valuable and how valuable it is, something the system has a very difficult time discovering; and hence, is much more capable of making sensible decisions about backup.

Complexity and Cost

According to the discussion on reliability, we advocate keeping the complexity of the system at a minimum. Such a philosophy obviously affects the utility of the system, although we tend towards keeping a complex mechanism if no simpler way can be found for implementation without substantially reducing the power of the system. Cost is also affected by the complexity, and in two ways. Obviously, the less complex system will cost less to program and probably will cost less to maintain. The IDA system, for example, which emphasized the minimum complexity approach cost approximately one man year to program including all the user "standard" subroutines for buffering, binding, storage allocation, error reactions, editing, etc., and approximately one man year more to adapt the manufacturers assembly programs, FORTRAN compiler and a machine independent SNOBOL compiler. On the other hand, reducing the complexity of the system can also decrease the running efficiency of the system. This cost factor is extremely difficult to estimate, often even very difficult to measure even between two established systems. However, we can cite a few examples. One example has already been touched upon; namely, that completely dynamic storage allocation can be had at a moderate increase in cost. The cost of storage allocation could be reduced somewhat by allowing in addition (we would not advocate a system with no dynamic allocation) perhaps a push down dynamic system and a fixed allocation option; either of these requires less storage than the completely dynamic system, but at a price for added complexity. Since the dynamic system includes both, we consider the added cost worthwhile.

A second example is the cost of duplicating sub-routines in internal core storage when two users require the same sub-routines, rather than sharing a copy (or at least the non-private part of a copy). While the choice of such a system was partly dictated by some of the reasons mentioned in previous sections and partly because of the structure of the machine chosen, at least some of the motivation for not sharing sub-routines was the very much less complex program mechanism required for the non-shared system. The addition of Control Data's extended core storage to our system will certainly overcome any penalties in time for reading extra copies of sub-routines; but of course, such a device is not free and part of its cost must be charged to its use for sub-routine storage. We feel, however, that the additional cost for maintaining private copies will easily pay for itself in offering the advantages outlined above.