MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

May 1968 MAC-M-372

SOME CONSIDERATIONS OF SUPERVISOR PROGRAM DESIGN FOR MULTIPLEXED COMPUTER SYSTEMS*

by F. J. Corbató
J. H. Saltzer

Abstract

One of the principal hurdles in developing multiplexed computer systems is acquiring sufficient insight into the apparently complex problems encountered. This paper isolates two system objectives by distinguishing between problems related to multiplexing and those arising from sharing of information. In both cases, latent problems of noninteractive systems are shown to be aggravated by interacting people. Viewpoints such as reversibility of binding, and mechanisms such as segmentation, are suggested as approaches to acquiring insight It is argued that only such analysis and functional understanding can lead to simplifications needed to allow design of more sophisticated systems.

PREPRINT

This memo is a preprint of an invited paper to be delivered at the International Federation for Information Processing 4th Global Conference, Edinburgh, Scotland, in August 1968.

* Work reported herein was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01) Reproduction in whole or in part is permitted for any purpose of the United States Government.

IFIP CONGRESS 68 February 29, 1968

PREPRINT

SOME CONSIDERATIONS OF SUPERVISOR PROGRAM DESIGN FOR MULTIPLEXED COMPUTER SYSTEMS*

by F. J. Corbató†
J. H. Saltzer†

Workers developing multiplexed computer systems, whether multi-programmed, batch processing, or time-sharing, have encountered a great deal of difficulty‡. This difficulty has been overcome neither by "human waves" of system programmers nor by teams of a few brilliant performers. Much of the difficulty probably stems from a need for insight into the principles underlying the multitude of mechanisms commonly comprising such systems. One reason for the lack of insight has been that system programming is so new that conventional tools for developing insight have not been brought fully to bear. Particularly

---

* Work reported herein was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

† Massachusetts Institute of Technology, Cambridge, Massachusetts U.S. A.

‡ Examples of such multiplexed systems currently under development or in use include OS/360, for the IBM 360 line of computers [2], TSS/360, for the IBM 360/67 [3], Multics, for the General Electric 645 [4], and EXEC-8, for the Univac 1108.

powerful tools are (1) strong modularity with clearly identified and functionally appropriate properties attached to each module, and (2) economy in the number of mechanisms so that a few do most of the work. A further reason why the problems have been difficult to unravel has often been confusion as to which objectives are wanted. With the exception of a few papers such as the recent one by Dijkstra [1] very little light has been shed on the differing objectives of different parts of supervisor programs.

One useful approach is to isolate the objectives of the system as well as possible so that whatever insight can be gained from the resulting less complex problems can be brought to bear on the system as a whole. As an example of this approach, this paper distinguishes sharply between two kinds of ideas encountered in a modern day computer system. One of these ideas, automatic resource allocation, we name multiplexing; the other idea, conscious communication and exchange of programs and data, we name sharing. For these two ideas in turn, we discuss what they are, why they are wanted, the kind of complexity they generate, and a specific example of organization which helps provide insight. The first topic is multiplexing.

Multiplexing and Its Contributions to Complexity

A basic question is "Why should one attempt to multiplex computing equipment at all?" The argument has three steps: (1) The

range and quality of hardware, supervisor, and library services that one desires for his full collection of problems is usually quite large. (2) Any single calculation is usually organized in a rather serial fashion and cannot make effective use of all components of the equipment and features of the operating system simultaneously. (3) There is a strong economic incentive to use the otherwise idle components and computing time for other users with calculations which similarly do not by themselves exhaust the resources of the system. This economic incentive is increased in an interactive system where a user has human reaction times and periods of thinking. Thus, a first level of complexity is injected by economically motivated multiplexing of the computing system.

The simplest form of multiplexing is with fixed time and space allocations. That is, each user is served by a processor in turn with fixed and unvarying time periods. Each user has a fixed invariant section of secondary storage at his disposal, and preassigned tape drives, card readers, printers, etc. The result is that each user receives a small virtual machine with an apparently slow processor and fixed equipment assignments; however, the problems of organization have been limited. In return for simplicity of organization, one pays a fairly high price, since there is no way to reallocate unused resources to another user who could take advantage of them.

A new order of complexity arises when one tries to eliminate the constraint of a small machine by introducing multiplexing with variable

allocations made on demand from pools of resources. In the most general form one has several processors and a variable number of jobs[5]. Each job has primary memory requirements which are possibly large, and in any case independent of the amount of primary memory actually available. A variety of new complications are introduced by this new set of conditions. For example, if there is more than one processor, it is necessary to interlock information such as supervisor data bases, and for this purpose it is convenient to have special hardware instructions. The variable number of jobs is often handled with software queues with ordinary list structuring. Since jobs are of variable length and are run for varying times, one must make proper use of an interrupting clock so that no job exceeds its own expectations. But this point is subtle since one must also ensure that excessive calculation time for one job does not cause another job to be completed unduly late. Thus, one must introduce the notion of scheduling and preemption. Finally, the problem of allowing the user a very large primary memory is commonly solved by means of paging hardware[6]. This hardware allows a job to be initially allocated any available noncontiguous blocks of primary memory. It also allows the job to be removed from primary memory and reintroduced elsewhere as it becomes necessary for scheduled switching from one job to another. With a minor addition to the hardware and appropriate supervisor procedures, the user job may even operate without all of its pages in primary memory with the expectation that some pages may not be needed before the next input-output wait or job switch occurs.

All of the above aspects of multiplexing, with variable allocations complicate the system but also enhance the richness of the capabilities of the virtual machine that the user job sees. In addition, an important property has been introduced: the system is capable of growth of processor and memory capacity without modification of user programs or awareness on the user's part. We consider a critical property of such a multiplexed system to be that the user be unconcerned with the intricacies of multiplexing. He should have the illusion of a private computer at his disposal.

An important thing to notice is that so far there has been nothing to distinguish an interactive time-sharing from a batch processing system. In fact, we do not believe there is any basic distinction to be made. To a first approximation, one can consider a person as a kind of erratic input-output device. But there are certain problems which are seriously aggravated by the introduction of interacting people. These problems are only latently present in a batch processing system and often go unrecognized. They arise from the need to stop a job unexpectedly and then restart it at a later time. It is obvious that interacting users may have this need since they frequently make mistakes, are called to telephones, remember other engagements in the midst of a calculation or are still busy working when the computer must be shut down. In a batch system, one sees the same effect with the sudden introduction in the midst of

a production calculation of a high priority job which must be completed within a fixed deadline. Such preemptive priorities can, of course, be handled easily if the preempted job is destroyed. Destruction, however, is wasteful, and moreover, jobs working with a stored data base may not be able to start over later if they have changed the data base. To ensure that these unexpected interruptions and restarts of programs do not cause difficulty, one must anticipate a requirement for what one may call "reversibility of binding".

The term binding is applied here to an operation which occurs at a variety of levels in a computer system: the choosing of a particular hardware and supervisor environment with which to implement a program construct. Thus a compiler binds a source language program to the instruction set of a particular computer and a specific set of library routines, paging hardware binds a program address to a particular absolute location, and when an instruction is in the midst of execution it is bound to a processor with a particular internal logic. A look at a computer system from the point of view of binding provides insight into the nature of several complex problems because the requirement of reversibility of binding comes up in many aspects of the system. The simplest case occurs when a processor is interrupted for a moment to attend some input-output device. The program that was running should not be so tightly bound to the processor that it must run to completion before interruption can safely occur without aborting the program. Typically, at least the instruction in progress

does have to run to completion; this requirement is sometimes overlooked in specification of elaborate multi-cycle instructions. In the example of paging already mentioned, a program which has been removed from the machine will be reloaded at a time when the primary memory allocations to other users are not the same as before and there may be new physical memory binding required. Similarly, if the system is shut down and a user with work in progress has his service temporarily disrupted, there is a need to allow his computation to continue when system service is restored. However, during this shutdown it may have been necessary to reload the secondary storage in a different physical arrangement. Changes to the system library or supervisor may also occur during the break. The possibility of such changes requires that when user programs are stopped, they be unbound from all dependence on the specific version of library programs or supervisor. Such unbinding and subsequent rebinding can succeed only if the overall functional properties of the hardcore supervisor and machine do not change in the interim. Of course, there is an intrinsic requirement of a compromise between the desire to have all programs continue to run no matter what happened while they were interrupted and the desire to be able to upgrade the machine or modify the supervisor, Good functional design emphasizing modularity can minimize this problem by reducing the chance that a desired change in a module requires a change in some interface specification to which a user may be bound.

The foregoing discussion of reversibility of binding illustrates how a single viewpoint can illuminate several complicated problems arising from the objective of multiplexing. This illustration completes our discussion of multiplexing and we now turn our attention to a second idea which is gaining rapidly in importance: sharing.

Sharing and Its Contribution to Complexity

An important opportunity arises in a time-sharing system in which different individuals operate programs simultaneously. This simultaneity allows them to interact with each other, to share and exchange information, and in general to operate collectively rather than individually[7]. This conscious sharing is in contrast to the hidden multiplexing which is also occurring. There are many examples in which sharing is desirable: members of a research group working together on a project; system programmers developing a new computer operating system; airline reservation clerks using a common data base; and students using a common compiler. It is our belief that sharing among users is going to become one of the more important properties of multi-access computers and that the difficult problems which accompany sharing will become the more critical in the system programming field.

For an example of the difficulties, all-or-nothing sharing, which is relatively easy to implement, is inadequate on grounds of privacy and the need for selective permission to access information. This

permission must be grantable both to individuals and to groups of individuals with common attributes, not all of whose members may be known at the time permission is given.

An often overlooked further problem of sharing is one of credibility: there is a need to have confidence in a computing system. One can identify a need for varying levels of confidence ranging from the presumably high confidence one has in the supervisor to the nearly total lack of confidence one has in his own untested program. This confidence varies through the layers of software in the system, from the official program library, through the local user community library and to a friend's private programs which may or may not be well done. Certification that a program works as expected, whatever its needed level of confidence, is a seriously difficult problem of systems which permit sharing. The current solution to this problem is based on judgment and experience with the programs in question, much as research articles are formally accepted by journals or drugs are used in the medical profession.

A further observation about sharing is that day-by-day changes which occur during any programming project introduce a strong incentive to avoid having multiple copies of programs and data. For example, when a programming mistake is found in a widely used library procedure such as the square root program, one needs to replace this program in such

a way that no future computations will use the erroneous copy. More significantly, since a large project cannot be done all at once, there is a need to be able to start writing and testing some programs with dummies and shortcuts replacing others. As different members of a programming group replace dummies and improve modules to remove shortcuts, each must have available other members' latest module versions for testing and integration, rather than copies which may be quickly outdated.

The impact of an evolving set of programs on sharing highlights the existence of a general problem of sharing: the minimization of undesired multiple copies of shared information. The reason for avoiding multiple copies is clear; updating or revision of one does not automatically imply updating of the other copies. Worse, if two people attempt to update different copies, catastrophe generally results when one or the other (but not both) replace the master. Throughout a system which has shared information, mechanisms for avoiding multiple copies abound. For example, a processor is conditioned to ignore interrupts (inhibited) at times when it is holding a copy of some system-wide datum in its registers, to guarantee that an interrupting routine does not look at the now obsolete copy in primary memory. Hardware instructions which permit read, alter, and rewrite of a memory cell in a single processor instruction are similarly provided to avoid the extra copy of the memory cell briefly required to implement the same sequence with conventional read and write instructions. The existence of a look-ahead scheme or an associative memory in a processor,

which allows it to quickly obtain contents of primary memory cells soon to be or recently addressed, must be taken into account as a potential multiple copy of information changed in primary memory by another processor. We will see below some implications of the desire to avoid multiple copies of a file of information when that file is placed in primary memory.

Of course, most of the problems which sharing introduces have always existed in batch processing and early interactive time-sharing systems where many users share a supervisor and program libraries. However, as one introduces an active on-line community which consciously shares and develops symbiotic relationships, it becomes essential to solve more generally some of the problems which we have discussed. We advocate as a sensible starting point the technique of segmentation proposed by Dennis[8], in which segments are the smallest separately shareable elements of information.

In the implementation of these ideas described by Daley and Dennis[9] segments are kept in an information storage system which gives the user the illusion that it always remembers perfectly. The critical properties of a segment in this implementation can be listed:

1. It is a logically contiguous block of arbitrary length centaining addressable data elements, typically the words of the hardware configuration.
2. It has a name and logical location supplied by the user.

3. It has associated with it a list of users and groups of users and for each, attributes with which they may enjoy access to the segment. These attributes, such as reading permitted, execution permitted, etc., apply dynamically to every reference made to the segment so that permission may be revoked at subsequent times for any particular user or group of users.
4. The segment has associated with it a unique identification, such as the date, time, place, and processor of creation, so as to be able to have a completely specific identification.

It will be noted that the above list of properties can be applied to a conventional file system using secondary storage only[10]. When one wishes to extend sharing of segments to the primary memory, so as to avoid multiple copies there too, then to maintain rapid access it is necessary to have special hardware. As noted before, if one does not share segments in primary memory, then many of the problems of temporal change become rampant; for example, when library subroutines are corrected of errors, not all occurrences are updated. In addition, lack of sharing in primary memory requires more primary memory for the redundant copies of programs simultaneously needed by different users. As a dividend, hardware to allow sharing of segments also permits the programming convenience of dynamically variable segment size in primary memory. When one develops a new algorithm, a new programming system or a similar application of a computer, there is

a desire to work out logical consequences without having to wrestle with storage management. Having available dynamically variable regions of program memory is of great assistance.

To illustrate more precisely how the sharing of a single copy of information in primary memory is accomplished, we examine a technique of implementing segmentation. This technique requires that the processor use for each word to be retrieved from primary memory, instead of the usual single component address, a two component address. The first component is interpreted as identifying the segment containing the word, the second as identifying the relative location of the word desired within the segment. Thus, instead of referring to a memory address as, say, 741, its position in a linear address space, the program would instead refer to the address (5,20) where the reference is to the 20th word in segment five. (Note that for implementation simplicity, segment names are mapped into numbers typically at the time of first usage in primary memory. As seen below, the number of a segment is used as an index to identify its properties in tables, and its value is assigned arbitrarily by the supervisor. In contrast, word numbers are usually assigned by compilers and denote position within a linear array.) The basic implementation of a segmented address space requires that the processor automatically construe the segment number (in our example, 5) to be an index into a table (the descriptor segment) which contains one entry per segment. This table is maintained by the supervisor exclusively for this user. Its location is loaded into a special processor register by the supervisor

and it contains for each segment both this user's access rights and a pointer to the actual physical location of the information in primary memory. Since a second user would have his own supervisor-maintained descriptor segment even if one of his entries points to the same physical information in memory, he can have distinct access rights. Further, there is no requirement that entries be made in the same order in all descriptor segments, so that each user may have a private address space with his own naming conventions, yet be using some segments in common with other users.

Segments thus offer a natural technique for solving the problem of shared procedures in primary memory. In addition, they allow a supervisor protection mechanism to be implemented within the hardware framework. Clearly, the descriptors which refer to supervisor segments can prevent a user from destroying or tampering with any of the procedures. Entering the supervisor merely becomes an attempted transfer to one of the supervisor procedures. One simple form of supevisor/user relationship is for the access bits to take on a modal interpretation when such an access is attempted. A description of a mechanism based on this approach, which allows concentric rings of protection, if found in a paper by Graham[12]. An alternative segment protection approach has been described by Evans and Leclerc[13].

By capitalizing on the flexibility of the descriptor segment, a supervisor may be composed of an arbitrary but self-consistent set of modules which are distinct for different users. Clearly, such different supervisors have to obey the same ground rules and follow common conventions

on data bases. But it is possible to have different supervisors for each person with the exception of a hardcore of the supervisor roughly embodying interrupt handling, primary memory multiplexing and the information storage system. Using this technique, system programmers may develop modifications of supervisor procedures and test them simultaneously with the operation of the old system.

Summary

In this paper we have isolated two distinct objectives of multiplexed computer system design. We have distinguished between those problems which are exclusively related to multiplexing and those which arise from conscious sharing of information between users. In both multiplexing and sharing we have seen how problems which are latent in noninteractive batch processing systems are aggravated seriously by people using the system simultaneously and in concert. Thus on one hand, interactive time-sharing systems develop a requirement for unpredictable interruptions of the use of computer resources and these interruptions in turn produce a need for reversible binding. On the other hand, because individual members of time-sharing communities are in a position to conveniently interact and share ideas and programs, the control and exploitation of sharing is a key idea in system design. Mechanisms such as segmentation allow solutions of many of the sharing problems that one encounters and also increase insight into the nature of the difficulties.

It is clear that the design of multiplexed systems has not reached its limit and it is also clear that it will not without further analysis and functional understanding of the principles underlying the mechanisms. Only this understanding will make it possible to find the necessary simplifications in mechanisms which will allow design of more sophisticated systems.

References

[2] Dijkstra, E. W., "Structure of 'THE'-Multiprogramming System," ACM Symposium on Operating Principles, Gatlinburg, Tennessee, October, 1967 (to be published in Communications of ACM, May, 1968).

[2] Mealy, G. H., et al., " The Functional Structure of OS/360," IBM Systems Journal, Vol. 5, No. 1, 1966, pp. 2-51.

[3] Comfort, W. T. , " A Computing System Design for User Service," AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 619-626.

[4] Corbató, F. J. and V. A. Vyssotsky, "Introduction and Overview of the Multics System," AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 185-196.

[5] Saltzer, J. H., " Traffic Control in a Multiplexed Computer System," Project MAC Technical Report MAC-TR-30 (Thesis), M.I.T., Cambridge, Massachusetts. July, 1966.

[6] Kilburn, T., " One-Level Storage System," IRE Transactions on Electronic Computers, Vol. EC-11, No. 2. April, 1962.

[7] Fano, R. M., " The Computer Utility and the Community," 1967 IEEE International Convention Record, Part 12, pp. 30-37.

[8] Dennis, J. B., " Segmentation and the Design of Multiprogrammed Computer Syatems," Journal of the ACM, Vol. 12, No. 4, October, 1965, pp. 589-602.

[9] Daley, R. C. and J. B. Dennis, " Virtual Memory, Processes and Sharing in Multics," ACM Symposium on Operating Principles, Gat1inburg, Tennessee, October, 1967 (to be published in Communications of ACM, May, 1968.)

[10] Crisman, P. A., ed., The Compatible Tune-Sharing System: A Programmer's Guide, 2nd ed., M.I.T. Press, Cambridge, Massachusetts, 1965.

[11] Glaser, E. L., et al., " System Design of a Computer for Time Sharing Application," AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 197-202.

[12] Graham, R. M., " Protection in an Information Processing Utility," ACM Symposium on Operating Principles, Gatlinburg, Tennessee, October, 1967 (to be published in Communications of ACM, May 1968).

[13] Evans, D. C. and J. Y. Leclerc, " Address Mapping and the Control of Access in an Interactive Computer," AFIPS Conference Proceedings, Vol. 30 (1967 SJCC). Thompson Books, Washington, D. C. 1967, pp. 23-30.