MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Memorandum MAC-M-181
October 1, 1964

# THE STRUCTURE OF ON-LINE INFORMATION PROCESSING SYSTEMS*

Jack B. Dennis

and

Edward L. Glaser

ABSTRACT

Many concepts regarding the organization of an information processing utility are pertinent to the design of on-line information systems in general. Two areas of particular importance form the subject of the present article. One concerns formalizing the structure of programs so that interrelations between processes operating in a system may be better understood. The other concerns the organization of system hardware so that computation resources may be effectively shared among many concurrent tasks.

## 1. INTRODUCTION

In this paper we outline some concepts of computer system organization that have arisen from the evolution of an information processing utility as envisioned by Project MAC at M.I.T. We hope to show how these concepts are of significance to the design of on-line real-time information systems in general.

Project MAC was established in May 1963 to bring together a variety of fairly advanced work in the on-line use of computation--work ranging from development of problem oriented languages for on-line applications to the system design of experimental multi-access computer systems. The overall objective of Project MAC is the evolution of the concept of an information processing utility, including the development of hardware organizations, philosophies of executive control and programming languages. The immediate object of research is an evolving multi-access computer system developed at the M.I.T. Computation Center.[1]

An information processing utility consists of processing units and directly addressable main memory in which procedure information is interpreted as sequences of operations on data, a system of terminal devices through which users may communicate with procedures operating for them, and mass memory where information is held when not required for immediate reference. One fundamental attraction of the MAC concept is the increased productivity of "computer catalyzed research"* that results from close man-machine interaction. Another attraction is the wealth of data and procedures that can be made accessible to a large user community through the file memory of an information processing utility.

As our concepts have crystallized, certain system attributes have impressed us with their preeminent importance for the success of a large scale system. These attributes also seem pertinent to on-line computer systems in general.

The ability to evolve is one of these system qualities. No large system is a static entity--it must be capable of expansion of capacity and alteration of function to meet new and unforseen requirements. The ability of a system to adapt to new hardware, improved procedures and new functions without interfering with normal system operation is mandatory.

*With apologies to E.E. David.

*Continuous availability* of a system to perform its function is of paramount importance in an on-line system.

*Organizational generality* is an attribute of underrated importance. The correct functioning of on-line systems imposes requirements that have been met *ad hoc* by current designs. Future system designs must acknowledge the basic nature of the problems and provide general approaches to their resolution.

In general, an on-line computation system is inherently multi-functional--many tasks are being processed concurrently in response to changes in the environment. This contrasts with batch processing where system resources are applied to one task at a time and individual tasks are run to completion. The multifunction nature of on-line computation creates urgent demand for resolution of issues that have lain dormant in conventional computer installations, but have become crucial to the success of future large-scale systems. We examine these issues in the remainder of this paper. Section II deals with the naming, addressing, allocation and protection of information in main memory, and the powers of processes that reference this information. A general model of program structure is presented based on the concepts of segmentation, spheres of protection, and computation processes. The model forms a rational basis for describing the interaction of multiple tasks present in on-line information systems, and provides a framework for attaining the attributes set forth above. Section III relates these attributes to the physical structure of a computer system, stressing the importance of modular systems assembled from a small set of basic components.

## II. CONCEPTS OF PROGRAM STRUCTURE

### Referencing Information and Allocation of Main Memory

A number of difficult programming problems facing the system designer relate to the naming and referencing of information:

(1) Variable size data objects - In on-line computation realistic bounds on the size of data structures are difficult, if not impossible to specify at the initiation of a process. In a conventional memory system the set of addresses serves two functions--it permits the selection of specific words in the physical main memory, and it is the name space from which the identifiers of all information referenced by a process must be chosen. The assigmment of variable size data structures to a conventional memory is a formidable problem because the space of physical locations for information and the space for names of information coincide. If too many contiguous memory locations are assigned, the unused physical locations are unavailable for other uses because of a possible future conflict in the use of addresses as identifiers. If too few locations are assigned to a data structure, difficulty arises when additional locations are needed. Not only must physical memory be reassigned, but the addresses used to refer to them must be reassigned in name space.

(2) Common procedures - In a multiprogrammed computer system the use of common procedures has offered an attraction in improving system efficiciency where it is likely that several system functions will invoke the same procedure. The principle of expressing algorithms in "pure procedure" form so that concurrent execution by several processes is possible is already widely appreciated. However, the problem of assigning locations in name space for procedures that may be referenced by several system functions and may perhaps share references to other procedures, is not widely recognized and leads to severe complications when implementation is attempted in the context of conventional memory addressing.

(3) Common data - The need for two or more processes to reference a common data structure is a frequent occurrence in on-line systems. This problem is similar to the problem of common procedures with the added complication that one or more processes may need to change the contents of the data structure. A mechanism for locking out references by some processes during modification of a data structure by a particular process is needed.

(4) Dynamic allocation - In on-line information systems provision for dynamic allocation of main memory among the processes becomes especially acute. The use of relocation registers coupled with periodic compaction of storage, is only a partial solution since moving information between physical memory locations disrupts the progress of processes in operation, and would constitute an intolerable burden in systems of increasing complexity.

(5) Page turning - The concept of a page turning memory--a scheme by which only pertinent blocks of information are kept in main memory while nonpertinent blocks are deleted through a suitable algorithm--is very useful for certain classes of procedures and data structures, but may be disastrous if applied to other classes of information. Schemes that have been discussed to date do not allow for flexibility of application as they are not tied to a philosophy of program structure, that is, the selection of information to be deleted is independent of its function in the system.

## Segment Structure of Programs

The most significant concept in our thinking about program structure is to view a program as concerning information that is grouped into a collection of objects called segments after Holt[2]. A segment is an ordered set of words. It is referenced by a segment name that distinguishes the segment from all others and an integer address that selects a particular member from the ordered set of words. At any time a segment contains a definite number of words, the length of the segment. However, the length may vary arbitrarily in the course of a computation process.

Each segment referenced in the course of a particular computation process has an associated class which determines what forms of reference to it are valid for that process. For the present discussion, three classes will be identified:

| | | |
|---|---|---|
| (a) | Procedure | The segment contains machine instructions that describe a computation process. |
| (b) | Data | The segment contains data that are referenced and possibly modified by a computation process. |
| (c) | Read only data | The segment contains data that may be referenced by a computation process but not modified |

We assume here that procedure segments are in pure procedure form.

A segment being actively referenced as a procedure by a processing unit of the computer system is said to be in execution. We will use the term process to denote the act of executing a single sequence of instructions taken from a succession of procedure segments. In a multiprocessor computer system, a number of processes may be in execution simultaneously.

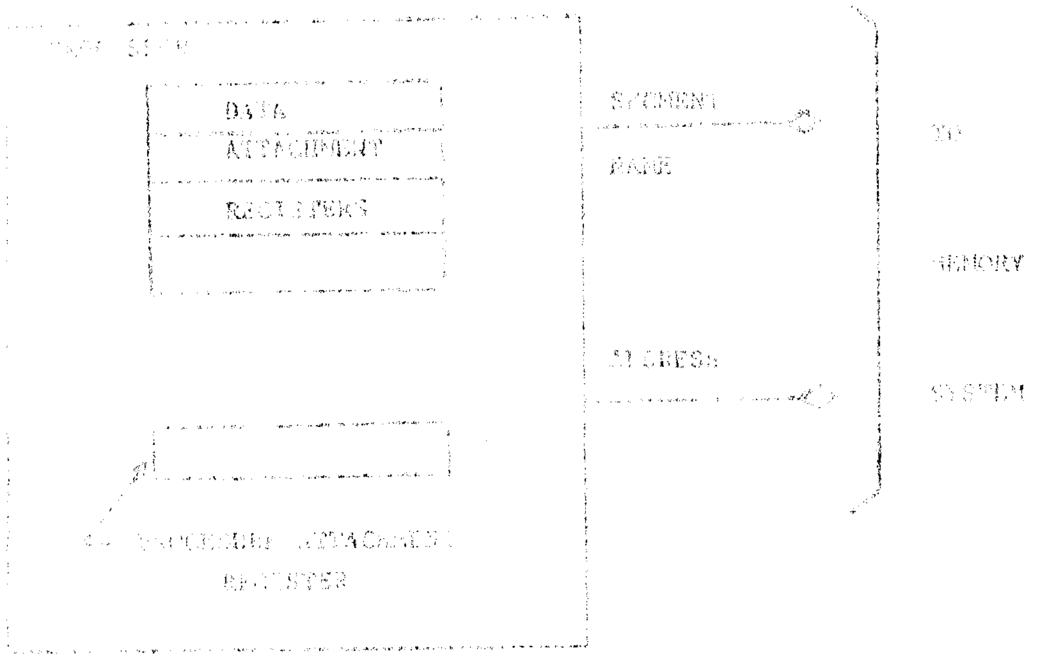
At any instant a process is referencing exactly one segment as procedure and one or more segments as data. The pure procedure convention for procedure segments permits several processes to reference the same procedure segment concurrently without interference.

Implementation of Segmentation

The implementation of segmentation requires that each memory reference generated by a process imply the name of a segment and an address within the segment. A scheme for accomplishing this is illustrated in Figure 1a. This particular scheme is chosen to illustrate and clarify the concepts that have been introduced. In practice, however, various compromises necessary in a particular system design could lead to a quite different implementation. Several special registers called attachment registers are included in each processing unit. The data attachment registers can be loaded with segment names by special instructions available to all processes. The typical single address instruction code format is expanded slightly asshown in Figure 1b, to include a field that selects a data attachment register. This attachment register contains the segment name pertinent to the data reference of the instruction. Procedure references by a processing unit are made to the segment named in the procedure attachment register. The procedure attachment register is automatically loaded from one of the data attachment registers when a transfer of control or a subroutine entry instruction is executed.

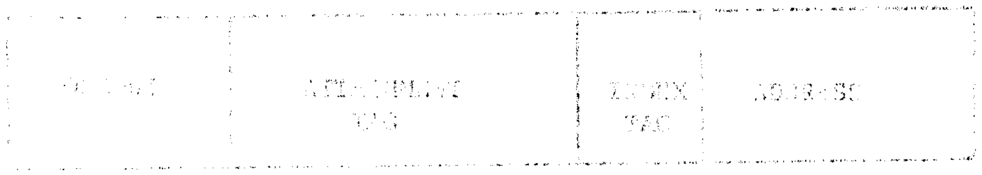
The memory references in terms of segment name and address are effected by dividing main memory into equal-sized blocks and using a page index as shown in Figure 2. Each segment is made up of an integral number of block sized pages which may be arbitrarily distributed throughout main memory. Therefore an address within a segment is broken into the concatenation of a page number and a line address within the page. Each entry in the page index memory contains an effective segment name, a page number and a block number. The block number gives the block in main memory where the indicated page of the named segment is to be found. When the processing unit makes a reference to main memory, it supplies to the page index the name of a segment from one of its attachment registers, and the effective address within the segment generated by normal indexing techniques. The effective address is split into page number and line,

Figure 1 Generation of Memory References



CENTRAL PROCESSOR REGISTERS

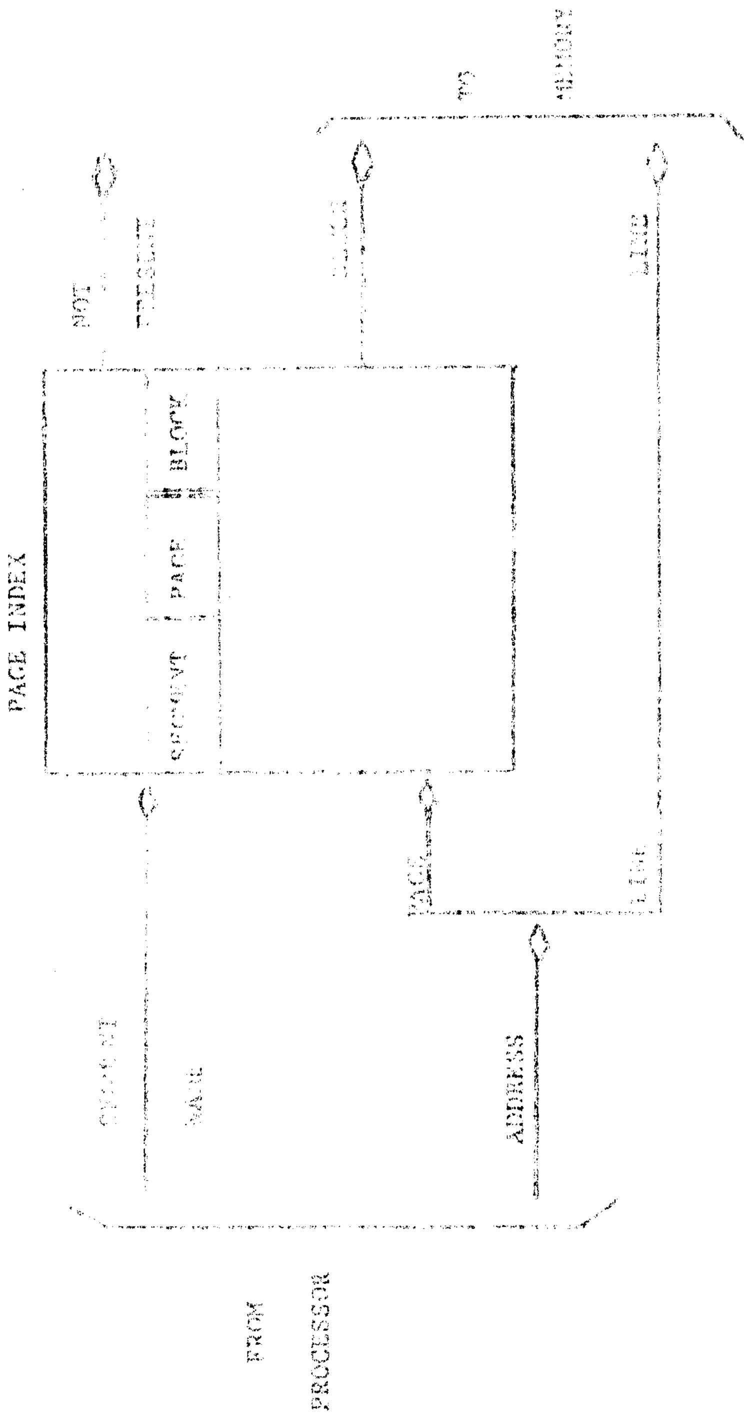
(a)



[illegible]

[illegible]

Figure 2 The Page Index



and the segment name and page number are used in an associative look-up in the page index to find the block number to be used for accessing main memory. If no match is obtained in the page index, the reference was to an address outside the current bounds of the segment and a fault condition is signaled.

Comments Regarding Segmentation

A segmentation scheme offers an attractive solution to the problems of naming and allocating information for reference by processes performing diverse functions in an on-line system. Dynamic storage allocation is achieved through changing the associations of blocks of physical main memory with pages of segments. Common reference to a data segment by several processes is possible simply through the use of the same segment name by both processes. It is natural to implement data lockout on a segment basis. Common execution of a procedure segment by two or more processes is possible through the communication of data segment names as parameters. This points out the necessity of not embedding segment names in procedures, but considering them as a system resource that is applied according to need. A variable sized data structure is handled by making it a distinct segment and allocating more or fewer blocks to it as the structure changes in size.

The "page turning" or "look -behind" concepts can be employed at three levels in connection with segmentation. First, page turning may be employed within any segment so that only pertinent pages are kept in main memory. Second, the look-behind principle can be applied to the problem of deleting entire segments from core memory so new information may be brought in. Finally, a look-behind technique can be used to avoid use of the page index except when reference is made to a new block of a segment.

Input/Output Functions

Fitting input/output operations into the model of program structure presented so far requires the development of a general treatment of communication between peripheral units and processes in a computer system. Preliminary consideration has led to the following characterization: A computer system includes a number of peripheral units that in general perform a greater number of input/output functions, each identified by a distinct I/O function name. These input/output functions have the property that their execution frequently requires a time that is very large compared with that required by other computation steps.

We suppose that each input/output function is at any time associated with a specific procedure step that calls for performance of the I/O function by its function name. When a process encounters an I/O procedure step, it ceases execution and enters I/O pause status. The process resumes execution once the function requested has been completed.

Using a multiple arm disk storage unit as an example, one I/O function and its corresponding routine could be associated with controlling seek actions for each arm. Another function would be the transfer of data to or from a specific segment in memory. Finally, execution of a "status function" could deliver status conditions pertaining to the disk peripheral as a whole.

## Creation and Termination of Processes

Parallelism is an inherent part of the description of many on-line computation procedures. It is the essence of multiprogramming, and is necessary for controlling possibly simultaneous input/output functions. The program fork is the fundamental mechanism by which new processes are created. A process that has completed its function terminates through a quit procedure step. In conventional machine organizations, parallel execution of input/output functions is provided through an interrupt system that causes some specific processing unit to switch processes upon each function complete signal. Conceptually, however, it is not clear which of several on-going processes should be interrupted. On the contrary, the completion of an input/output function merely makes a new process available for execution with an associated priority. These thoughts lead to the concept of a queue of processes awaiting execution by processing units. The management of the queue would be according to the following principles.

(1) Processes are entered into the queue by program forks and the completion of I/O functions.

(2) A process is taken from the queue and placed in execution whenever a processing unit becomes free.

(3) A processing unit becomes free when it encounters a quit or I/O function procedure step, or the process has exceeded its allotted time.

## Protection of Processes

In present-day on-line systems, protection among processes performing different tasks is either absent or confined to one level of object processes that run under the egis of a master control program. In consequence, the evolution of the system in terms of new equipment or new functions (tasks) is painful: All hardware and software must be debugged before a system can be made operational. Reprogramming cannot be accomplished on the operational facility; thus duplicate equipment is required for development effort. Furthermore, new functions cannot be tested in a realistic environment even with duplicate equipment, before being entered in the operational system. Since there is no absolute guarantee that a program is ever completely error free, there is always the possibility that a program malfunction will induce a catastrophic system failure when adequate protection mechanisms are not provided. In a complex evolving on-line system such possibilities may well become probabilities or certainties. Protection mechanisms, by limiting the scope of possible damage resulting from an error, can greatly lessen the chance of disastrous failure. If the performance of input/output functions requires specialized coding in the master control program of a system, then altering the set of peripherals or changing in/out functions requires modifications of the master control programs, leading to the same problem of coping with evolution that has just been discussed.

In the context of an information processing utility as viewed by Project MAC, protection is a necessity for several additional reasons. A number of users are testing procedures under development that are not free from errors. Program malfunctions induced by such errors must not interfere with correct execution of tasks proceeding concurrently for other users. Also, the qualities of privacy and security of information belonging to the various users and user groups is essential. A user must only be permitted access to procedures and data owned by him, and other information as authorized by its owner.

## Spheres of Protection

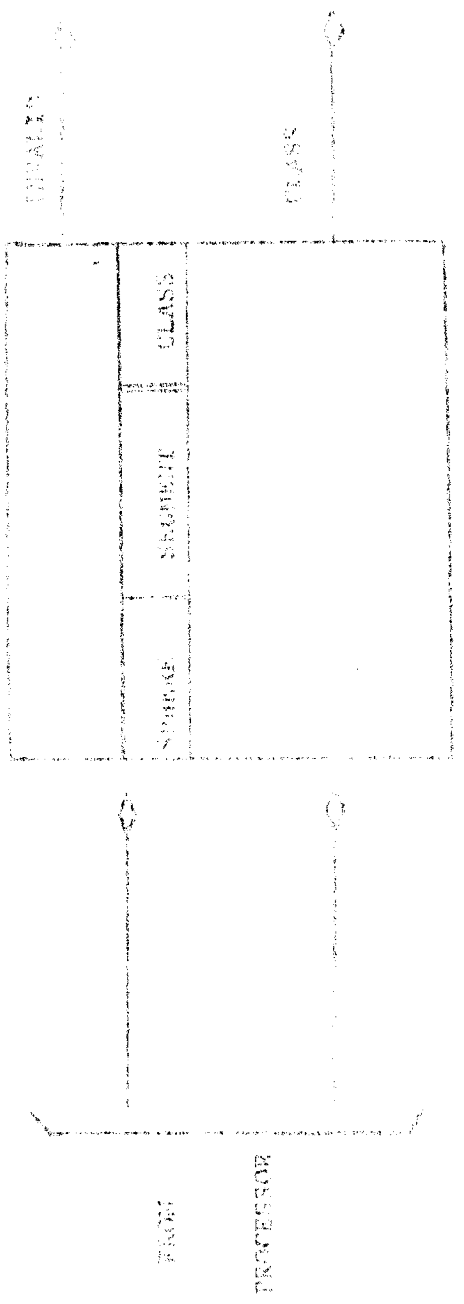
The segmented structure of procedure information and data discussed in the preceding paragraph provides a natural basis for a concept of protection.

We think of each process as operating with a sphere of protection* containing the name of each segment that may be legally accessed by the process, and the class of reference permitted. The sphere of protection also contains the names of I/O functions assigned to processes operating in it. References by a process to segments or I/O functions not within its sphere of protection are illegal, as are references of incorrect class to valid segments. Implementation of the sphere of protection concept with respect to segments occupying main memory can be accomplished through the use of a segment index as shown in Figure 3. Each entry of the segment index contains a sphere name-segment name pair and a class code that indicates the nature of reference to the segment that are legal within the associated sphere of protection. Each processing unit is equipped with an additional special register that contains the sphere of protection name for the process being executed. Whenever the process attempts to load an attachment register with a new effective segment name, the segment name and the sphere of protection are presented to the segment index. This pair is associatively matched against the corresponding fields in the segment index. If a match is found, the new segment name is legal; the class code is placed in a class indicator associated with the attachment register, and execution of the process is continued. If no match is found, reference to the segment is not valid in the current sphere of protection. This is a fault condition that terminates the process. From the foregoing, it is evident that the attachment registers will only contain segment names to which valid references may be made within the established sphere of protection, with an indication of the class of reference permitted.

*After E. Van Horn

Since a segment may be in several spheres of protection at once, we have a basic framework for allowing independent tasks to share access to procedures and data as appropriate. Note that within this scheme many levels of common access are possible: Two tasks may share certain procedures and data segments for which access is denied to other processes. Each of these tasks may reference private information and public routines as well.

Figure 3 The Segment Index



## Relations Between Spheres of Protection

It is useful to think of a sphere of protection B as having been established through the action of a process operating in a distinct sphere of protection A. In this connection, we shall refer to A as the superior of B, and B as an inferior of A. We suppose there is exactly one sphere of protection that has no superior and we call this the master sphere. Before discussing the reasons for establishing this hierarchy of spheres of protection we must mention the powers possessed by a process in connection with the hierarchy. A process operating in a sphere of protection B that is inferior to sphere A may

(a) create and release segments.

(b) create an inferior sphere of protection C.

(c) enter a segment valid in sphere B as valid in C.

(d) initiate a processs in C.

(e) halt all processes in C and its subhierarchy of inferior spheres.

(f) destroy sphere C and its subhierarchy.

The relation between spheres of protection in the hierarchy would not be complete without mention of exceptional conditions. A procedure step encountered by a process that is meaningless in its sphere of protection yields an exceptional condition. Examples are a reference to an invalid segment, an invalid class of reference to a valid segment, reference to a non-existent address within a segment, or an undefined operation code. We postulate that an exceptional condition arising from a process in sphere B terminates that process and initiates a specific process in the superior sphere of protection, sphere A.

## Illustrations

Program development is an essential function in an evolving on-line system and an illustration of the value of a hierarchy of spheres of protection. A person creates procedures to perform new tasks by communicating with a programming language system. Suppose the processes performed by the programming system on behalf of the programmer are carried out in a sphere of protection B. These processes create a number of segments which are referenced as data in sphere B and constitute the coding of the user's procedure. To perform the user's procedure, the programming system running in sphere B creates an inferior sphere of protection C in which the segments of the user's program appear as procedure or data, according to declaration made to the programming system, and then initiates a process in sphere C. Exceptional conditions arising in sphere C terminate the process and reestablish a process in sphere B. Exceptional conditions should not occur in the execution of the language system procedures in sphere B as they are presumably debugged programs. If one does occur, a process is created in the sphere A that is superior to B.

The reasons for placing C inferior to B rather than directly under A are several. First, it is natural that the programming system in B should have the power of creating, deleting, and allocating resource to sphere C. Second, the programming system in B is aware of the interpretation to be made for exceptional conditions arising in the programming system itself require action by a higher system. Clearly, further levels may be needed--for example, a person may be debugging a programming language system.

Another illustration of the use of hierarchy of spheres is a teaching program that runs under a monitor and interacts with many students whose files must be held confidential. A related example is a group of students working on a large design project with the aid of on-line information processing. Each student is assigned the design of a component, working with partly private and partly common information. The instructor monitors performance and alters specification through processes operating in a superior sphere. An analogous situation in military command and control is the training of new personnel in the operation of a specific system under the guidance of an instructor.

The Master Sphere

So that an on-line system has a high availability to perform its function, we require that the effect of a malfunction (due to either a programming error or a transient hardware fault) of a process operating in a sphere of protection be confined to itself and processes operating in inferior spheres. Thus it is necessary that modification of data describing the current allocation of main memory, I/O functions and processing units be disallowed for any process except one operating in the master sphere of protection. Otherwise, some process operating in an inferior sphere could, through a malfunction, leave the allocation data in an unauthorized or unmeaningful state. The resulting interference with processes operating in other pshères inferior to the master would be in violation of our premises. Thus, processes executed in the master sphere must perform all acts of allocating resources and scheduling processes. On the other hand, the more functions assigned to the master sphere, the greater the chance of catastrophic failure, since master processes may be critical to system operation. Consequently, it is envisioned that processes in the master sphere serve only the following functions:

(1) Maintain allocation tables for main memory and I/O functions and prevent conflicts in their assignment.

(2) Maintain queues of processes available for execution and waiting for input/output events.

(3) Take appropriate action upon exceptional conditions arising in immediately inferior spheres.

(4) Establish and delete spheres of protection inferior to itself in response to commands given by staff personnel through a suitable private terminal.

Inferior to the master sphere, several executive systems could exist, each within its own sphere of protection. Each system would <u>authorize</u> allocation of system resources to spheres inferior to itself, and execute allocation and scheduling acts by communicating with the master sphere. With this arrangement, one or more of the executive systems could be in operation while another was being debugged or modified.

Carrying these thoughts a step further, it is attractive to arrange the supervisory functions of an on-line system into modules of procedure operating in separate spheres of protection. On-line debugging of supervisory modules would then be possible in parallel with normal system operation, and the partitions between modules would be a safeguard and tool of diagnosis in the event of a malfunction.

## Communication Between Spheres of Protection

There are several situations in on-line systems where a function must be accessed by many processes operating in different spheres of protection. One such instance arises when one peripheral unit is a common system resource for many tasks--a disk file, for instance. In the private use of common procedure segments it is sufficient protection to limit processes to procedure (instruction fetch) references to the common segment. However, this is not the case where the procedure segment operates on a common data segment since manipulation of the data segment would have to be valid in the sphere of protection of all calling pr ses.

A disk file control program is such a procedure since it must operate on common tables describing the allocation of storage space on the disk units and the nature of its contents. Thus, it is attractive to arrange the programming so that the disk control processes operate in their own sphere of protection. We call such a program a protected service routine. To implement a subroutine call to a protected service routine, it is necessary to establish a mechanism whereby a process operating in one sphere of protection can initiate a process in an authorized other sphere, and supply it with the parameter values necessary for performing its function. It is necessary to limit the entry point permitted to the calling program because entering a procedure at an unexpected step is likely to cause disorganized modification of its common data and eventual system malfunction. It must be possible for a process invoking a protected common service routine to authorize the called procedure to reference a specific segment. This can be accomplished by passing the name of the segment as a parameter of the call. The special nature of the call causes the segment to be entered into the service routine's sphere of protection. (The service routine must release the segment upon completion of the requested function).

Other functions that could profitably be implemented as protected service routines come to mind. One is a set of control routines for handling message traffic through a multi-line communications terminal where the messages must be transferred to and from many processes performing unrelated tasks. The use of separate protected service routines for these functions makes it possible to adapt or add such functions in an operational system as requirements change or new peripheral units are added. In addition, there is the additional safeguard against the malfunction of a service function routine resulting in a catastrophic failure.

## III. CONCEPTS OF SYSTEM ORGANIZATION

Achieving the qualities of availability, organizational, generality, and ability to evolve in an on-line system has important implications with respect to the gross physical organization of the machine. These interrelations are taken up in the following paragraphs.

### Allocation of Computation Resources

In an on-line computer system there are, in general, several or many functions sharing the computation resources of the system through the mechanism of multiprogramming. The resources available for allocation among system tasks include addressable main memory, processing capability, secondary memory, and peripheral units, and are fixed in their total amount for a particular configuration of equipment at a particular time.

The demand of any function for resources can vary widely according to the demand for the services of this function, and according to the complexity of processing required to provide the service. Two general problems arise. How should an on-line system be structured to facilitate the efficient sharing of resources among its several functions? How is it possible to to select an equipment configuration that achieves a balance between the resources available and the resources required to perform the services, and yet retains the flexibility to expand without the need for a radical change or reorganization of hardware?

In the context of a typical batch processing computer installation where a single task at a time has access to system resources, a true balance of resources is difficult to attain. The configuration of the installation is tuned to the requirements of certain large jobs that constitute a major portion of the load. In consequence, other jobs are made to fit within this configuration, possibly resulting in compromised performance, and very likely wasting a large fraction of the system's resources that are of no value for these tasks.

In the context of a system in which a number of functions are served concurrently, resources not being employed in the execution of one task are available for allocation to other tasks. The system requirements for each variety of computation resource (main memory, processing, files and peripherals) are related to the sum of the different functions and is therefore better defined under average operating circumstances by the statistics of sums.

Within a framework where dynamic allocation is meaningful, the view of the user or the designer of procedures with respect to computation resources becomes quite different. No longer is it necessary for him to be concerned about applying the full resources of a system to his job--resources not needed by him may be allocated to other tasks. On the other hand, so that the system as a whole operates smoothly, it is essential that processes return resources as they become unneeded so they may be reallocated to other tasks. This is an important distinction, for in batch processing each task has a well-defined end point at which all resources ever allocated to it are released. In a multifunction on-line system, most tasks have no well-defined end point.

In an information processing utility, the allocation of resources is especially dynamic. Users call for performance of tasks that invoke a wide range of resources and make their request at unpredictable points in time. A scheduling and allocation policy must be adopted that applies resources to the tasks in a way that is consistent, equitable, and provides quick response for simple requests. This is accomplished through the assignment of each process to a priority level dependent on the amount of system resources required for its operation, and allotting quanta of processing time to tasks in each level in turn as long as higher levels are empty[3]-- the principle of time sharing. In the implementation of such a policy, it is inherently necessary to assign resources to a process not just once, but many times for the successive quanta granted to the process. This is in contrast with a telephone exchange where a resource, once released from a particular process will not be assigned to the same process again. It should be pointed out that a process may not use all of a quantum assigned to it as it may be suspended by execution of an I/O function. In this circumstance, the process loses the remainder of its time quantum which becomes available for other tasks.

Modular System Structure

The problem of matching system requirements for computation resources with specific equipment suggests a modular approach to system configuration. Each variety of computation resource is supplied in the form of a component (processing unit, memory module, file unit, communications unit). A system is formed from these building blocks by including as many of each component as required to meet the total demand of the system application for that variety of computation resource. If the system is planned with proper attention to generality, the modular approach makes possible a continuous balance between available and required resources in an evolving system.

To attain the necessary generality, the system designer must insure that all system modules representing the same resource are equivalent and equally applicable to any task performed by the system. Many designs have exhibited a tendency toward unnecessary specialization of the tasks performed by certain modules of a particular resource, for example, the presumption in a system design that a special processing unit will perform the function of organizing all I/O activity. Such specialization is essentially a built-in allocation of resources and limits the flexibility and reliability of the system. Whenever a resource is committed to a specific function, one is either wasting the resource because it is not being fully utilized, or one is in danger of not being able to meet peak demand for the function. System availability also suffers where specialization of function exists, since fewer substitutes are present for a specialized component that has failed.

The quality of organizational generality requires a symmetrical configuration of system components. For example, equivalent communication paths should exist between each processing unit and each main memory module. Furthermore, any processor should be able to directly request any I/O function. In this respect, a computer system should be modeled after modern telephone exchanges where pools of inherently equivalent trunks, receiving registers, markers, etc., are available for application to the separate requests for service. The telephone exchange analog is also an existence proof of a system capable of orderly expansion of capacity while maintaining efficient operation with a balance of resources.

## Multiprocessing

The view of system modularity expressed above, places the concept of multiprocessing in its proper framework. The key technique is multiprogramming--the application of a computer system to many concurrent processes through a rapid scheduling of resources. The multiprocessor idea is merely a means of achieving processing capacity in balance with other resources in a large computer system--the existence of more than one processor makes it possible to execute a queue of process at a faster rate. The presence of multiple processing units in a system yields the additional benefit that a processing unit may be assigned entirely to a particular task if a real-time requirement so demands. Also, parallel execution of a procedure by two or more processing units can make more effective use of procedure information in main memory.

## Distributed Executive

In a discussion of resource allocation in a multiprocessor computer system, it must be realized that, in a sense, programs themselves become an allocatable resource. The executive program of a more conventional computing system is written as a single entity perpetually occupying a portion of main memory. Coding the executive as a pure procedure allows it to be executed simultaneously by several processing units. The various tasks in progress call on various portions of the executive program whenever supervisory functions are required. As the number of tasks in progress increases through system expansion, via multiprocessing, the number of calls for supervisory functions likewise increases, and the heavily used portions of the executive program may become major system bottlenecks. On the other hand, certain other executive functions are only rarely required, and keeping these procedures in main memory is a waste. The application of the segmentation notions to the majority of the executive program as suggested in Section II of this paper can alleviate the latter problem. The bottleneck brought about by the possibility of several processors colliding in the execution of a common executive procedure can be resolved by replicating the procedure among several memory modules to eliminate conflicts.

With several copies of a system procedure in main memory, there is a range of mechanism from simple to complex for distributing calls between the copies. In the simplest arrangement, each processor would be assigned a specific copy to use. Alternatively a common procedure of a few instructions could process the calls for a systems procedure and distribute them among the copies. Since the need for a system function will vary dynamically just as the demand for other resources, the system design would have to include means for automatically providing copies of procedure, and for deleting them later, according to fluctuating demand

## System Availability

A most important characteristic of any on-line real-time system is its availability. By availability we mean the ability of the system to perform its overall function in the presence of the possibility of component failures. Availability does not imply reliability in the sense of guaranteeing the accuracy of individual results, where this is accomplished by a compromise leading to a significant fraction of shutdown time for the system as a whole.

The use of modular configuration of components and the distributed executive principle just described insures there are multiple components of each system resource. The failure of any single component need not produce failure of the system as a whole. Since processes are independent of the particular processor or memory module they use, or the particular copy of a procedure they may call, the process of reconstruction after a breakdown is greatly simplified. If there are more than two components of each system resource, we see that a highly reliable system is possible with less reliable components, without the necessity of complete duplication of the computation resources required by the systems's mission.

Acknowledgement

The material in this paper evolved directly from th authors' concern with the organization of computing hardware suitable for furthering the objectives of Project MAC. The contributions of A. W. Holt toward formalizing the structure of programs have been an inspiration to our work. The development of these ideas to their present state [4] has benefited from discussion with many of our colleagues and with visiting scientists from other institutions, both in the United States and abroad. Special mention must be made of Earl Van Horn whose influence has been substantial.

## REFERENCES

[1]M.I.T. Computation Center, The Compatible Time-Sharing System: A Programmers Guide (M.I.T. Press, Cambridge, 1963).

[2]Holt, A. W., "Program Organization and Record Keeping for Dynamic Storage Allocation," Information Processing 62 (North Holland Publishing Co., Amsterdam, 1962), p. 539.

[3]Corbato, F.J., Merwin-Daggett, M. and Daley, R.C., "An Experimental Time-Sharing System," AFIPS Conference Proceedings (The National Press, Palo Alto, Calif., 1962), Vol. 21, P. 335

[4]Dennis, J. B., Program Structure in a Multi-Access Computer, Project MAC Technical Report, MAC TR-11 (M.I.T., Cambridge, Mass., 1964).