Segmentation and the Design of Multiprogrammed Computer Systems

JACK B. DENNIS

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. Problems that must be solved by any scheme for multiprogramming include: (1) dynamic allocation of information to a hierarchy of memory devices, (2) means for programs to reference procedures and data in a manner that is independent of their location in physical memory, (3) provision for the use of common procedure and data information by many programs, (4) protection of system resources from unauthorized access, and (5) rapid switching of computation resources from one program to another.

The concept of *name space*, the set of addresses a process can generate, is contrasted with the *memory space*, the set of physical memory *locations*, and memory referencing schemes are described by *address mappings* from name space into memory space. In this context, the inadequacies of several approaches for solving the problems of multiprogramming become evident. The segmentation of procedures and data forms a model of program structure that is the basis of an address mapping function which will be a valuable feature of future computer systems.

1. *Introduction*

In an increasing number of computer system designs, the hardware and software are planned so that the computation resources of the system (main memory, processing units, auxiliary storage, input/output channels) are shared by many concurrent computations. The set of techniques for accomplishing this mode of computer system operation is called *multiprogramming.*

The planners of a multiprogrammed computer system must recognize that computations differ greatly in their demand for computation resources: some require only a small area of main memory for efficient execution while others require vast amounts; some computations pause frequently for input/output events while others run continuously for long periods. Moreover, the demand of a particular computation for system resources will in general vary greatly during its course of operation.

A multiprogrammed computer system must be designed so that the available computation resources are applied to the computations in progress in a manner such that these resources are most effectively utilized.[1] To maintain a balance between demand and supply and to ensure a fast response of the system to changing external conditions, it is essential that the system design include features that allow physical resources to be rapidly reallocated among the computations being multiprogrammed. The present paper is concerned with a study of the problem of making effective reference to data and procedure information which will reside in

Presented at the IEEE International Convention, New York, N. Y., March 1965.

Work reported in this paper was supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract number Nonr-4102(01).

[1] Balancing supply and demand of computation resources may call for the use of multiple processor units in a computer system design. The use of a *multiprocessor* design has further potential advantages in reliability and flexibility. However, the techniques of applying such systems to multiple computations fall in the domain of multiprogramming.

different physical memory locations during the course of a computation as a result of reallocation activity.

2. *Preliminary Considerations*

Main memory allocation. For multiprogramming to maintain a reasonably high utilization of the processors, main memory and input/output channels of a computer system, the data and procedure information relevant to the current state of several computations must occupy main memory at any instant. Since main memory not used by one computation is available to the other computations sharing main memory, it is advantageous for a computation to relinquish space in main memory that contains information no longer relevant. From the viewpoint of the system, one may think of auxiliary storage as containing the totality of information required for the complete execution of all computations; it is the task of the system to maintain in main memory a portion of the totality that is relevant to some subset of the active computations.

Name/location map. If the allocation problem for main memory is to be at all tractable, the system must not be required to place a data or procedure object in the same set of locations of main memory each time it is to be referenced by a computation. Under this assumption there must be information in the computer system which relates the addresses used by a computation to the physical locations containing the desired words and to a mechanism that applies this information in the execution of memory references. We call this information and associated mechanism the *name/location map* of a multiprogrammed computer system. The nature of the name/location map and the associated addressing mechanism form the major subjects of our discourse.

Evolution. Large computer systems of the future must be planned so they may evolve in an orderly manner to meet unforeseen requirements and take advantage of new technologies in hardware and software. Thus, the programming of any large system is continually undergoing revision as new system functions become necessary, as better procedures are designed, and as new equipment is added. Several large computer systems are operating or being planned, in which program development is done online and concurrently by many users [1, 2]. It is clear that protection mechanisms are an essential part of the design of such systems. Programs very likely to contain errors must be run but must not be permitted to interfere with the accurate execution of other concurrent computations. Moreover, it is an extremely difficult task to determine when a program is completely free of errors. Thus, in a large operational computer system in which evolution of function is required, it is unlikely that the large amount of programming involved is at any time completely free from errors, and the possibility of system collapse through software failure is perpetually present. It is becoming clear that protection mechanisms are essential to any multiprogrammed computer system to reduce the chance of such failure producing catastrophic shutdown of a system.

Common procedures and data. The ability of a computation to reference data and procedures in common with other computation processes is recognized as an important feature of multiprogrammed computer systems. The use of common data is essential where two or more computations are performing related functions and must communicate with each other. The use of common copies of programs by several computations (which requires that these programs be written in read-only or "pure procedure" form) allows better utilization of memory, and reduces the record-keeping that would otherwise be required.

The state word of a computation. By the *state word* of a computation we mean the information that must be placed in a processor unit of a computer system to establish execution of the computation. The state word includes the contents of operand and address (index) registers, and registers whose contents put into effect the name/location map and establish the state of protection appropriate to the computation. Since switching a processor from one computation to another is accomplished by swapping state words, it is important to design for a small state word so that processor switching can be done at little cost.

3. *Simple Relocation*

First consider the addressing problems of a computer system that is multiprogramming a number of entirely independent computations. By "independent" we mean that no one of these computations ever references data or procedure information referenced by any other computation. The implications of common reference to information by several computations is considered later.

In the typical processor/memory configuration shown in Figure 1a the address serves two functions. First, addresses serve to distinguish among the words that must be referenced by a computation. In this regard addresses are *names* of information. Secondly, addresses specify particular *locations* in physical main memory where the required words reside.

Consider running several independent computations in this system as it stands with addresses of a bits in length. One alternative policy is to have only one computation in an operable state in main memory at a time. Then each computation has free use of addresses within the 2^a-element address space of the machine—both as names of information and as designators of physical locations. Switching execution from one computation to another in the system of Figure 1a requires dumping of information from main memory for two reasons: The new computation may require more space in main memory than is left unoccupied by the former, so that a reallocation of physical locations is necessary. However, dumping or moving within main memory also may be required because of conflict in the use of names even

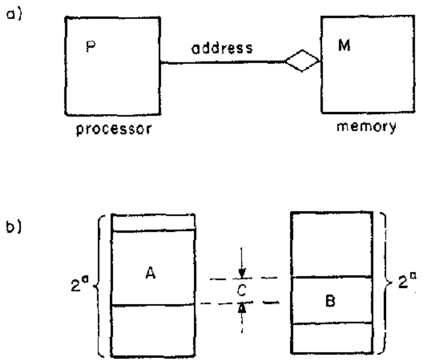


FIG. 1. Conflict of names in a typical processor/memory configuration

though there are enough physical locations available. Figure 1b shows the addresses being used by two computations A and B. Since the set of addresses C is used by both A and B for names of information, dumping is required although the totality of addresses used by A and B may be considerably less than 2^a.

This conflict in the use of physical memory locations arising from a conflict in the use of addresses as names can be avoided in the system of Figure 1 only by allocating nonoverlapping areas of the address space to each computation. This is exactly what has been done in a number of real time systems in which all procedure information is permanently assigned to physical locations in main memory. This alternative is not possible when the total naming requirements of the computations exceed the size of the address space.

The use of a relocation register as indicated in Figure 2a introduces a mapping of addresses as illustrated in Figure 2b. The *name space* N consists of those addresses that may be generated by a processor during execution of a computation. The *memory space* M consists of the set of addresses that correspond to memory locations. The mapping performed by a relocation register translates a contiguous set of addresses in N into a contiguous set of addresses in M. Such a mapping permits the data and procedure information relevant to a computation to occupy different sets of memory locations during successive periods of execution, while occupying the same position in name space. It is important to observe that the function of a relocating loader is not comparable to the function of a relocation register. The former merely permits a program to be placed at a choice of positions in name space *before* execution commences. A program cannot in general be moved within name space after execution has started without elaborate conventions [3] because information computed by the program is dependent on the values of addresses used as names.

The use of a relocation register gives the system freedom in the assignment of physical memory locations. However, dynamic allocation of main memory is made difficult by the fact that contiguous sets of addresses in N always map into contiguous sets in M. The technique usually employed is to periodically move the information relevant to different computations down in memory space to close up areas left unoccupied by suspended computations. New information is assigned locations at the top of M as shown in Figure 2c. This closing up or "compacting" operation requires moving information between memory locations which is wasteful

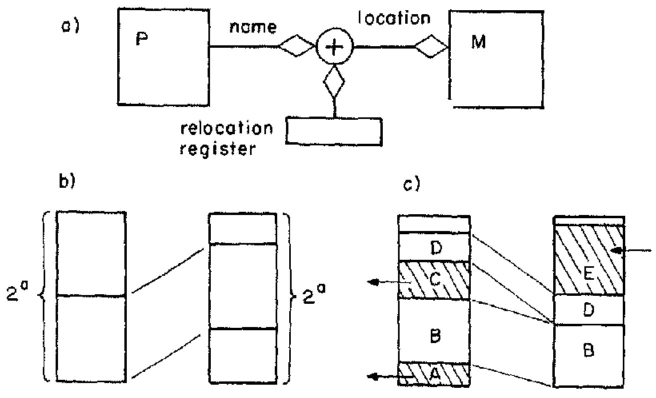


FIG. 2. The relocation register as a name/location map

as it accomplishes no actual reallocation of memory and will interfere with the operation of computations not involved in the reallocation.

The compacting technique is successful in systems where reallocation of main memory is a relatively infrequent event. Where many concurrent computations are continually changing their demands on main memory this technique appears certain to create an intolerable penalty in supervisory overhead.

4. *The Block Index*

The problem of successfully using the relocation register stems from its property of mapping contiguous sets of names into contiguous sets of memory locations. This restriction may be overcome by partitioning main memory into many equal-sized blocks of locations and assigning locations to computations in units of one block. The blocks are associated with the name space in which a processor is generating addresses through a *block index* as shown in Figure 3a. The name space is divided into *pages* of size equal to the block size, say 2^b. An address generated by the processor is split into a page number and line number and the page number is used to determine the associated block number by use of the block index. The block number and line number are concatenated to form the location address in main memory. As a mapping from name space N into memory space M, the block index scheme has the effect illustrated in Figure 3b. Note that the name space is not required to be the same size as physical main memory; here we show processor addresses (names) of n bits and memory addresses (locations) of m bits. The memory space may be either larger or smaller than the name space.

The block index provides an acceptable solution to the problem of allocating main memory among several computations. Once a page of information relevant to a computation has been assigned to a block of main memory, it need not be moved until it is no longer relevant, even though all other blocks might be reallocated between these times. When a new page is to be placed in main memory, one free block is as good as another and the problem to be solved by the system is just to decide which pages should occupy main memory at each instant.

However, the block index does not solve the problem of allocating addresses in

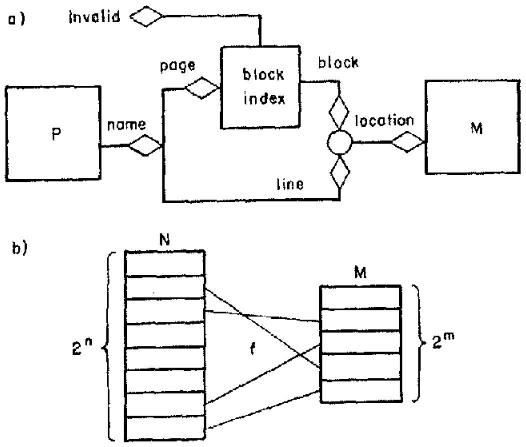


FIG. 3. The block index as a name/location map

name space among different computations. We observed earlier that with no name/location map (i.e., N identical with M) a conflict in the use of names made reallocation necessary. With the block index, reallocation of physical locations is avoided, but a conflict in the use of name space between two computations requires that the name/location map must be altered whenever control is shifted from one computation to the other. This could be a very time-consuming operation depending on the implementation of the block index. For example, the use of the block index for two computations, A and B, is illustrated in Figure 4. If each computation is to have complete freedom in the use of name space N, the name/location map must be changed whenever control is shifted from one computation to the other. Note that to establish protection, those pages of N outside the region of N containing information relevant to the operation of a computation A must be marked as invalid during execution of A.

Implementation of a block index. The information required to describe the name/location mapping determined by a block index depends on the nature of the index. Two possible arrangements of the index are: (1) as a table in which the block representing a page may be determined by a lookup on the page number, and (2) as a table with one entry per block containing the number of the page it represents. In the second case an associative search must be used to perform the lookup. The latter technique becomes appropriate only when the name space is considerably larger than main memory, so that a large saving in storage for mapping information is realized in return for the complexity required to implement a fast associative search.

The data that determines the name/location map may reside either in a processor or the main memory of the computer system. If it resides in the processor, there must be a memory element in the processor to hold this information. Under the assumption that a different name/location map will apply for each computation, the mapping information is an extension of the state word of a processor and must be set up whenever a suspended computation is to be resumed. On the other hand, placing the mapping data in the processor allows address translation to be accomplished fairly quickly, since a rather fast memory element could be justified.

Now suppose the name/location map is represented in main memory. Then the processor must have knowledge (a pointer, say) of where in memory the map is located. Changing between maps corresponding to two computations is simply a matter of changing the pointer in the processor. Thus the pointer is the only addition to the state word of a computation. However, a penalty of at least one main memory cycle is paid every time a reference to the map is required.

Look-behind. To combine the advantages of both choices to a certain degree, a

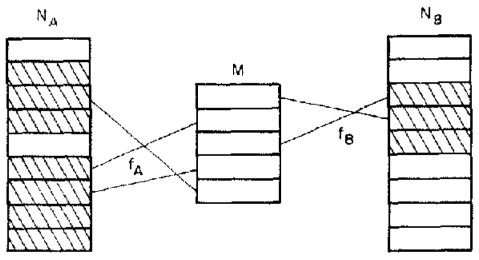
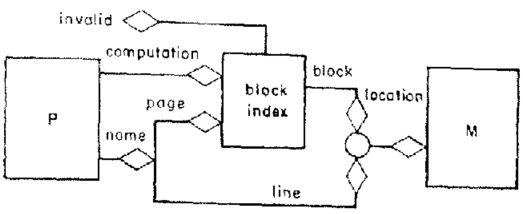


FIG. 4. Name/location maps for two computations



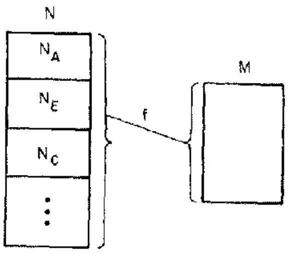


FIG. 5. The block index as a name/location map for all computations

mechanism can be designed that causes the mapping information most relevant to the present state of a computation to be kept in special, fast access registers of the processor. A scheme appropriate to an implementation of a block index is to keep the page and block numbers of a fixed number of most recently referenced pages in flip-flop registers of the processor. Whenever a memory reference is made, the page number is compared in parallel with page fields of the "look-behind" registers. If a match is found, the block number required is immediately available; otherwise a reference to the name/location map in main memory is made. The newly found page/block combination is placed in a look-behind register whose prior content is least likely to be pertinent to future memory references. Probably the simplest choice is to discard the entry least recently used.

With a pointer register in the processor that selects the block index relevant to the computation being run, the processor specifies memory references by two pieces of information —an address from the name space of a computation, and the contents of the pointer register which identifies the computation. The situation is illustrated conceptually in Figure 5. The block index now associates a page of some computation with each block of memory, and thus incorporates the name/location map of each computation. The processor generates names in the much larger name space that is the union of the name spaces of the component computations, and the name/location map defined by the block index is from regions within this name space into the memory space M.

5. *Use of Name Space by an Independent Computation*

Next we examine two issues concerning how a computation may employ addresses within its 2^n-element name space. These are the problems of overlay and variable-size data structures.

The overlay problem. Suppose a computation A requires reference to data and procedure objects A through E. As indicated in Figure 6, only certain combinations

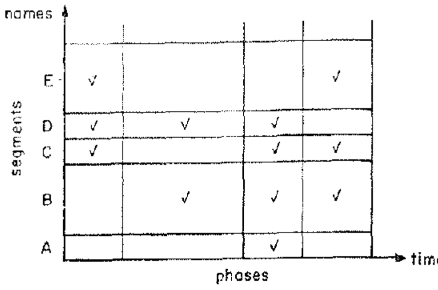


FIG. 6. Naming requirements for the overlay problem

of these objects are needed for immediate access during each of several phases of the computation. If the name space N_A available to the computation is sufficiently large, each data and procedure object may be assigned to a unique set of addresses in N_A. Then arbitrary combinations of the objects may be placed in main memory with no possibility of conflict in the use of addresses in N_A. If N_A is not large enough, distinct names cannot be assigned, and in some phases of computation A there may be a conflict in the use of addresses.

The resolution of such a conflict requires enlarging name space so that unique addresses may be assigned, or a renaming by computation A of objects within its own name space N_A. In the case of data objects, renaming is frequently done by double indexing or through indirect addressing, with either device requiring additional time and using a processor feature that might be used for other purposes by a computation. In the case of procedure objects the situation is more difficult as neither indexing or indirection are generally applicable to instruction fetch memory references. Renaming of procedure objects can be done only with additional hardware in the processor, for example, a "procedure base register" that relocates instruction fetches within name space.

Of course, the use of indexing or indirect addressing does not solve the whole problem. The computation must solve the dynamic allocation of data and procedure objects within its own name space. In this regard, note that a computation, in order to reallocate an existing object within its name space, must either move information between physical locations or cause the system to alter the mapping between N_A and M. Moreover, if distinct names for all entities of interest are not available within the addressing structure of the machine, the computation must establish a system for naming relevant information outside the addressing structure of the machine and maintain records of the correspondence between the two naming systems.

Variable-size data structures. Examples of variable-size data objects include arrays, list structures, pushdown lists and stacks. With the trend toward more online use of computation, it is increasingly difficult to determine the eventual size of a data structure at the time it is created. This consideration leads to the name space allocation problem shown in Figure 7.

In Figure 7a a large area in the name space of a computation is reserved for a data object D that starts as a rather short collection of words, but will grow to an unknown extent as the computation progresses. If D remains small the computation has denied itself the use of a sizeable portion of its name space.

On the other hand, if the name space is divided as shown in Figure 7b, such that

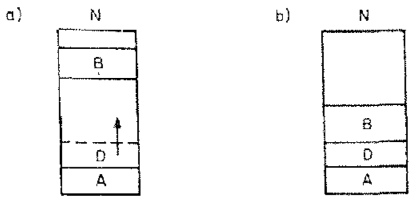


FIG. 7. The naming problem for variable size data structures

D has no room for expansion, it will be necessary to reallocate objects in name space if enlargement of D should be required. To avoid such a reallocation of information, the name space, even when a number of variable-size data objects are involved in a computation, must be *large enough so that it is only sparsely occupied by the totality of information the computation will ever reference.* The concept of segmentation provides a practical means of achieving this end. Before discussing this topic, let us consider another class of problems—ways by which several computations may reference common data objects or call upon common procedure objects.

6. *Referencing Common Information*

Arranging a multiprogrammed computer system so that two or more computations may reference the same copy of a procedure or data object in main memory presents two problems of immediate concern. The first is the naming problem: By what mechanism do the computations reference the common information? The second problem is one of protection: How is a computation permitted to reference information only in a manner that is authorized?

Naming common information. To study the naming problem, consider the situation shown in Figure 8. Let A and B represent the active procedure and data information relevant to two computations, each of which occasionally requires the use of subroutine S. Figure 8b shows these entities allocated to regions of locations in main memory. These regions are shown as contiguous sets in M for simplicity, even though they might in general be broken up by a paging mechanism. Figures 8a, c show the name spaces N_A and N_B seen by computations A and B respectively. During execution of A or B the name/location maps f_A and f_B respectively are presumed to be in effect. The question arises as to what name/location map is in effect during execution of the subroutine S. Two possibilities are apparent: (1) use f_A or f_B according as the subroutine is operating on behalf of computation A or B respectively, or (2) employ a distinct name/location map f_S whenever S is operating, whether for A or for B.

Consider the consequences of chosing the second possibility. Suppose S makes memory references within a private name space N_S established by a name/location map f_S that is put into effect whenever S is operating. This would have the disadvantage that a change of map must be made whenever S is entered or exited. What is worse—data on which A wishes S to operate must be assigned names in N_S so that they may be referenced by S. One cannot suppose that the same region of N_S can be assigned as in N_A, since there may be a conflict with S itself (see Figure 9a). The probability of such a conflict becomes considerably magnified where the use of many common subroutines is needed by a computation. Again one is

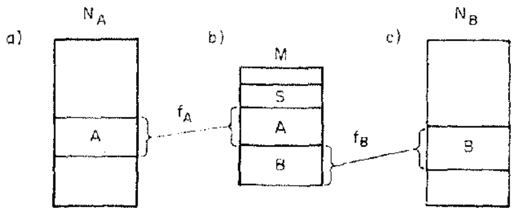


FIG. 8. The naming problem for a common subroutine

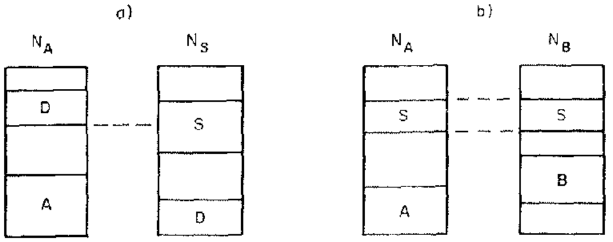


FIG. 9. Alternatives for common use of a subroutine

faced with the necessity of using distinct names for the same computation entity, and consequently we have the problem of establishing and maintaining records for a naming system outside the memory reference mechanism of the computer.

Next, suppose the subroutine S operates in the name space of its caller N_A or N_B. If N_A consists of all names the processor can generate while operating on behalf of computation A, then, during execution of S, a portion N_{AS} of these addresses must be "borrowed" from N_A for use by the subroutine for reference to itself (internal branch addresses and addresses of constant values within S). It follows that computation A must at some point set aside a region within N_A for S, again presenting A with an allocation problem in its name space. Also if a different region N_{BS} is reserved for S in the name space of computation B, references internal to S could not be consistent for both computations A and B. The only way of permitting one copy of S to operate correctly even though it occupies different positions in name space is to introduce a "procedure base register" that allows S to be arbitrarily relocated within name space. Such a base register simply performs an address transformation in addition to the mapping already postulated, and constitutes an elaboration of addressing mechanism which is a step in the direction of segmentation.

The need for a base register is avoided if S is assigned the same region within the name space of each computation with a requirement for S, as in Figure 9b. Then references by S internal to itself would be consistent regardless of whether S makes memory references through f_A of f_B. If this is so, consider the problem of having a number of procedures in M that *could* be used by several computations. To ensure that any procedure referenced by two or more computations could be assigned to the same position in their respective name spaces without overlapping with other procedures, every potentially common procedure must be assigned a specific position in the name space of every computation. To be possible, this requires that the name space of a computation be large enough to provide a permanent unique set of names for each potentially common procedure.

Protection. A protection scheme must distinguish between data that may be modified by a computation and information that may be referenced as procedure, since an erroneous computation must not be allowed to write into procedure information referenced in common by other computations. Where data is referenced in common by a group of several computations, reading may be authorized for all computations of the group, while the privilege of writing in the data is reserved to one computation. These points illustrate the dependence of protection on the nature of information and the authority of computations to reference it, rather than the location of information in memory. Hence it is preferable that a protection mechanism operate in the name space of a computation rather than the space of locations where protection data would be continually invalidated by reallocation activity.

7. *Segmentation*

The arguments presented above are intended to point out the inadequacies of several frequently suggested techniques of meeting the addressing requirements of multiprogrammed computer systems. In studying the limitations of these techniques we have formulated some criteria by which new approaches may be judged.

(1) A computation should have the use of a name space sufficiently large that all information it references may be assigned unique names, and such that reallocation of information within its name space is never necessary.

(2) Data objects of a computation should be expandable without requiring a reallocation of name space.

(3) Information referenced in common by several computations should have the same name for all computations that reference it.

(4) A protection mechanism should operate in name space to permit access to information by a computation only in an authorized manner.

These requirements can be met by a system in which a computation addresses information by generating a *segment name* and a *word address* for each memory reference.

A *segment* is an ordered collection of words with an associated segment name, from which a particular word is selected by the word address as shown in Figure 10a. The number of bits a in the word address is chosen to allow for the largest collection of information that is to be addressed as an ordered array of words. The number of bits s used to represent a segment name is chosen large enough for all segments needed by concurrent computations in a multiprogrammed system to be distinguished. A computation then has use of a name space with $2^a \cdot 2^s = 2^{a+s}$ elements, as shown in Figure 10b.

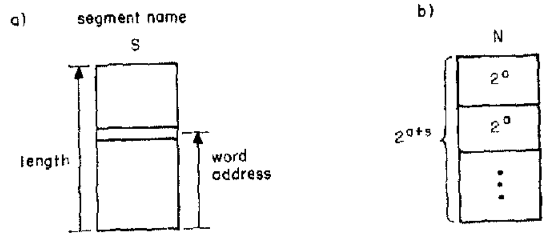


FIG. 10. A segment and the name space of segments

A segment has a length, equal to the number of words it contains, that may vary during the course of a computation. The information of interest to a computation at any time will fill only a tiny fraction of the name space, as it is likely to concern just a few segments, most of which will be miniscule compared to their potential length. Thus, the name space is large enough that once a name (consisting of a segment name and word address) is associated with an information word it remains associated throughout the course of all computations that require reference to it.

Each procedure or data object to which common reference must be made by several computations is represented as a segment distinct from all segments private to a computation, so that it may be referenced by name. All computation processes generate addresses within the same name space so they may potentially address any segment (procedure or data) in common.

A protection system must be implemented so that a computation may only make authorized reference to memory. It is natural to make the segment the basis for protection. We say that each computation is executed within a *sphere of protection* that permits it to reference such segments as authorized, and these segments only in a designated manner. Three classes of reference to segments are distinguished:

(1) *Procedure.* The segment may be executed as pure procedure but not referenced as data.

(2) *Read only data.* The segment may be read as data but not written or executed.

(3) *Data.* The segment may be read or written, but not executed as procedure.

8. *Implementation*

The mechanism discussed here for implementing the notion of segmentation is chosen to best illustrate the concepts and is not expected to represent a best compromise for an actual system design. A practical system designed around these concepts is presented in a paper by Glaser, Couleur and Oliver [4].

A processor running a computation must specify a sphere of protection, a segment name and a word address for each memory reference. Since it is not practical to

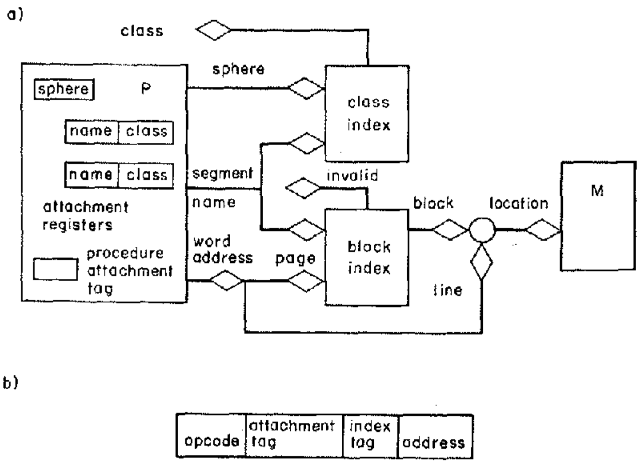


FIG. 11. An implementation of segmentation

include a segment name in the address field of each instruction, an abbreviation scheme must be adopted. As shown in Figure 11a a processor is equipped with a small number of *attachment registers* which a computation may load with segment names. The instruction format is then arranged as in Figure 11b to include an attachment tag that selects one of the attachment registers for data references and branch addresses. The *procedure attachment tag* register specifies which attachment register is to be used for instruction fetches. It is loaded by transfer-of-control instructions. Protection is accomplished by permitting a computation to load only attachment registers with the names of segments valid within its sphere of protection. When an attachment register is loaded a class code becomes associated with it that specifies the manner in which the segment may be referenced.

Figure 11 also shows an arrangement for applying mapping information represented in a *segment class index* and a *block index* for effecting memory references by a computation. The segment class index is consulted whenever a processor attempts to load an attachment register to determine whether the segment is valid and, if so, to determine its class for the present computation. Each segment in main memory is represented by an integral number of equally size blocks that are associated with pages of the segment by the page index. On each memory reference, the page number is masked off the word address and is used with the segment name from the pertinent attachment register to find the corresponding block number from the block index.

9. *Conclusion*

The ideas presented in this paper have grown from experience gained while implementing general purpose time-shared computer systems at MIT [5, 6], together with and our desire to create a second generation of computer utility with greater power and efficiency of operation. It is our conviction that these concepts are applicable to the design of online systems in general [7].

Acknowledgments. Our thinking has been guided by the work of Holt [8] in characterizing the dynamic storage allocation problem and we have adopted much of his terminology. We recognize the pioneering steps of the Rice University and Manchester University groups [9, 10] in exploring hardware features for aiding storage allocation.

The writer has benefited greatly from numerous discussions with his colleagues, F. J. Corbato, J. Couleur, E. L. Glaser, R. Graham and E. C. Van Horn, during the course of the work.

RECEIVED FEBRUARY, 1965

REFERENCES

1. CORBATO, F. J., MERWIN-DAGGETT, M., AND DALEY, R. C. An experimental time-sharing system. *AFIPS Conference Proceedings 21*, National Press, Palo Alto, Calif., 1962, pp. 335–344. Now available from Spartan Books, Washington, D. C.
2. SCHWARTZ, J., COFFMAN, E. G., AND WEISSMAN, C. A general purpose time-sharing system. *AFIPS Conference Proceedings 25*, Spartan Books, Washington, D. C. 1964, pp. 397–411.
3. CORBATO, F. J. System requirements for multiple access, time-shared computers. Proj. MAC Tech. Rep. MACTR-3, MIT, Cambridge, Mass., 1964.
4. GLASER, E. L., COULEUR, J., AND OLIVER, G. System design of a computer for the time-sharing application. To be presented at the Fall Joint Comput. Conf., Las Vegas, Nev., Nov. 1965.

5. MIT Computation Center. *The Compatible Time-Sharing System, A Programmer's Guide.* MIT Press, Cambridge, Mass., 1964.
6. Dennis, J. B. A multiuser computation facility for education and research. *Comm. ACM 7*, 9 (Sept. 1964), 521–529.
7. —— and Glaser, E. L. The structure of on-line information processing systems. Proc. Second Cong. on Information Systems Sciences, Homestead, Va., Nov., 1965, pp. 1–11.
8. Holt, A. W. Program organization and record keeping for dynamic storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 422–431.
9. Iliffe, J. K., and Jodeit, J. G. A dynamic storage allocation scheme. *Comput. J. 5* (Oct. 1962), 200–209.
10. Kilburn, T., Edwards, D., Lanigan, M., and Sumner, F. One-level storage system. *IRE Trans. EC-11*, 2 (Apr. 1962), 223–235.