MAC-TR-11

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

PROGRAM STRUCTURE IN A MULTI-ACCESS COMPUTER

by

J.B. Dennis

"Work reported herein was supported (in part)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."

This empty page was substituted for a blank page in the original document.

I. Introduction

A multi-access computer (MAC) system 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 procedures and data may be 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 wealth of data and procedures that are accessible to a large user community through the file memory of a MAC system.

The practicality of the MAC concept depends on the idea that the power of a large computer system should be a better match to the union of many diverse tasks than it is to any particular one. The amount of main memory actually required for efficient execution of a procedure varies from a few hundred words to many times the size of memories in existing machines, depending on the nature of the procedure. Moreover, the memory requirement of a procedure typically varies over a wide range during its execution. If a number of diverse procedures can share a large main memory, the total memory requirement will be subject to less fluctuation with time in consequence of the statistics of sums. Procedures also differ in the frequency with which interactions with the machine environment interrupt processes in execution, and the length of pauses that result. If many processes are available for execution in a machine structure, the statistics will insure that the processing units of the system will be kept more fully occupied than would otherwise be possible.

---

* with apologies to E.E. David

For a computer system that places particular emphasis on strong interaction with a user community, it is evident that memory and processing capacity must be freely reassignable among the active processes. The time scale desirable for reallocation events to take place in a MAC system is certain to be several orders of magnitude beyond what has been accomplished or contemplated with existing systems.

The formulation of a computer system organization and operating philosophy raises many important questions. Two broad issues concern us in this paper:

1. What features of machine design are necessary or desirable to facilitate dynamic allocation of computation resources among many concurrent processes?

2. What are appropriate policies for governing the allocation of machine resources to insure their effective utilization, and through what techniques should these policies be implemented?

For evaluation of machine organization and features, and for realistic study of the resource allocation problem, a suitable model of program structure is required. It is no longer adequate to consider a program as occupying a single block of memory and requiring a specific length of time for execution. The varying demand of a program for space in main memory, the referencing of common procedures, data, and files by several programs, the possibilities of parallel processing, and the rate of interaction with environment in a MAC system require a more sophisticated view of program structure.

In the following paragraphs some thoughts are developed that may form a reasonably adequate model of program structure. These concepts have grown out of many discussions with colleagues in Project MAC*, and our experience to date in the design and operation of multi-access computer systems.[1,2] The work on dynamic storage allocation reported by the Atlas group[3] at Manchester and the Rice University group[4] are pioneering steps toward the objective of our research. The formulation of the storage allocation problem in terms of segments of memory and phases of execution by Holt[5] has been very influential on our thinking. At this writing, the ideas do not form a consistent whole,

---

* and E. Van Horn in particular

but it is hoped they will serve as the basis for a more comprehensive study. Later in the paper several machine features that have been inspired by program structure concepts are introduced. Although it is felt that these specific proposals might prove quite valuable in a multi-access computer system, they should be regarded as very preliminary in nature with considerable further study and refinement being required before incorporation in a computer system. For the purpose of this paper, the machine features discussed are primarily examples of directions in which models of program structure may be expected to influence the design of future machines.

II. Some Concepts of Program Structure

Segments

The first and most important concept is to regard a program as concerning information that is grouped into a collection of objects that we shall call segments. A segment is an ordered set of words. It is referenced by a name that distinguishes it from all other segments and an integer address that selects one word from the ordered set of words that constitutes the segment. At any time a segment contains a definite number of words called the length of the segment. The length may vary arbitrarily as a computation process progresses. From an abstract viewpoint, the permissible length of a segment should range over all integer values from zero to infinity. In practice, a limit is placed on segment length by the number of words available in main memory.

Each segment referenced in the course of a particular computation process has an associated class that determines what forms of reference to it are valid. 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.

In this discussion it is assumed that procedure segments are in pure procedure form; that is, reference to a segment as procedure never alters information contained in the segment.

Processes and Phases

A segment being actively referenced as 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 multi-processor computer system, a number of processes may be in execution simultaneously.

At any instruction step a process is actively referencing exactly one segment as procedure and one or more segments as data. Observe that the pure procedure convention for procedure segments permits several processes to actively reference the same procedure segment without interference.

During one phase of a process, a certain group of procedure and data segments are in working status. From the programmer's viewpoint the set of working segments constitutes the collection of information that should reside in main directly addressable memory for efficient execution of the current phase of his procedure. Transition of a process from one phase to another occurs when a new segment must join the working group, or a working segment is dropped from the group.

Creation and Annihilation of Processes - Input/Output

We suppose that each input or output step of a procedure refers to an input or output unit identified by an input/output device name. An input/output procedure step includes the appropriate device name. A process is terminated by 1) a stop instruction, or 2) an input/output step referencing a device that is not ready. In the second instance we say an input/output pause has commenced. A process is created by 1) another process - a fork, or 2) the end of an input/output pause.

Declaration of Segments and Phases

It is expected the spatial division of information into procedure and data segments and the temporal division of a process into phases is determined by the programmer in consort with a programming language system. This view is taken because the person who is preparing the description of a procedure is in the best position to specify when a segment is to join the working collection of segments, and -- what is more important -- when a segment is no longer needed in the working collection. For this purpose, programming language systems should be designed so that the programmer is encouraged to segment his procedure and data in a sensible manner. Of course, a programming system should be capable of generating a reasonable segmentation and phasing of a procedure in the absence of declarations by the programmer.

Powers of a Process -- Meta-Instructions

The foregoing discussion has endued processes with certain powers relative to segments and processes. These powers are realized through special procedure steps termed meta-instructions. Functions performed by meta-instructions include:

1) Creation of new data segments or read-only data segments.
2) Erasing an existing segment.
3) Initiation of a new process -- a fork.
4) Termination of the process.
5) Entering a segment in the working collection.
6) Deleting a segment from the working collection.
7) Requesting assignment or release of an input/output device.
8) Changing the length of a data segment.

Active and Inactive Segments

The form taken by the name of a segment in a working process will depend on means chosen to make references to segments effective during execution and will be called the effective segment name. A specific way of mechanizing references by segment name is suggested in a later section of this paper.

The form taken by effective segment names will be limited in bit length by considerations of hardware and programming economy. Therefore, the number of distinct effective names will be finite, but must certainly be large enough to cover all working segments at any time. Since the cost of adding a bit to the length of segment names is not great and the total number of working segments is unpredictable it is appropriate to choose a length such that the expected number of working segments requires only a small fraction of all possible effective segment names.

Some segments participating in the course of a process are created by the process itself. Other segments referenced by the process constitute procedure and data objects normally residing in file memory. We will use the term file name to designate the descriptor (including the context in which the descriptor is used) that selects a procedure or data segment for retrieval from file memory. The set of file names of segments in an operational system will, in general, have an elaborate prefix structure in consequence of the hierarchy of user groups, the characteristics of programming language systems, and the interrelations among public procedures. The necessity for this prefix structure makes it difficult if not impossible to specify the required length of a direct binary encoding of the file names of segments.

We will say that a segment is active whenever it has an associated effective segment name such that references to it arising during the execution of any process are effective. If no effective name is associated with a segment, the segment is inactive. To clarify the meaning of these terms, we suppose the computer system is operated in such manner that the following conditions are met:

1) All segments occupying main memory are active.
2) The mass memory is divided into two functional parts - auxiliary memory and file memory.
3) All segments occupying auxiliary memory are active.
4) All segments occupying file memory are inactive.

Condition 2 is not meant to imply that auxiliary memory and file memory are physically distinct in a MAC system. In a practical realization of a multi-access computer system the main memory will be finite in size and, in general,

the sum of all working segment lengths will substantially exceed this capacity. The auxiliary memory serves as an extension of main memory used to keep working procedure and data segments not currently in main memory for execution.

It is important to understand that two categories of decisions have been implied by our discussion - those made by the user or his programming system, and executive decisions made to effect allocation or scheduling functions.

1) The decision of which segments have working status for a process is part of the specification of the process. Thus, these decisions are made by the user or the language system within which he is working.

2) The decision to move a segment between main memory and auxiliary memory is concerned with allocation of main memory. Decisions of this type implement executive or supervisory functions of the system.

3) The insertion of program forks and the termination of processes form part of the description of a procedure and are specified by the designer of the procedure.

4) The assignment of physical processing units to processes is a supervisory function.

Conservation of Effective Names

The process of making a segment active occurs with the first occurence of the segment file name during the execution of any process. At that point an effective segment name must be taken from a pool of available effective names. The segment must be retrieved from file memory, and its association with the selected effective name must be established to permit working references.

A working segment created through the execution of a process is automatically associated with a unique effective name from the pool of effective names, by the act of its creation. Similarly, a process may erase a segment, thus returning its effective name to the pool. A supervisory process must have the power to revoke names issued to a process if the process has hogged many names

for an excessive time. In normal operation, it would not appear unreasonable for a user to retain some associations of effective names for an extended period, perhaps many months, were this required by the nature of his work.

Spheres of Protection

One cardinal principle in the design of a MAC system is that a computation proceeding for one user must not interfere with correct execution of any other computation. Each ongoing process in the computer system is concerned at any time with a certain group of procedure and data segments and with certain input/output devices. The process must be denied access to segments and devices that is not properly authorized. This is necessary so that possibly faulty programs may be run in the system without endangering other computations. On-line program debugging would not otherwise be practical. It is convenient to think of each process as operating within a sphere of protection* containing all segments that may be legally referenced and input/output devices with which the process is permitted to communicate. References by a process to segments or devices not within the sphere of protection are illegal and result in termination of the process.

It is helpful 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 immediate superior of B, and B as an immediate inferior of A. We suppose there is exactly one sphere of protection that has no immediate superior and is called the master sphere.

The set of all spheres of protection together with the superior-inferior relation form in general, a tree in which the master sphere is the vertex. In this tree a sphere A is superior (inferior) to a sphere B if there is a downward (upward) path in the tree from A to B. In later paragraphs we discuss reasons for permitting the hierarchy of spheres of protection to have many levels. In relation to the hierarchy of spheres of protection, processes must have further powers realized through meta-instructions. If sphere B is an immediate inferior to sphere A, a process in A must be able to:

---

* After E. Van Horn

1) create B.
2) enter a segment valid in sphere A as valid in sphere B.
3) initiate a process in sphere B.
4) terminate all processes in sphere B.
5) delete sphere B, and in consequence all spheres inferior to B.

The relation between spheres of protection would not be completely specified without mention of exceptional conditions. A procedure step encountered by a process that is meaningless in its sphere of protection causes an exceptional condition. Examples are a reference to an invalid segment or device name, a non-existent address within a segment, or an undefined operation code. An exceptional condition arising in a process terminates that process and initiates a specific process in the immediately superior sphere of protection.

Program Development

The user of a MAC system develops a new program by communicating with a programming language system. Suppose the processes performed by the programming system on behalf of one user are carried out in a distinct sphere of protection we shall label A for short. These processes create a number of segments which are referenced as data in sphere A and constitute the coding of the user's procedure. To perform the user's procedure, sphere A creates an inferior sphere of protection B in which the segments of the user's program appear as procedure or data, according to declarations made to the programming system, and then initiates a process in sphere B. Exceptional conditions arising in sphere B terminate the process and reestablish a process in sphere A. Exceptional conditions should not occur in the execution of the language system procedures in sphere A as they are presumably debugged programs. If one does occur a process is created in the sphere C that is immediately superior to A.

The reasons for placing sphere B inferior to A rather than directly under C are several. First, it is natural that the programming system in A should have the power of creating, deleting, and allocating resources to sphere B. Second, the programming system in A is aware of the interpretation to be made for exceptional conditions encountered by a process proceeding in B, whereas exceptional conditions arising in the programming system itself require action by a higher system.

Clearly, it could readily be desirable to extend the superior-inferior relationship to more levels: A user may be debugging a programming language system; a teaching program may run under a programming system, and interact with many students whose data must be held confidential.

Allocation and Scheduling

We assume that it is essential for successful operation of a MAC system 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, modification of segments containing the current allocation of devices,main memory, effective segment names and other system resources, must be disallowed for any process except one operating in the master sphere. Thus, a process wishing to have a system resource assigned or released from its domain must communicate with the master sphere (by means of meta-instructions).

It is envisioned that processes in the master sphere serve the following functions:

1) Maintain allocation tables and prevent conflicts in assignments.
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 authorize allocation of system resources to spheres inferior to itself, and execute allocation and scheduling acts by communicating with the master sphere. 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 a supervisor in a MAC system so that executive functions are done by 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. Furthermore, the effects of hardware or program failures occuring in supervisory operation could be confined to a limited part of the supervisory system - only master sphere failures would be catastrophic.

III. Machine Features

Memory References by a Processing Unit

To exploit the segment structure of programs, it is evident that a processing unit must supply the name of the intended segment as well as the address whenever reference is made to main memory. Including the segment name as an extension of the conventional address is impractical for several reasons: For any reasonable length of effective segment name, the efficiency of procedure representation in memory would suffer badly. Secondly, since effective segment names are not assigned until execution time, including them directly in the instruction format would require violation of pure procedure coding.

A solution is to include several special registers called attachment registers in the processing unit as in Figure 1a. The data attachment registers can be loaded with segment names by instructions open to all processes. The typical single address instruction code format is then expanded slightly as shown in Figure 1b to include a field that selects the data attachment register containing 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 could be automatically loaded from one of the data attachment registers when a transfer of control or a subroutine entry instruction is executed.

Storage Mapping Hardware

The storage mapping hardware discussed below was devised with the following objectives:

1) It should be possible to redistribute main memory when working reference to new segments is required without having to move segment content between physical memory locations.

2) Modification of segment content should not be necessary to preserve effective references among segments when the allocation of memory is changed.

These objectives are accomplished by interposing two control memories called the segment index and the page index, and some control logic, between the processing unit and main memory, as shown in Figure 2. For simplicity only one processing unit is presumed, though the principle is equally valid for a multiprocessor system. The segment index contains entries, each consisting of a sphere name-segment name pair and a code that indicates the nature of references to the segment that are legal within the associated sphere of protection. Whenever the process in execution 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 an exceptional condition that terminates the process. From the forgoing, it is evident that the attachment registers will only contain segment names to which valid references may be made within the current sphere of protection.

The page index is used to rename equal-size blocks of main memory, and contains one entry for each block of main memory. Each segment consists of an integral number of block-sized pages. Therefore, an address within a segment is broken into the concatenation of a page number and a line number 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 techniques. The effective address is split into page number and line, 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. The page number and block number are loaded into an extention of the attachment register so further references to the same page do not require use of

the page index. If no match is obtained in the page index, the reference was to an address outside the current bounds of the segment and an exceptional condition exists.

The equality searches required in the segment index and page index could be performed by hardware asspciative memories. However, pseudo-associative memory realized through conventional location addressed memory and hash addressing is presently more economical and probably faster on the average. The page index memory must be very fast, as a reference to it is needed for a sizable fraction of main memory references. Its size is rather small, e.g. 1024 entries for a main memory of 2^{20} words partitioned into 1024-word blocks. The segment index memory does not have to be so fast, but requires a number of entries that is dependent of the nature of the processes active at any time. The segment index might, itself, be one of the segments sharing the main memory.

REFERENCES

(1) M.I.T. Computation Center, The compatible time-sharing system: A programmer's guide, M.I.T. Press, Cambridge, Mass., (1963).

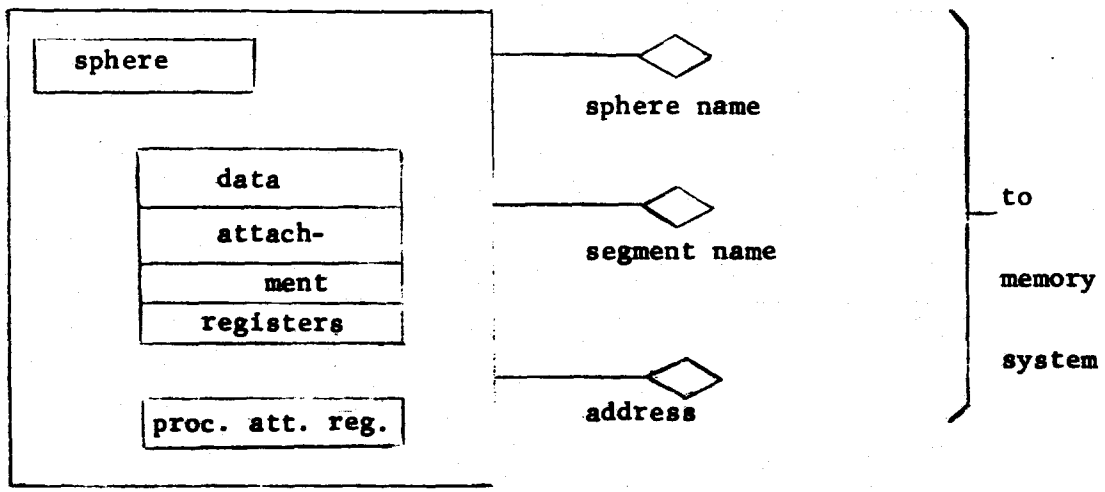
(2) Dennis, J.B., A multiuser computation facility for education and research, Comm. ACM, Vol. 7, no. 9, pp. 521-529 (September 1964).

(3) Kilburn, T., et. al., One-level storage system, IRE Trans. on Elect. Comp., Vol. EC-11, no. 2 (April 1962).

(4) Iliffe, J.K., and J.G. Jodeit, A dynamic storage allocation scheme, Computer J. Vol. 5, pp. 200-209 (October 1962).

(5) Holt, A.W., Program organization and record keeping for dynamic storage allocation, Comm. ACM, Vol. 4, no. 10, pp. 422-431 (October 1961).

a)



special processor registers

b)

opcode	attachment tag	index tag	address

modified instruction format

Figure 1 - Generation of Memory References

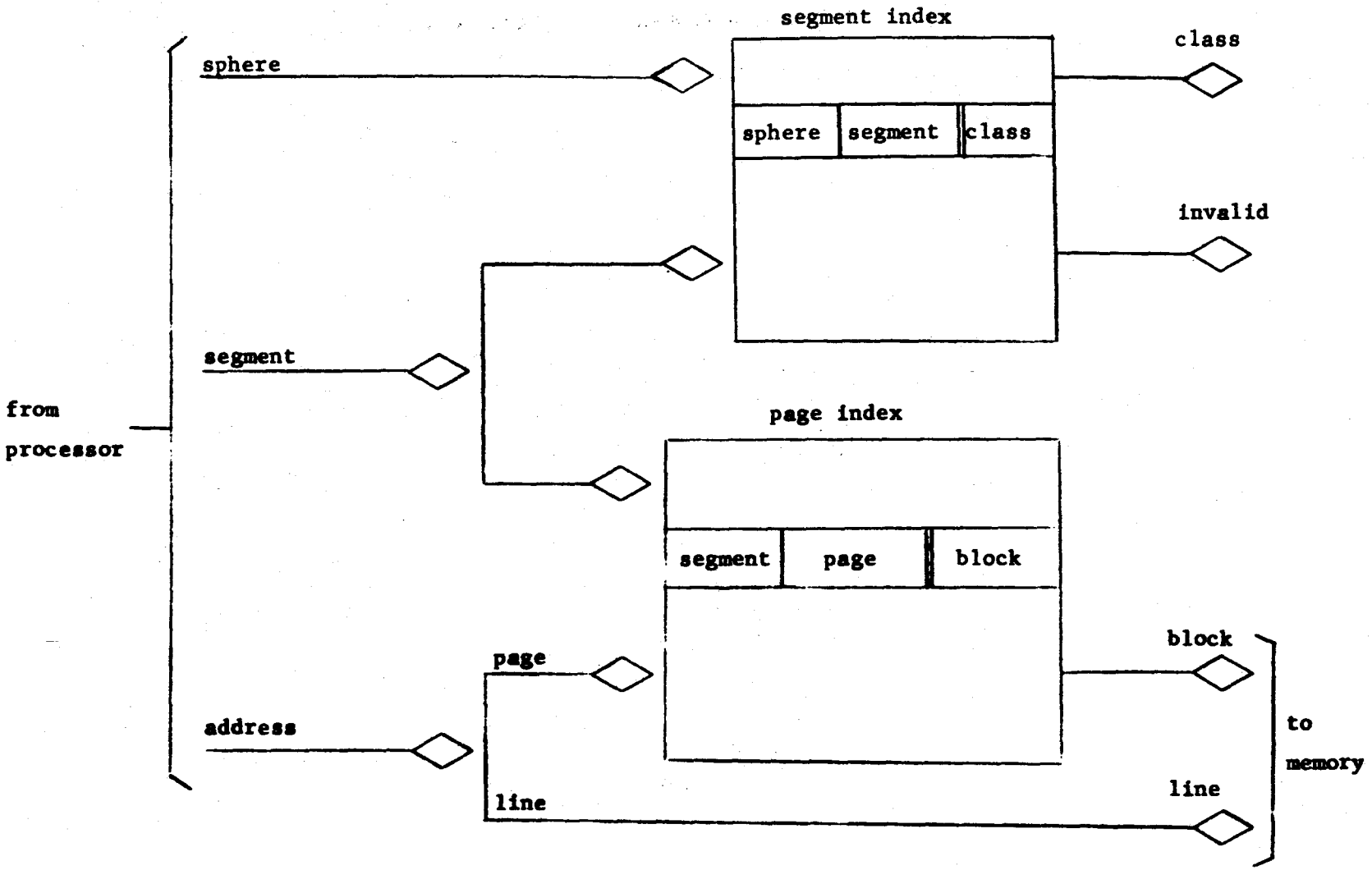


Figure 2 - Storage Mapping Hardware