MODELLING A COMPUTER SYSTEM UTILITY -

CONTROLLED SHARING OF INFORMATION *

Michael J. Spier
Massachusetts Institute of Technology, Project MAC
and
Elliott I. Organick
University of Houston, Department of Computer Science

ABSTRACT

This is the first of a sequence of planned papers that develop a model for a computer system utility. The present paper describes those parts which provide services for the protection and controlled sharing of private information. The requirements for these services are defined, and a computer, named the "ideal computer", is postulated with hardware-implemented capabilities for meeting these requirements.

It is demonstrated that the ideal machine effectively supports protection as well as controlled sharing of information in a fully general sense. It is shown that it is possible to build the desired system utility model on the ideal machine within the framework of its built-in protection capabilities, making it possible to a) dispense with the customary "supervisor/user-master/slave" type modes of operation, and b) further develop any desired structures of mutually protected interacting subsystems.

Some capabilities of the model systems are mentioned. Among these are: a) the ability to support parallel processing computations and b) the ability to support tree-structured hierarchies of computations (processes).

1. INTRODUCTION

In this paper we report on our research toward a model of a computer system utility which provides its users with an idealized programming environment as one in which a user may successfully communicate the exact nature of his problem to the system (i.e., by writing a program) in some suitable language which is meaningful both to him and to the system. In this activity he may, at no penalty, be unaware of hardware/software machine dependencies related to extent or immediacy (speed) of storage, processor or i/o device resources. Moreover he should also be free of concern or of need-to-know the internal structure of other objects in the computing environment.

Specifically, we would like our idealized system to meet the following requirements:

a The system must "understand" programming languages which are meaningful to the programmer and which meaningfully apply to the problem at hand.

b The programmer need not know about the art of programming any more than is absolutely necessary in order for him to be able to correctly state his problem.

c The system must be capable of supporting, for any one given user, any number of coexisting sequential- or parallel processing computations.

d The system must be capable of supporting any number of coexisting users.

e The system must possess an infinitely-large on-line storage in which all of a user's private information may be permanently stored and directly accessible. It must be possible for the user to organize his private information into any structure convenient to him.

f The system must provide absolute protection for the user's private information, and guarantee the inviolable integrity and privacy of that information. It must be possible for the user to further protect any subset of his information from instances of himself.

g It must be possible for the user to dispense with some or all of the protection of any subset of his information, and to grant one or more other users (or other instances of himself) controlled access to that subset; the word "controlled" implies that the user must have the ability to precisely define the kind of access granted to any other user (or instance of himself).

h We seek a generalization to allow us to apply

* 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) and in part by Project Themis, at the University of Houston under Office of Naval Research Contract Number N00014-68-A-0151. Reproduction in whole or in part is permitted for any purpose of the United States Government.

the capabilities described under requirements a through g to the software parts of the system itself, achieving a simplifying compatibility between "system" and "user" programs, and making it possible to dispense with the customary "supervisor/user"- "master/slave"-type modes of operation.

This working paper deals with requirements f, g, and h and describes the parts of the computer system model which provide services to fulfill these requirements. Requirements b through e may then be realized by building upon the partial model described in this paper, and are the subjects of successor papers planned for the future. We consider requirement a to be satisfactorily met through the postulated availability, on our model system, of high-level language (e.g., FORTRAN, ALGOL, COBOL, PL/1, BASIC, APL etc.) compilers and interpreters. We note also that satisfying requirements f and g also provides a basis for achieving objectives of programming generality or "building on the work of others" as described by Dennis [1] and Fano [2].

The development of our model requires that certain assumptions be made concerning the hardware machine on which it is to be implemented. In order to be able to develop a model of sufficient generality, we postulate a machine which we name the ideal machine and which consists of a single very large primary (e.g., core) memory, and a very large number of independent processors which are each potentially capable of accessing any given part of the memory. We assume that the present state of the art is such that an implementation of our ideal machine (through an appropriate use of special-purpose hardware and software) is practically feasible. By building our desired model on the ideal machine, we conveniently circumvent the problems of resource limitations (both qualitative and quantitative) which would only obscure the issues at hand.

This approach enables us to develop a generalized model, defined in terms of formalized abstractions, whose correctness may be judged by logic alone. An actual implementation of an operating system based on our model will of course require that a certain amount of generality be sacrificed in order to comply with the practical demands of real life.

Our model, while essentially new in the approach to its construction has been built on the work of others. The multiple contributions of Dennis and coworkers has been especially important to us. Thus the concepts of segments, spheres of protection and capability lists of Dennis and Van Horn [3] form a basis for this work. The attempts to exploit such ideas by Ingve and Fabry, as reported by Wilkes [4] added further to our understanding and to our ability to apply these ideas in our own model. The previously mentioned objectives of programming generality and its relationship to controlled sharing of information as pointed out by Vanderbilt [5] has also provided motivation. Concepts of protection, access control, intersphere call etc., used in our model are also related to corresponding ideas expressed by Evans and Leclerc [6] and by the several Multics designers [7,8, 9,10].

There clearly exists a gross correspondence between the objectives and components of our model and those of certain existing or proposed systems (Corbato and Vyssotsky [7], Hauck and Dent [11], Cleary [12], Wilkes [4], Vanderbilt [5].) Significant differences may show up as we are able to make a careful study of these related models. Hopefully (and subsequently) a searching comparison of this kind will lead to increased understanding and additional simplification. For this paper however we have limited our objective simply to an explanation and examination of our model, hoping to receive some suggestions for its further improvement.

The body of this paper (Sections 2 and 3) develops the model in a step-by-step, almost-reference-manual fashion. Ideally some synopsis would be desirable here so that the reader might more fully anticipate what is developing as he proceeds with his reading. Such an overview has been provided in a companion paper [13].

2. THE IDEAL MACHINE

The ideal machine consists of two types of devices, the ideal memory and the ideal processor. The machine has a single very large ideal memory which may be structured into any number of distinctly addressable segments (the term is defined below), and a very large number of ideal processors each of which is capable of addressing any given segment of memory. Every ideal processor is a distinctly named and addressable device.

2.1 THE IDEAL MEMORY

The ideal memory is a device consisting of three components:

data storage which is an array of consecutive elements, each of which is capable of storing a single bit of information. Each element is addressable by an integer S, its index.

descriptor storage which is an array of consecutive elements, each of which is capable of storing a single descriptor (the term is defined below). A descriptor is addressable by an integer D, its index.

memory controller which is the only device capable of accessing an element of either the

data or the descriptor storage. Any access to storage must be done through the intermediary of the memory controller.

2.1.1 Segmentation

A segment is an array of consecutive data-storage elements, defined by the integer pair [S, n] where S is the address of the first element of the array. We define no a priori structure of segments, they may be defined and redefined, recursively and overlappingly.

A segment is also associated with a data type which is a predetermined integer defining the way in which the information items within the segment should be interpreted. Examples of data types are: a double precision floating-point variable, a boolean variable, an instruction, a variable-length character string, etc.

The memory controller considers a segment to be an array of information items as defined by the data type. (A segment which may consist of a single variable would be treated as an array of size one.) The segment is the only addressable unit of data-storage in our ideal machine.

2.1.2 The descriptor

A descriptor describes a segment; we also say that a descriptor "points" to a segment. Any number of descriptors may point to any one segment, or to any subset of a segment. A segment may only be indirectly addressed through a descriptor which points to it.

The descriptor is a structured variable. We shall use the notation D.x to refer to element x of structure D. The descriptor is structured as follows:

D.pointer is an integer pair which may be either a segment pointer [S, n] or an indirect pointer [D, i] where D is the address of a descriptor pointing to the segment and i is the index of an information item within that segment. Indirect addressing may be recursive to any desired level, provided that the recursion be terminated with an [S, n] segment pointer.

D.dtype is the segment's data type. For the purpose of this paper we need only concern ourselves with the "instruction" data type, as described further on.

D.specifier is an item which defines certain properties associated with the segment (e.g., size of array) and the data type (e.g., length of character string), and does not concern our discussion.

D.access is an access name, which is an integer. Access names are defined in section 3.

D.owner is an access name.

D.attributes is an array of binary indicators. A binary indicator, known by the name of attribute, is addressable by an integer i, its index. An attribute may assume either of the values TRUE or FALSE.

Our machine features a set of privileged instructions (we shall discuss later the requirement necessary for the execution of a privileged instruction) to access and manipulate a descriptor. Excepting those instructions, there is no way with which a program may access a descriptor other than for the purpose of indirectly accessing a segment.

2.2 THE IDEAL PROCESSOR

The ideal processor is a device which is capable of performing an ordered sequence of computational operations. Before defining the term "computational operation", we wish to describe some elements of the ideal processor.

We begin with the definition of the following two terms:

instruction an instruction is an information item which is composed of an operator f followed by a list of operand addresses a, and may be represented by the notation $f(a_1, a_2, \ldots, a_i, \ldots, a_n)$. The operator specifies the operation to be performed on the operands addressed by (a_1 through a_n). An operand address a_i may refer to a segment, or to an element of a segment, or to a descriptor, or to an internal processor register.

procedure a procedure is a segment which consists of an array of instructions. An instruction may be addressed through an instruction pointer which is an indirect address pair [P, i] where P is the address of the descriptor for the procedure segment, and i the index of the instruction. (note: we use notation P rather than notation D to refer to the procedure segment's descriptor. We do this for reasons of convenience and readability; notation D could have been used.) A segment which is not a procedure segment is said to be a data segment.

The ideal processor contains the following internal registers which enable it to perform an ordered sequence of computational operations

(see also Figure 1):

P-register this register may contain the address P of a procedure segment's descriptor.

I-counter this register may contain the index i of an instruction.

The processor features operators which redefine the contents of either the I-counter or the P-register or both. The pair [P-register, I-counter] constitutes an effective instruction pointer. If, as a result of program execution, the value of this instruction pointer is redefined, we say that a transfer of control has occurred; if the P-register has been redefined, then the transfer of control is either an inter-procedure call, or an inter-procedure return... or more simply a "call" or "return".

The computational operation (also known as "instruction execution") consists of the following sequential steps:

step 1: fetch the instruction pointed to by the [P-register, I-counter] instruction pointer.

step 2: perform the operation specified by operand f on the operands specified by the address list (a_1, ..., a_n). One of the effects of the performance of operation f may be a transfer of control.

step 3: if no transfer of control occurred during step 2, then increment the value of the I-counter by one.

step 4: resume step 1.

A processor performing a sequence of computational operations within a given procedure segment P (i.e., none of the computational operations is an inter-procedure call) is said to be "executing procedure P"; another figure of speech is to say that "procedure P executes".

2.2.1 Attributes

Built into our processor is a set of operators (f_1, f_2, ..., f_i, ..., f_n) which is divided into subsets. A subset is identified by a predetermined integer value which we name attribute. For the sake of convenience, we shall associate selected attributes with symbolic names. Examples of attributes are: "read"- the subset of operators which only read information from storage, "write"- the subset of operators which only write into storage, "descriptor" - the subset of operators which manipulate descriptors, etc.

We expand the concept of "attribute" to include functional capabilities which are not directly identified with any one specific operator, but which the processor is capable of recognizing. Thus, we associate the "execute" attribute with the function of fetching instructions from storage (step 1 of the computational operation).

We shall define, as we go along, those attributes which are essential for the construction of our model.

2.2.2 Protection hardware

We now define a number of internal registers which we build into our processor in order to enable us to implement the desired scheme for protection and controlled sharing of private information. The purpose and function of these registers will be described later on in the paper.

A-stack this register, known as the "access stack", is a LIFO (last in, first out) list of access keys. We use notation K to denote an access key, and notation K[t] to denote the topmost (last in) access key in the list.

An access key is a structured variable. We shall use notation K.x to refer to element x of structure K. The access key is structured as follows:

K.access an access name

K.mask an array of binary indicators. A binary indicator, known as "attribute mask" is addressable by an integer i, its index. An attribute mask may assume either of the values TRUE or FALSE. By convention, there is a one-to-one correspondence between the elements of arrays D.attributes and K.mask.

R-stack this register, known as the "return stack", is a LIFO (last in, first out) list of return pointers. We use notation R to denote a return pointer, and notation R[t] to denote the topmost (last in) return pointer in the list.
A return pointer is a structured variable. We shall use notation R.x to refer to element x of structure R. A return pointer is structured as follows:

R.pointer an instruction pointer [P, i].

R.gate a binary indicator which may assume either of the values
TRUE or FALSE

P-sphere (or protection sphere) this register may contain an access name.

A-pointer (or argument pointer) this register may contain a descriptor address D.

2.2.3 Inter-procedure calls

A procedure is an ordered sequence of instructions, whose execution may only be meaningful if commenced at predetermined locations. We name an instruction with which execution of the procedure may be meaningfully started entry point. Likewise, the logic of a procedure may require that at some point of the execution a call be made to some other procedure, and that control eventually return to the instruction following the call. We name an instruction with which the execution of a procedure may be meaningfully resumed return point. Our system guarantees that transfers of control into procedures be effected only at approved entry - or return points.

Associated with every entry point is at least one entry pointer. An entry pointer is a descriptor whose "entrypoint" attribute is TRUE, and whose D.pointer contains the address [P, i] of the entry point. A call instruction (requesting the performance of an inter-procedure call) is executed only if it references, as its operand, an entry pointer.

A return point is dynamically defined through the execution of an inter-procedure call. Associated with a return point is a return pointer, as defined in sub-section 2.2.2.

Some entry and return points are also associated with the "gate" attribute. We shall define this attribute in section 4. We only mention it here in order to be able to describe, in the following subsection, the logic of the call and return instructions.

2.2.4 Operations on the protection hardware

Following is a description of some of the operations with which the protection hardware described in subsection 2.2.2 may be manipulated.

a) reset-stacks this operation resets both A-stack and R-stack to null lists.

b) setmask(newmask, oldmask) consists of the following sequential steps,
b.1: oldmask := K[t].mask
b.2: K[t].mask := newmask

c) call(D, A) where D is the address of an entry pointer and A is the address of a descriptor which points to an argument list. We use notation FAULT to indicate that the current computational operation is to be abandoned and that an error condition is to signalled. Notation (register) denotes "value of register". The call operation consists of the following sequential steps:
c.1: if D.attributes["entrypoint"] = FALSE then FAULT.
c.2: fabricate a virgin return pointer and append it to R-stack (i.e., "push" the return stack); this new return pointer becomes the current R[t]. Initialize it as follows:
R[t].pointer := [(P-register), (I-counter)+1]
R[t].gate := FALSE
c.3: if D.attributes["gate"] = TRUE then
c.3.1: fabricate a virgin access key and append it to A-stack (i.e., "push" the access stack); this new access key becomes the current K[t]. Initialize it as follows:
K[t].access := D.owner
set all attribute masks K[t].mask[i] to TRUE
c.3.2: R[t].gate := TRUE
c.4: (A-pointer) := A
c.5: [(P-register), (I-counter)] := D.pointer

d) return this operation consists of the following sequential steps,
d.1: [(P-register), (I-counter)] := R[t].pointer
d.2: if R[t].gate = TRUE then remove K[t] from A-stack (i.e., "pop" the access stack).
d.3: remove R[t] from R-stack (i.e., "pop" the return stack).

Additional operations exist which manipulate the processor's protection hardware. Among them are operations to access a call argument pointed to by the A-pointer register, and a privileged operation (associated with the "processor" attribute) to put an access name into the P-sphere register.

2.2.5 Additional processor registers

The processor features additional registers which we do not describe here because their purpose lies outside the scope of our present paper We wish, however, to briefly mention the processor's execution stack.

The execution stack is a segment consisting of an array of "stack frame" items. The processor contains an internal register, named E-stack which may contain a [D, i] stack pointer, where D is the address of the descriptor pointing to the stack segment, and i is the index of the current stack frame.

.3 THE MEMORY CONTROLLER

The memory controller is a device which controls all access to both data- and descriptor storage. Any request for storage access must be made to the memory controller in the form r(s, a, D {,i}) where:

- r is the request operator specifying "fetch", "read" or "write".
- s is the storage designator specifying either the descriptor or the data storage.
- a is the attribute of the currently-executing instruction (if r = "fetch" then a is undefined).
- D is the addressed descriptor.
- i is an optional index.

.3.1 The access control logic

A request is honored by the memory controller only if the requestor is capable of substantiating his right to make the request, by presenting proper credentials. We shall describe the memory controller's verification ("access control") logic. The access control logic is explained in sections 3 and 4.

The memory controller has access capabilities to the processor's registers. We use notation (P-register) to denote the address of the descriptor pointing to the currently-executing procedure. Notation FAULT indicates that the current request for access is to be denied and that an error condition is to be signalled. Boolean operation AND denotes intersection.

Whenever the memory controller addresses an element of either the descriptor or the data storage, it checks for invalid addressing (e.g., an illegal descriptor address, an out of bounds reference to an element of an array, etc.). The detection of an invalid address causes the memory controller to FAULT.
The access control logic consists of the following steps:

step 1: locate the descriptor addressed by D.
step 2: if s designates the descriptor storage then
  2.1: if (P-register).attributes["descriptor"] = TRUE then go to step 8.
  2.2: FAULT
step 3: if r designates a "fetch" request then
  3.1: if (P-register).attributes["execute"] = FALSE then FAULT.
  3.2: if (P-register).attributes["public"] = TRUE then go to step 8.
  3.3: if (P-register).access = K[t].access then go to step 8.
  3.4: FAULT.
step 4: if D.attributes["public"] = TRUE then go to step 8.
step 5: repeat this step for all access keys K of the A-stack; if D.access = K.access then
  5.1: if (D.attributes[a] AND K.mask[a]) = TRUE then go to step 8.
  5.2: FAULT.
step 6: if D.access = (P-sphere) then
  6.1: if D.attributes[a] = TRUE then go to step 8.
  6.2: FAULT
step 7: FAULT
step 8: access control logic satisfied. Perform request.

3. RESOURCE MANAGEMENT

We now present the operational framework which will enable us to positively protect any segment from unauthorized access.

We introduce the concept of access sphere, which is a set of segments and processors. An access sphere is identified by a distinct access name which is an integer. At any given time, an access sphere is defined by the collection of segments and processors which are currently associated with it; this collection is comprised of zero or more processors and zero or more segments which may be either procedure or data segments or both. We say that the segments and processors "belong to" or "reside in" the access sphere.

3.1 ACCESS CONTROL

A segment belongs to the access sphere whose access name is posted in element D.access of its descriptor. A processor belongs simultaneously to several access spheres. For the purpose of execution (i.e., instruction fetch) it belongs to the sphere whose access name is posted in element K[t].access (i.e., of the A-stack's topmost access key.) For the purpose of data access it belongs a) to the access spheres whose access names are posted in elements K.access of the A-stack, and/or b) to the access sphere whose access name is posted in the P-sphere register.

All access to segments must be validated by the memory controller. The memory controller allows a segment to be accessed (be it for execution, read or write) only if the segment and the accessing processor belong to the same access sphere, i.e., access to the segment is permitted only if both processor and segment can present matching access names.

The reader is invited, at this point, to re-examine the memory controller's access control logic (subsection 2.3.1), and in particular

step 3 which verifies that the executed procedure belongs to the access sphere whose name is posted in K[t].access,

step 5 which checks whether or not the accessed data segment belongs to one of the access spheres whose names are posted in an element K.access of the processor's A-stack,

step 6 which checks whether or not the accessed data segment belongs to the access sphere whose name is posted in register P-sphere, and

step 7 where access is refused and an error condition is signalled because the accessing processor and the accessed segment do not both belong to the same access sphere.

3.2 THE PROCESSOR SPHERE

Every processor is associated with a distinct access name, posted in its P-sphere register. This access name is unique among all other access names in the system. We say that every processor belongs to its own processor sphere. Any number of segments may also belong to the processor sphere. A processor sphere is comprised, by definition, of a single distinct processor, and zero or more segments which may only be data segments. An application of the processor sphere concept and mechanism is given in Section 4.2.4.

3.3 RESOURCE OWNERSHIP

One of the important features of our system is the ability to selectively group segments and processors into collections identified by distinct access names, or in other words, the ability to "allocate" segments and processors into access spheres. We associate the ability to allocate resources with the concept of "ownership". The owner of a resource need not be presumed to be a person; it is more general to associate the concept of ownership with that of an access sphere (it is, after all, always possible to establish a person's ownership by associating the person's identity with that of an appropriate access sphere).

A resource may be simultaneously owned by several access spheres. However, in order to be able to resolve the conflicts which may arise through simultaneous application of ownership rights, we define that ownership may only be delegated through the act of allocation, and that all owners of a resource must be hierarchically related in direct order of allocation. Thus, suppose that resource a belongs to sphere A, and suppose that A allocates a to B which allocates a to C. Spheres A, B and C are the owners of resource a; however, in case of conflict A outranks B which outranks C. The act of allocation is performed by dedicated system modules, protected within distinct access spheres. These modules, which we shall discuss further on in the paper, keep historical records of resource allocation to establish the ownership hierarchy. Needless to say, an access sphere may always exercise its ownership privileges to reclaim a resource which it had previously allocated to some other sphere (in our example above, sphere B may reclaim a from sphere C, in which case only A and B remain as a's owners).

3.3.1 Segment ownership

A segment may be owned by several hierarchically related access spheres. A segment is defined by its descriptor, whose element D.owner contains the access name of the sphere in whose behalf that descriptor was produced. As we shall see in section 4, a segment may be shared among (i.e., belong to) several access spheres. Thus a segment's associated D.access defines the access sphere to which the segment belongs for the purpose of data-storage access, while its associated D.owner defines one of the spheres which may request the segment's descriptor to be accessed, as described below.

The owner of a segment may perform the following control applications on the segment: allocation, retrieval (i.e., opposite of allocation), redefinition (i.e., an owner may request the production of any number of descriptors to redefine the segment or any subset thereof, thus effectively creating new segments), destruction, and sharing. Of course two descriptors that redefine a segment into subsets that overlap (in the sense of section 2.1.1) may not be allocated to different owners (although sharing of overlapped subsets in the sense to be described in section 4.2 would be permitted).

3.3.2 Descriptor ownership

Descriptors being the tool to enforce the access sphere protection scheme, they may not in turn be protected by that very mechanism. Therefore descriptors are accessible (i.e., "belong") to procedure segments whose descriptors feature the "descriptor" attribute.

Let us suppose that all the procedures in the system which feature this privileged attribute belong to a single access sphere which we shall name "segment-manager". It follows that all descriptors in the system must be fabricated within the segment-manager sphere. If we further suppose that the segment-manager safeguards its "monopoly" over descriptors by never granting the "descriptor" attribute to any other access sphere in the system, it follows that some access sphere A may have its descriptors manipulated (or new descriptors created) only by requesting that the segment-manager do it in A's behalf. Thus, the segment-manager is the

exclusive owner of all descriptors in the system.

The segment manager keeps the ownership history of all segments in the system, and hon-ors only descriptor-manipulation requests re-garding segments owned by the requesting sphere.

3.3.3 Processor ownership

The concept of processor ownership relates to the ability to reset the values of certain key registers in the processor. These registers are available to procedure segments whose des-criptors feature the privileged "processor" attribute.

Analogous to the segment-manager sphere, we define a "processor-manager" sphere which "owns" all the processors in the system. We may think of the processor-manager as owning a pool of available processors. An access sphere may re-quest at any time that a processor be allocated to it. The mechanics of processor allocation are rather simple (the reader may wish to con-sult subsection 2.2.4 "Operations on the protec-tion hardware") and may be described by the following three steps,

step 1: initialize various registers, in-cluding initialization of register P-sphere to a unique (among access names) value,

step 2: perform operation reset-stacks to reset both the A-stack and R-stack.

step 3: perform a call to a "gate" entry point in the requesting sphere.

The sphere may later return the processor to the processor-manager's pool simply by exe-uting a return instruction.

.4 PROTECTION FROM THE PROCESSOR, A REVIEW

Up to now we have described how our access ontrol scheme guarantees that a processor be llowed to access a segment only if it belongs n the segment's access sphere. It is however mportant to be able to further restrict the ex-cution powers of a processor relative to a iven segment; in fact, it is desirable to allow processor to access a segment only to the ex-ent actually required by the purpose of the egment. This approach makes it possible to inimize the penalty for error. Thus, it is in-dvisable to allow a processor to write into a egment whose purpose does not require that it e written into; if an erroneous procedure exe-uting within that segment's access sphere ever ttempts to alter the contents of the segment, he hardware will not allow it to do so.

The apparent advantages of this approach re twofold, namely a) the damaging potential f an erroneous procedure is restricted, and) the probability for speedy error detection is increased.

We achieve this protection by associating every segment with the set of attributes (i.e., subsets of the processor's operator repertoire) corresponding to the purpose of the segment. For example, a segment which is accessed only for the purpose of instruction fetches (we name such a segment "pure"- or "non-selfmodifying procedure") requires only the "execute" attri-bute; an "impure"- or "selfmodifying procedure" requires the "execute", "write" and perhaps even the "read" attributes.

Our machine is even potentially more re-strictive (for reasons associated with controll-ed sharing, discussed in section 4) in that a segment's attributes may sometimes be "masked off" by the attribute masks K.mask of the appro-priate access key. The reader may wish to re-consult steps 2, 3, 4, 5 and 6 of the memory controller's access control logic (subsection 2.3) and observe that access to storage is per-mitted only in accordance with predetermined attributes.

4. PROTECTION AND CONTROLLED SHARING

We have seen how resources may be select-ively grouped into distinct access spheres. We shall now show how this grouping may be used to permit a segment owned by a certain sphere to be accessed, in a controlled fashion, by (or in behalf of) some other access sphere. We shall show that our model allows an owner of a segment to precisely specify the exact degree of access to be granted to any other sphere.

To illustrate, suppose that access sphere A owns a segment V which it is willing to make accessible to spheres B, C, D and E; we shall use notation V.x to denote element (or subset) x of segment V. Owner A may decide to:

- grant B complete and uninhibited access to segment V,
- grant C read-only access to subset V.s of segment V,
- grant D access to variable V.v in seg-ment V, but only for the purpose of computing V.v := f(V.v),
- grant E complete and uninhibited access to a single bit V.b in segment V,
- etc. etc.

As may be gleaned from the example above, the number of possibilities for controlled sharing is potentially very large. To attempt to justify the design of our system by citing numerous examples would be a futile approach. Instead, we shall attempt to formally define the requirements for controlled sharing, and show how our machine fulfills these require-ments. The interested reader is encouraged to

put our machine to the test by formulating and resolving his own examples. A convenient way for doing so might be to make use of graphic representation, drawing objects within circles to denote access spheres and their contained segments and processors, having the "access sphere" circles overlap where appropriate, and using arrow notations to represent a processor's flow of control.

4.1 REQUIREMENTS FOR CONTROLLED SHARING

In the following discussion, we shall make use of the term "lender" to refer to an access sphere which makes one of its owned segments available to some other sphere, which we shall refer to as "borrower". We use the term "sharing" to describe the act by which a lender gives a borrower access rights to one of the lender's owned segments; we say that such a segment is a "shared segment", and that it belongs (i.e., for the purpose of access) to both the lender and the borrower access spheres simultaneously. We wish to emphasize once more, that only an owner of a segment may share that segment with one or more borrowers, and that the act of sharing does not bestow upon the borrower any ownership privileges relative to the shared segment.

The requirements for controlled sharing may be stated in the following words: "the ability of the lender to specify to the system in what precise manner an authorized borrower may access a shared segment, and the guaranteed protection, by the system, of a segment from unauthorized access."

4.1.1 Access authorization

In a computer system, an authorization to access an item of information must be comprised of three parameters, namely a) the identity of the information item to be accessed, b) the identity of the borrower, and c) the definition of the computations to which the borrower may subject the information item.

An access authorization is any desired intersection of three sets (α AND β AND γ) where:

- α is the set of all distinctly addressable information items in the system,
- β is the set of all entities which may access an information item, and
- γ is the set of all possible computations which the system is potentially capable of performing on an information item αi.

4.2 CONTROLLED SHARING ON OUR MODEL MACHINE

The aforementioned three sets are represented on our model system as follows:

- α The concept of segmentation as defined in this paper makes it possible to give any group of contiguous bits of information a distinct "segment" identity. The concept of the access sphere allows us to define any desired set of segments, i.e., represent any group of contiguous or non-contiguous bits of information in a way which is comprehensible to and enforced by the system. Thus, in our model system, set α is the set of all access spheres.

- β As mentioned before, our system allows a person to be identified with an access sphere. This in turn provides the means for positively identifying the procedures authored by that person. Permission for a procedure to access an object in the set α is then directly related (as it should be) to a permission for the author of that segment. Thus, in our model system, set β is also the set of all access spheres.

- γ is the the set of all computations to which a segment may be subjected. Our system features a positive access control logic, i.e., all access to a segment is a priori forbidden, unless specifically authorized. Therefore, we need concern ourselves only with the set of computations which the owner of the segment specifically approved of, which consists of the following two subsets:

  - $\gamma 1$ the set of all computations which the processor itself is capable of recognizing, and which is represented by the set of attributes, and

  - $\gamma 2$ the set of computations that are too complex for the processor to recognize, and which is the set of procedures approved by the owner of the segment.

  For the purpose of this development therefore, we define $\gamma = \gamma 1$ AND $\gamma 2$.

The introduction of the procedure as element of subset $\gamma 2$ introduces an added dimension of complexity into our scheme for controlled sharing, because the procedure is in itself a segment to be protected. We are clearly confronted with a recursive problem which may not be solved in a linear fashion.

Our ideal machine allows us, in principle, to specify any particular access from the inter-

section α AND β AND γ, provided that γ is understood to be further specified in terms γ1 AND γ2. We shall now demonstrate, in four progressive stages, that the model as described in this paper is capable of supporting generalized controlled sharing of information.

stage 1, protection: shows that in our system, an owner X may positively protect any set of information items α from being accessed by any procedure in the system except for a set of authorized procedures γ2 which operate within the scope of attributes γ1, by allocating segments α and γ2 to a private access sphere β, i.e., specify (α AND β AND γ1 AND γ2).

stage 2, direct sharing: shows how a lender X may grant any procedure γ2 which belongs to a borrower β access to segments α within a specified scope of attributes γ1, i.e., specify (α AND β AND γ1 AND γ2).

note: stages 3 and 4 which follow treat two subcases for α, where we assume that α = (α1 OR α2), OR denoting inclusion.

stage 3, indirect sharing: shows that a lender X may grant a borrower β indirect access to segments α1 through the execution of an approved set of "caretaker" procedures γ2 within a predetermined scope of attributes γ1, i.e., specify (α1 AND β AND γ1 AND γ2).

stage 4, argument sharing: shows that, in addition to the indirect sharing described in stage 3, borrower β may in turn lend arguments α2 to X's procedures γ2 for the purpose of accomplishing the indirect sharing, and that these arguments must not be made generally available to borrower X. This stage allows us to specify ((α1 OR α2) AND β AND γ1 AND γ2).

The four stages just described cover all possible needs for the protection and controlled sharing of information in a computer system. Our model system allows any desired combination of the four stages to be specified.

4.2.1 Protection

Suppose that owner X wishes to protect segment α, and make it accessible only to procedure γ2 within the scope of attributes γ1. Owner X allocates segment α and procedure γ2 into a new access sphere β. The only descriptor in the system pointing to segment α looks as follows:

α.access = β
α.owner = X
α.attributes = γ1

The only descriptor in the system pointing to procedure γ2 looks as follows:

γ2.access = β
γ2.owner = X
γ2.attributes["execute"] = TRUE

We further suppose that a) no other descriptor in the system features D.access = β, and b) that the owner has requested the processor-manager to allocate a new processor into access sphere β (see subsection 3.3.3), which implies that the processor's registers are initialized as follows:

R-stack contains a return pointer pointing to the processor-manager sphere, and need not concern us here.
A-stack contains a single access key K[t], initialized as follows:
K[t].access = β
K[t].mask contains an all-TRUE attribute mask.
P-sphere contains a unique access name

We may now easily show that under the conditions just stated only procedure γ2 may ever access segment α, and that only within the scope of attributes γ1. We consult the memory controller's access logic as described in subsection 2.3.1. The processor is capable of fetching instructions only from procedures which belong to the access sphere whose name is posted in K[t].access. The only procedure which fulfills this requirement is procedure γ2. Furthermore, the processor may only access a data segment which belongs to a sphere whose name is posted either a) in an access key K, of which we have only one, or b) in register P-sphere which contains a unique access name which is guaranteed to be posted nowhere else in the system. Also, the processor may access the segment only for the purpose of an operation which lies within the segment's declared scope of attributes.

4.2.2 Direct sharing

Suppose that lender X wishes to give borrower β access to segment α. Any procedure γ2 belonging to borrower β may access α, provided that the access be within segment α's declared scope of attributes γ1. Lender X calls on the segment-manager to produce a new descriptor D which looks as follows:

D.pointer = α.pointer i.e., segment D is in fact a redefinition of segment α
D.access = β i.e., segment D belongs in access sphere β

D.owner = X but is still indisputably owned by X which possesses the exclusive right to delete descriptor D whenever he so wishes
D.dtype = α.dtype or perhaps any other data type that X chooses to specify
D.specifier = α.specifier or perhaps any other specifier of X's choice
D.attributes = γ1

The reader is invited to convince himself that only borrower β may access "incarnation" D of segment α.

The sharing just described applies to both data and procedure segments. However, a procedure may only be indirectly entered through an entry pointer, so the lender must provide the borrower with an entry pointer as well. Thus, in order for X to share procedure α with β, X must request the segment-manager to produce two descriptors, namely a) descriptor P (describing segment α) and b) descriptor Q (pointing to an approved entry point in α), as follows:

P.pointer = α.pointer
P.owner = X
P.access = β
P.attributes["execute"] = TRUE

and,

Q.pointer = [P, i] where i is the approved entry point
Q.owner = X
Q.access = β
Q.attributes["entrypoint"] = TRUE

The reader is invited to convince himself that a) borrower β may access procedure P only at entry point Q, b) procedure P belongs in access sphere β and may access any data or procedure segments in sphere β, and c) borrower β may not misuse procedure P in order to access segments which do not belong to sphere β.

4.2.3 Indirect sharing

The only way in which lender X may enforce that borrower β will only access segment α1 through the intermediary of procedure γ2 is by not sharing α1 and γ2 with sphere β as described in stage-2, but rather by allowing a processor from sphere β to execute γ2 in sphere X.

By definition, a processor may only execute in the access sphere whose access name is posted in K[t].access. Our machine permits intersphere calls (i.e., transfers of control among procedures belonging to different access spheres) by exercising its ability to "push" the access stack and append a new K[t] which allows the processor to execute in the called sphere. An inter-sphere call is recognized by the machine when an attempt is made to transfer control through an entry pointer featuring the "gate" attribute. The access stack defines the inter-sphere call history, which may be unwound by performing an appropriate series of returns.

Lender X grants borrower β access to α1 through the execution of γ2 by having the segment-manager produce a gate entry pointer Q as follows:

Q.pointer = [γ2, i] where i is the approved entry point
Q.access = β
Q.owner = X
Q.attributes["entrypoint"] = TRUE
Q.attributes["gate"] = TRUE

Assuming that β's processor is initialized as described in subsection 4.2.1, after it had performed a call to entry point Q (the logic of the call instruction is described in subsection 2.2.4), its access stack contains two access keys, as follows

1. topmost access key where K[t].access = X
2. "pushed down" access key where K.access = β

4.2.4 Argument sharing

We have seen how a processor belonging to sphere β may execute an inter-sphere call into access sphere X, there to execute procedure γ2 and through the execution of that procedure indirectly access segment α1. Normally, however, such a call also involves the passing of arguments to the called procedure, which in turn implies that procedure γ2 which belongs to sphere X must be able to access arguments belonging to sphere β. This in effect means that procedure γ2 becomes the borrower of the arguments. Our system allows sphere β to establish a normal lender/borrower relationship with procedure γ2, disregarding the fact that it simultaneously maintains a borrower/lender relationship with sphere X.

Our machine features two mechanisms with which a borrower procedure may be granted access to the lender's arguments a) the access stack mechanism which is cheap to use but which grants the borrower access to all of the segments in the lender's sphere [this mechanism implies that the caller sphere (lender) trusts the called procedure (borrower) and does not wish to bother and protect itself], and b) the processor sphere mechanism which is a regular direct sharing mechanism (section 4.2.2) involving the production of special descriptors. [this mechanism is more expensive to use than the access stack mechanism, but it provides the lender with the ability to restrict the borrower's access to the arguments only, and that within their declared scope of attributes].

For the following discussion, suppose that
is a descriptor pointing to an argument list
hich for our purpose is an integer array of size
1), that closedmask is an array of FALSE attri-
ute masks, and that storemask is a segment in
hich an array of masks may be stored. We ex-
end our indirect sharing as described in subsec-
ion 4.2.3 to include the passing of argument
2 to procedure γ2.

Let us first consider the case of access
tack sharing. Descriptor α2 belongs to sphere
(i.e., α2.access = β). Sphere β calls entry
oint Q as follows:

a) puts the integer α2 into W(1)
b) executes call(Q, W)

s mentioned before, the processor which now ex-
cutes in procedure γ2 has two access keys in its
ccess stack, featuring access names X and β,
nd therefore procedure γ2 is capable of address-
ng both α1 and α2; it is, however, equally cap-
ble of accessing any other segment which be-
ongs to sphere β.

In the case of processor sphere sharing,
he lender first puts the arguments into a spe-
ial access sphere, then calls the borrower in
uch a manner that the borrower has access to
he arguments but not to the lender's original
phere. Arguments are allocated into the pro-
essor sphere because it is guaranteed to be in-
ccessible to any other processor in the system.
rocessor sphere sharing is done as follows:

a) β requests the segment manager to make a new descriptor E which redefines α2 as follows,
   E.pointer = α2.pointer
   E.access = (P-sphere)
   E.owner = β
   E.attributes = whatever β specifies
b) it also requests the segment-manager to allocate W into the processor sphere (i.e., W.access = (P-sphere)), and puts integer E into W(1).
c) it disables the access stack sharing mechanism by executing setmask(closed mask, storemask)
d) it calls sphere X by executing call(Q, W)
e) (upon return from α2), it restores the old attribute mask by executing setmask(storemask, closedmask)
f) it asks the segment-manager to destroy descriptor E

third possibility exists, namely that the call-
r wishes to deny the called sphere all access
ltogether in the caller's sphere, in which case
he calling sequence is,

a) setmask(closedmask, storemask)
b) call(Q, 0) where 0 ("zero") may indicate absence of arguments
c) setmask(storemask, closedmask)

Any combination of the two mechanisms just described is also possible. For example, the caller may grant the called sphere restricted "read only" access to all of the caller's segments, by performing an appropriate setmask, and in addition grant the called sphere more liberal access rights to a number of selected arguments by using the processor sphere mechanism.

4.3 Public access

It is essential to the construction of any multiple-user computer system that certain segments be made accessible to all users, and that some of these segments be generally known by a predetermined segment address D. Examples of such segments are the gate entry pointers to the "segment-manager" and "processor-manager" access spheres, which have to be known by predetermined segment addresses. We provide for this requirement by incorporating into the access control logic the concept of the "public" attribute, which specifies that access to this segment is granted to any access sphere in the system. Thus a segment with the "public" attribute is said to belong to all access spheres in the system. The reader may wish to review steps 3 and 4 of the access control logic (subsection 2.3.1) which check for the "public" attribute.

4.4 Explicit and implicit sharing

Three sharing mechanisms have been presented in the foregoing discussions: a) direct- and indirect sharing by explicitly granting access to the borrower sphere through the fabrication of special descriptors, b) implicit argument sharing by using the A-stack mechanism, and c) argument sharing by using the P-sphere mechanism. Of these, mechanism (a) which provides explicit sharing, and mechanism (c) may be regarded as essential while mechanism (b) can be considered to be, in some sense, a fanciful extension of (a).

Mechanisms (b) and (c) which provide implicit sharing are functionally distinct from mechanism (a). Explicit access is granted to an access sphere which is known in advance; lender X grants borrower β access to segment α by giving β a descriptor D which redefines α and whose D.access = β. Implicit access, however, is granted to a function whose execution may extend over any number of a priori unknown access spheres. Suppose that sphere A calls sphere B passing B an argument α. It is unknown whether B will actually process α, or whether B will call C which will call ....which will call Z

which is actually the sphere which manipulates α. Moreover, argument α was made accessible to the called sphere for the express purpose of participating in the function for which the inter-sphere call was made. Argument α must not be made generally available to the called sphere (if it were, explicit sharing would have been used). Our system does not make any assumptions about the number of processors which may, at any given time, concurrently execute within any given access sphere. Argument sharing must be accomplished so that they may not be accessed by other processors which may be executing within the called sphere at the time of the inter-sphere call. The mechanisms for implicit sharing fulfill the requirements for argument sharing by associating the access rights to the arguments with the calling processor, rather than with the called sphere.

5. COMMUNICATION AMONG ACCESS SPHERES

We have seen that our model system supports controlled sharing of information by allowing a segment to be redefined to reflect the different access privileges associated with different borrowers. Thus segment α belonging to lender X is known as segment (D1, D2, ..., Di, ..., Dn) to borrowers (β1, β2, ..., βi, ..., βn) respectively. This means that a shared segment is known to different spheres under different names. A lender X who wishes to share segment α with borrower β by providing β with a special descriptor D must thus have a facility for communicating to β the segment number D, which is dynamically defined and hence a priori unknown to β. The only way in which information may be communicated among access spheres is through a shared segment in which β may retrieve the communication posted by X. Thus, information sharing among access spheres presents a recursive problem which has to be resolved through the application of some system wide convention by which a shared segment is accessible and known to all access spheres in the system by its predetermined descriptor address. It is the "public" attribute which enables us to incorporate this capability in our system.

We are, moreover, faced with a synchronization problem. Namely, access spheres include processors which are capable of independent and asynchronous execution. Some method must be devised by which segments could be shared among access spheres without restricting the spheres' potential for asynchronous execution.

And thirdly, a naming scheme must be provided by which independent access spheres may all meaningfully refer to a single shared segment, even though in actuality the segment is known to each access sphere by a unique address D which is not meaningful to any other access sphere. This problem is a fundamental one and in our view can only be solved by an agreement between the sharers made outside the system. (This necessary agreement has been recognized in a previous paper [14] where, in the context of inter process communication through shared data bases it was termed "ipc-setup".)

5.1 SEGMENT CATALOG

To solve the problems just outlined, we introduce the concept of a segment catalog which is a data base maintained by the segment-manager. The segment catalog is an array of entries E, where each entry describes a shared segment. A catalog entry is a structured variable; we shall use notation E.x to refer to element x of structure E. The catalog entry is structured as follows:

- E.names which is a list of symbolic segment names, where a symbolic segment name is a unique (within the catalog) character string. We use notation E[i] to refer to catalog entry E featuring symbolic name i.
- E.address is the storage address [S, n] of the shared segment.
- E.owner is the access name of the sphere which created this catalog entry.
- E.accesslist is a list of access tags T. An access tag is a structured variable as follows:
  - T.access is the access name of a sphere which shares this segment
  - T.dtype is the data type under which the segment is known to this sphere
  - T.specifier is the segment's specifier.
  - T.attributes this sphere's set of attributes

5.2 EXPLICIT SHARING BY SYMBOLIC NAME

An owner of a segment, say sphere X, may now share one of its segments with sphere β by associating a unique symbolic name with that segment, say "symb", and by requesting the segment-manager to create a catalog entry for this segment. It further on requests the segment-manager to put an access tag T[β] in segment "symb"s catalog entry, thus authorizing the segment manager to grant sphere β access to segment "symb".

Sphere β may now, at any time, call the segment-manager requesting a descriptor pointing to segment "symb". It may do so by performing

call segment-manager("symb", D)

where D is a return argument in which the segment-manager returns to the caller the integer D which is the address of the requested descriptor

tor.

The segment-manager looks up the catalog until it finds the entry which features the symbolic name "symb". It then locates, within that entry, the access tag T[β] which features sphere β's access name, and produces the requested descriptor D as follows:

D.pointer = E["symb"].address
D.dtype = T[β].dtype
D.specifier = T[β].specifier
D.access = β
D.owner = E["symb"].owner
D.attributes = T[β].attributes

6. CONCLUSION

We would like to add a number of remarks, commenting on the model and suggesting some applications for it.

6.1 The reader interested in the feasability of actually building an ideal processor as defined in this paper may wish to acquaint himself with the Burroughs' B6500/B7500 computer [11] which is in many ways similar to our machine. It features a hardware-implemented stack mechanism suitable for the implementation of our R-stack and E-stack registers. As for the implementation of our A-stack register, an associative memory would most probably be indicated.

6.2 The ideal memory could be implemented in the form of a paged virtual memory. Bensoussan,[10], describes in great detail and clarity the implementation of such a virtual memory.

6.3 Our system does not differentiate between "system" and "user" code; both are implemented by using the same access-sphere protection mechanism. This allows flexibility in system development and expansion possibilities; new "supervisor" routines may be coded and tested in an unprivileged "user" environment. Also, the system may be expanded by layers of subsystems which play "supervisor" roles relative to their respective "users".

6.4 There is no extra execution overhead associated with an inter-sphere call; in effect, the cost of a call is the same for both intra- and inter-sphere calls. This may encourage programmers to compartmentalize the modular components of their programs for the sake of modularity as well as for maximum protection against programming errors.

6.5 The problem of establishing meaningful naming conventions and structures for segments is an entirely open subject which is well worth exploring. Access spheres may maintain internal naming structures for private segments as well as external ones for communicating with other access spheres. A most interesting possibility is that of implementing hardware-supported ALGOL nested-declaration block structures, where each block is implemented as an independent access sphere.

6.6 A subject which would be equally fascinating to explore is that of multiprocessed computations, and the possibility for implementing tree-structured hierarchies of computations(processes). PL/1 tasking, for example, seems to be easily implementable on our model; for example, the fork statement corresponds to a call to the processor-manager requesting it to allocate another processor to the requesting computation.

6.7 No mention was made in this paper to input/output (I/O) devices. Our system is well suited for the implementation of (simulated logical I/O devices within dedicated access spheres. Such an approach would allow the system to hide all instances of I/O device management behind a facade of logical devices and, by allocating a logical device to a user's computation upon demand (even though such a device need not necessarily map into a corresponding actual device at the time of allocation) it resolves the problems of hardware resource multiplexing by queueing up requests for (possibly) delayed service.

7. ACKNOWLEDGEMENT

This paper is an outgrowth of three years work and study of operating systems centered on Multics. We are indebted to Professors Corbato, Saltzer, Graham an d others of M. I. T.'s Multics design team for this opportunity and for countless hours of teaching and counsel in improving our understanding of operating systems. Special thanks is also due Andre Bensoussan of Cambridge Information Systems Laboratory (the G. E. Component of the Multics effort) whose special help and encouragement was instrumental in the development of this paper.

8. REFERENCES

1. Dennis, J.B., "Programming Generality, Parallelism and Computer Architecture", M.I.T. Project MAC, MAC-M-409, 1968
2. Fano, R.M., "The MAC System: The Computer Utility Approach", IEEE Spectrum 2, Jan. 1965, pp 56-64
3. Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations", Comm. A.C.M., 9, 3 March 1966 ,

pp. 143-155.
4. Wilkes, M.V., "Time Sharing Computer Systems" American Elsevier Press, New York, 1966
5. Vanderbilt, D.H., "Controlled Information Sharing in a Computer Utility", M.I.T. Project, MAC TR-67, 24 October, 1969
6. Evans, D.C., and Leclerc, J.Y., Proc. AFIPS 1967 Spring Joint Computer Conference, Vol. 30, Thompson Books, Washington, D.C. pp 23-30.
7. Corbato, F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System", Proc. AFIPS 1965 Fall Joint Computer Conference, Vol 27, Part 1, Spartan Books, N.Y., pp 185-196.
8. Daley, R.C., and Newumann, P.G., "A General Purpose File System for Secondary Storage" Proc. AFIPS 1965 Fall Joint Computer Conference, Vol 27, Part 1, Spartan Books, N.Y., pp 213-229.
9. Graham, R.M., "Protection in an Information Processing Utility", Comm. ACM 11, 5 May 1968, pp 365-369.
10. Bensoussan, A., Clinger, C.T., and Daley, R.C., "The Multics Virtual Memory", Proc. Second ACM Symposium on Operating Systems Principles, October 20-22, 1969, Princeton University, pp 30-42, based on a report by Bensoussan et al., of the same title. Internal document G0093, The General Electric Co., Cambridge, Information Systems Laboratory (CISL) Technology Square, Cambridge, Mass.
11. Hauck, E.A., and Dent, B.A., "Burroughs B6500/ 7500 Stack Mechanism", Proc. AFIPS 1968 Spring Joint Computer Conference, Thompson Book Co., Washington, D.C. pp. 245- 251
12. Cleary, J.G., "Process Handling on Burroughs B6500", Proc. Fourth Australian Computer Conference, Adelaide, South Australia, August, 1969, pp 231-239
13. Spier, M.J., and Organick, E.I., "Modelling A Computer System Utility - A Summary", Fourth Annual Princeton Conference on Information Sciences and Systems, March 26-27, 1970
14. Spier, M.J., and Organick, E.I., "The Multics Interprocess Communication Facility", Proc. Second ACM Symposium on Operating System Principles", October 20-22, 1969, Princeton University, pp 83-91.

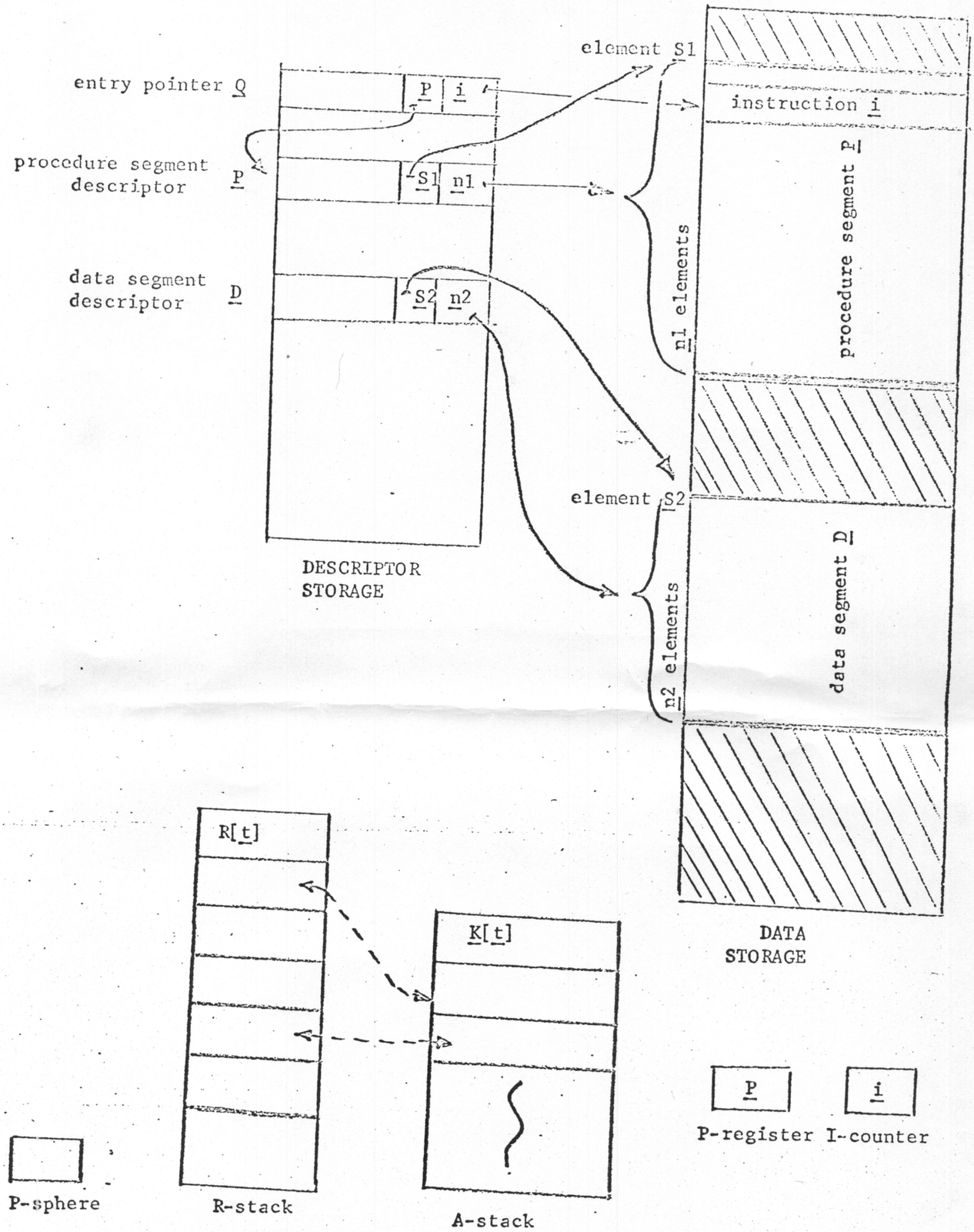

Figure 1: The essential components of the ideal machine