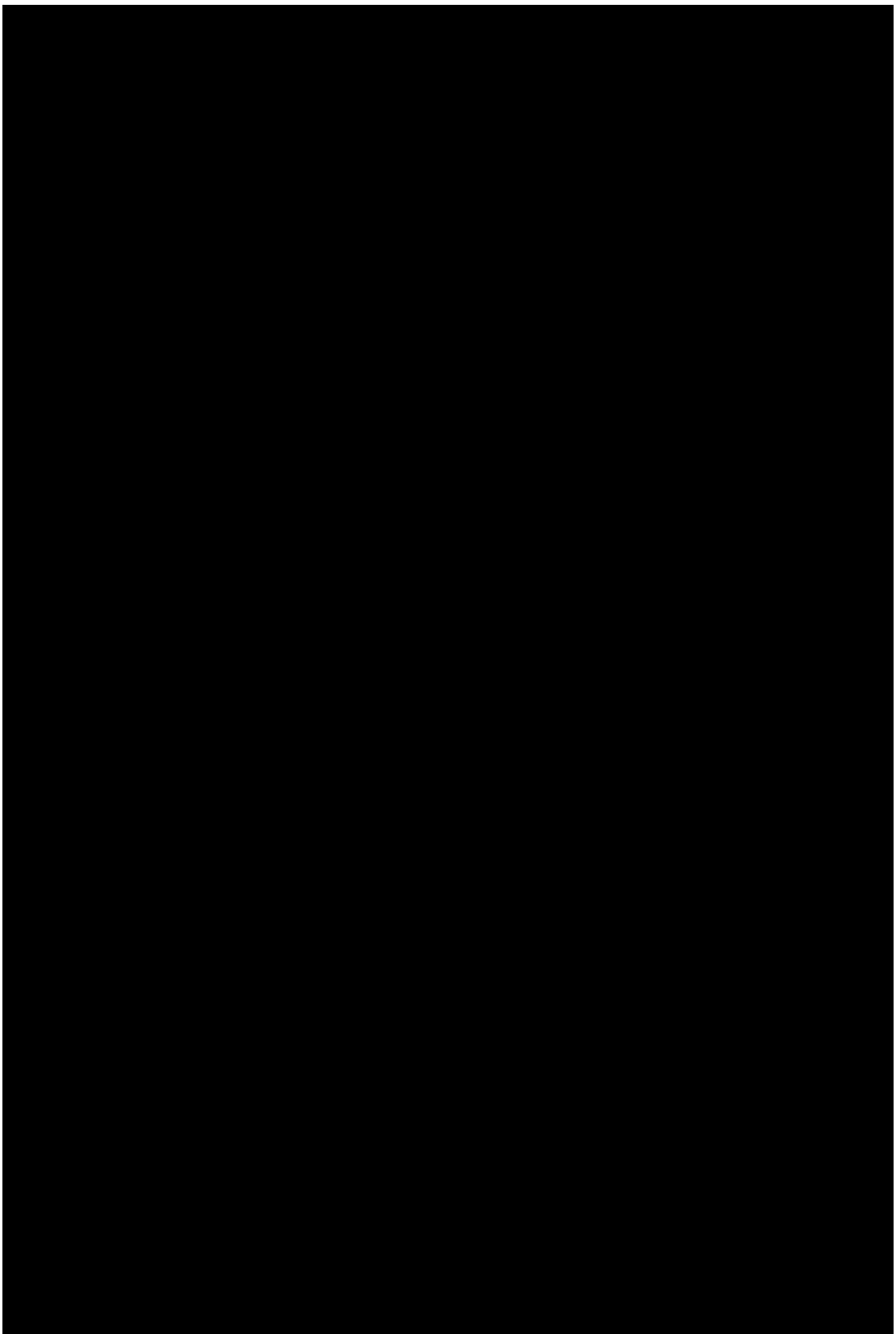

This blank page was inserted to preserve pagination.

MAC-TR-23

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

PROGRAMMING SEMANTICS FOR MULTIPROGRAMMED COMPUTATIONS

by

Jack B. Dennis and Earl C. Van Horn

December 1965

This paper was presented at the Association For Computing Machinery Conference on Programming Languages and Pragmatics, San Dimas, California, August 8-12, 1965

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

ABSTRACT

The semantics are defined for a number of meta-instructions which perform operations essential to the writing of programs in multiprogrammed computer systems. These meta-instructions relate to parallel processing, protection of separate computations, program debugging, and the sharing among users of memory segments and other computing objects, the names of which are hierarchically structured. The language sophistication contemplated is midway between an assembly language and an advanced algebraic language.

TABLE OF CONTENTS

Section		Page
ABSTRACT		
LIST OF ILLUSTRATIONS		
I	INTRODUCTION	1
II	PROPERTIES OF MULTIPROGRAMMED COMPUTER SYSTEMS	3
III	CONCEPTS AND TERMINOLOGY	5
	Segments	5
	Protection	5
	Processes	7
	Computations	8
	Principals	8
IV	THE SUPERVISOR	11
	The Process List	11
	Allocation and Scheduling	11
	Accounting and Control	12
V	PARALLEL PROGRAMMING	13
	Basic Primitive Operations	13
	Lockout	15
	An Example	17
	Input/Output	18
	Motivation for Parallelism	19
VI	SPHERES OF PROTECTION	21
	Inferior Spheres	21
	Exceptional Conditions	23
VII	PROTECTED ENTRY POINTS	27
VIII	DIRECTORIES AND NAMING	31
	Sharing of Retaining Objects	31
	Desiderata for Names	31
	Ambiguous Names	32
	False Names	33
	Preview	33
	Directories	34
	Ownership	35

TABLE OF CONTENTS (Cont.)

Section		Page
	Use of Directory Structure	35
	Creation and Deletion of Retained Objects	36
	The Structure of Names	38
	Sharing Mechanisms	38
	Using a Programming System	41
REFERENCES		44

LIST OF ILLUSTRATIONS

Figure		Page
1	A Computation	9
2	The **join** Procedure Step	14
3	**Lock** and **unlock** Meta-Instructions	16
4	Control of an Inferior Computation	22
5	Entry to and Exit from a Protected Procedure	29
6	A Directory Structure	39
7	Using a Programming System	42

I. INTRODUCTION

An increasing percentage of computation activity will be carried out by multiprogrammed computer systems. Such systems are characterized by the application of computation resources (processing capacity, main memory, file storage, peripheral equipment) to many separate but concurrently operating computations.

We can cite three quite different examples of multiprogrammed computer systems to illustrate their diversity of application. The American Airlines SABRE passenger record system couples ticketing agents at dispersed offices to a central data file [1]. The computer support systems of NASA provide real time control and monitoring of manned space flights [2]. The Project MAC time-sharing system permits research workers closer interaction with the powers of automatic computation [3]. Although these are all on-line systems, multiprogramming techniques have also been used successfully in systems that perform computations on an off-line, job-shop basis.

We will review some of the distinctive properties of a multiprogrammed computer system (MCS), and then introduce some concepts and terminology that have proven useful in studying the properties of multiprogrammed computations. As we proceed, we will define a number of meta-instructions that embody powers mostly absent from contemporary programming languages, but essential to the implementation of computation processes in an MCS. These powers relate to 1) parallel processing; 2) naming objects of computation; and 3) protection of computing entities from unauthorized access. The character of these meta-instructions is such that they might form part of a language intermediate in sophistication between an assembly language and an advanced algebraic language for an MCS. In fact, the semantics of these meta-instructions could be incorporated in the definition of an intermediate language that might be employed at some stage in the translation of a more advanced language.

We do not claim completeness for the set of meta-instructions [illegible]. Additional operations will prove necessary in practice for a specific MCS. In particular, no means is discussed whereby an object computation may advise the supervisor of special scheduling or allocation requirements. Also, conventions for dynamic control of segment length have been omitted.

II. PROPERTIES OF MULTIPROGRAMMED COMPUTER SYSTEMS

Five properties of multiprogrammed computer systems are important to the present discussion.

1) Computation processes are in concurrent operation for more than one user.

A Multiprogrammed Computer System is generally the creation of many individuals working in part toward a common objective and in part for private goals. A successful MCS must include mechanism for preventing undesired interference among computations.

2) Many computations share pools of resources in a flexible way.

In consequence, the individual planner of a computation need not be concerned about efficiently using a certain fixed amount of memory and processing capacity which would otherwise go to waste. Resources not used by one computation are available to other concurrent computations.

3) Individual computations vary widely in their demands for computing resources in the course of time.

An MCS must have mechanisms (explicit or implicit) through which a computation may request and release resources according to need. Where many computations are active which are not closely coupled in their demands for resources, the peak demands of some computations will coincide with the slack demands of others. As the number of computations in the system is increased, the instantaneous total demand for resources will hover closer to the sum of the individual average demands. Therefore, the amount of physical resources required in such an MCS is governed by the average demand over all computations rather than by the sum of their peak demands.

4) Reference to common information by separate computations is a frequent occurrence.

In an MCS it is advantageous to allow information to be common among computations proceeding for different users to avoid needless duplication of procedures and data. Also, communication among separately

planned computations is essential to many MCS objectives. Furthermore, the sharing of a peripheral device by several computations is sometimes required.

5) An MCS must evolve to meet changing requirements. An MCS does not exist in a static environment. Changing objectives, increased demand for use, added functions, improved algorithms and new technologies all call for flexible evolution of the system, both as a configuration of equipment and as a collection of programs.

To meet the requirements of flexibility of capacity and of reliability, the most natural form of an MCS is as a modular multiprocessor system arranged so that processors, memory modules and file storage units may be added, removed or replaced in accordance with changing requirements [4].

III. CONCEPTS AND TERMINOLOGY

SEGMENTS The smallest unit of stored information that is of interest in the present discussion is called a word. An ordered set of words grouped together for purposes of naming is called a segment. A segment is created at some point in time and has a definite length (which may vary with time) at any instant of its existence.

Any reference by a computation to data or procedure information is specified by a word name

$$w = [\ i, a\ ]$$

consisting of the index number i of the segment containing the desired word, and a word address a giving the position of the word within the segment. The index number may be thought of as an abbreviation for the name of the segment. The correspondence between an index number and a name is established by meta-instructions which will be defined subsequently.

In the programming examples (which are written in a pseudo-Algol format) variable identifiers, array identifiers and labels will stand for word names. We will write word names as $[\ i, a\ ]$ only when the index number must be explicitly mentioned.

The concept of segment has influenced the design of a commercial computer (the Burroughs B5500), an experimental machine [5], and one military system (the Burroughs B825). The use of segments in software systems is discussed by Greenfield [6], Holt [7] and others. The design of addressing mechanisms for MCS's is discussed by Dennis [8]. A fuller implementation of these concepts in a machine organization has been discussed by Glaser, Couleur and Oliver [9], and interesting work in a similar direction is in progress at the M. I. T. Lincoln Laboratory [10], IBM [11], and is continuing at Burroughs [12].

PROTECTION In an MCS, a computation must be denied access to memory words and other objects of computation unless access is authorized. In particular, it seems natural to implement memory

protection on a segment basis. Thus, we think of a computation as proceeding within some sphere of protection [13] specified by a list of capabilities or C-list for short. Each capability in a C-list locates by means of a pointer* some computing object, and indicates the actions that the computation may perform with respect to that object. Among these capabilities there are usually several segment capabilities, which designate segments that may be referenced by the computation and also give, by means of access indicators, and indication of the kind of reference permitted.

X	executable as procedure including internal read references for constants.
R	readable as data but not executable.
XR	executable as procedure and readable as data.
RW	readable and writable as data.
XRW	executable as procedure and readable and writable as data.

Other types of capability are also permitted in the C-list of a computation, and will be introduced as appropriate in the discussion. Every capability contains an ownership indicator (O for owned, N for not owned) Computations have broad powers with respect to owned computing objects through mechanisms to be described. In the case of an owned segment, for example, a computation may delete the segment, and grant or deny other computations access to the segment.

During the execution of a computation, capabilities will frequently be added to and deleted from the C-list defining its sphere of protection

* We use the term "pointer" here because of its familiarity to most workers. The permanent representation of a pointer should not be a hardware address in the machine (main or auxilary storage) as it is essential that the entire naming structure be independent of physical device addresses if reallocation of storage media is to be feasible. The authors suggest the association of a unique code (called an effective name in ref. 13) with each computing entity (segment, directory, etc.) which is assigned at the time the entity is created.

through the use of meta-instructions to be described in later sections. The linear subscript of a capability within a C-list is called its index number. It is through the use of the index number that the capability is exercised by processes. For example, a segment is referenced by giving the index number of the segment in a word name. We assume that the allocation of these index numbers is carried out by the system (i.e., the supervisor program) during the execution of an object computation.

PROCESSES We consider that the system hardware comprises one or more processors, which we can identify as being distinct from the main memory, the file storage devices and the input/output devices. Each processor is capable of executing algorithms that are specified by sequences of instructions. A process is a locus of control within an instruction sequence. That is, a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor.

In a physical computer system a process is represented by the information that must be loaded into a processor in order to continue execution of the successive instructions encountered by the process. We call this set of information the state word of the process, and note that it must not only contain the accumulator words, index words, and the word name of the next instruction to be executed, but must also indicate the C-list applicable to the computation to which the process belongs.

A process is said to be running if its state word is contained in a processor which is running. A process is called ready if it could be placed in execution by a processor if one were free. Running and ready processes are said to be active. A process that is not active is suspended, and is awaiting activation by an external event, such as the completion of an i/o function.

COMPUTATIONS Loosely speaking, a computation may be thought of as a set of processes that are all working together harmoniously on the same problem or job. More precisely, we define a computation to be a set of processes having a common C-list such that all processes using that same C-list are members of the same computation.

Notice that two processes having separate C-lists are always members of separate computations, even though these C-lists might describe the same set of capabilities. Notice also that there exist one-to-one correspondences among computations, spheres of protection, and C-lists; each computation operates within the restrictions of a unique sphere of protection that is specified by a unique C-list. The relationship among these entities is shown schematically in Fig. 1.

PRINCIPALS The ordinary notion of a user of an MCS is of an individual who requests computing service from an MCS, or who interacts with a time-shared MCS from a console. We generalize this notion by defining the term principal to mean an individual or group of individuals to whom charges are made for the expenditure of system resources. In particular a principal is charged for resources consumed by computations running on his behalf. A principal is also charged for retention in the system of a set of computing entities called retained objects, which may be program and data segments, for example. The structure and identification of these retained objects is discussed in a later paragraph.

We can clarify our notion of a principal by giving some examples. Each individual user of the MAC time-sharing system acts as a principal since he is able to utilize system resources to achieve any personal goal, and is restricted only by an accounting of his expenditure of basic resources. He may create, modify, and delete segments of procedures and data solely according to his personal objectives. In the MAC system we also find principals consisting of groups of individuals. Such a group principal might be responsible for the maintenance of a system of

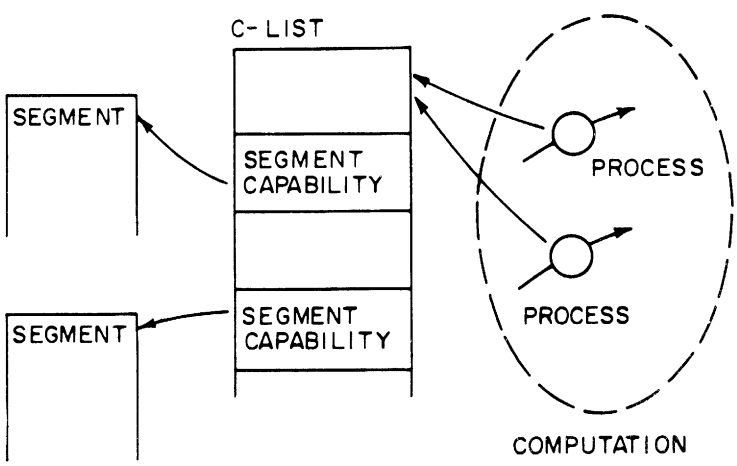

Figure 1. A Computation

procedures that solves a certain class of mathematical problems (e.g., matrix operations or statistical analysis). Another group principal might have cognizance over a programming language system including editing routines, compiling routines and debugging aids. Still a third principal might oversee the common procedures of an extensive design project involving the cooperative effort of many people.

In the case of an airline information processing system, the agents do not participate as principals, but simply communicate with a set of procedures that enable them to perform well-defined interrogations of and operations on a centrally-stored data base. In such a system, a principal might consist of a team of system planners and programmers responsible for the success of a single aspect of the system's mission. Examples of such separate aspects are passenger records, aircraft scheduling, and accounting.

In the case of computer support for a manned space flight, separate principals could be responsible for different aspects of the mission -- guidance during propulsion, tracking while in orbit, orbital computation, medical data processing, etc.

IV. THE SUPERVISOR

We use the term supervisor to denote the combination of hardware and software elements that together implement a core of basic computer system functions around which all computations performed by the system are constructed. For present purposes we suppose that the core of functions includes mechanisms for

1) allocation and scheduling of computing resources.
2) accounting for and controlling the use of computing resources.
3) implementing the meta-instructions.

We do not inquire in the present paper as to the internal workings of the supervisor required to perform the above functions. Instead it is our aim to point out the essential features of the interface between the supervisor and user processes which operate in lower spheres of protection. However, it is helpful to think in more concrete terms about how the supervisor accomplishes some of its functions.

THE PROCESS LIST Specifically, let the process list be a data structure within the supervisor, with an entry for each process existing in the system. Entries are created in and removed from this list by various meta-instructions and by other mechanisms that will be described. Each entry can hold the state word of its corresponding process, as well as accounting and scheduling information. As mentioned before, each process is either running, ready, or suspended.

ALLOCATION AND SCHEDULING At any time segments of information will be distributed among a hierachy of storage devices (core, drum, disk, and tape, for example) with that information most relevant to the on-going computation processes located in the more accessible media. With each computation there is associated a set of information to which it requires a high density (in time) of effective reference. The membership of this working set of information varies dynamically during the course of the computation. The supervisor's problem is to decide how information (segments) should be distributed in the storage hierachy and how the queue of active processes should be disciplined to make most effective use of system resources in accomplishing the MCS mission.

ACCOUNTING AND CONTROL We suppose the charges for the expenditure of computation resources associated with the execution of a process are assigned to the principal that was responsible for the creation of the process. We also assume that each principal is given an allotment of resources, and that appropriate action is taken by the supervisor if this allotment is exceeded.

V. PARALLEL PROGRAMMING

BASIC PRIMITIVE OPERATIONS The basic primitive operation of parallel programming is implemented by the meta-instruction

fork w;

as suggested by Conway [14] where w is a word name. A **fork** meta-instruction initiates a new process at the instruction labelled w. The newly created branch process is part of the same computation as its creator or main process, that is, it is associated with the same C-list. A process that has completed a sequence of procedure steps is terminated by the meta-instruction

quit;

after which the process no longer exists and its state word is discarded from the process list. A set of primitives for parallel programming must include a mechanism whereby one process may be continued just when all of a certain set of processes have completed. All that is required is a procedure step that will decrement a count and test for zero. We use the instruction

join t, w;

which is essentially Conway's join instruction. Here t is the word name of the count to be decremented and w is the word name of an instruction word to be executed if the count becomes zero as indicated in Fig. 2. It is essential that the three references to the count t not be separated in time by references to t from other processes. This requirement is indicated by the dashed box in the figure and is readily achieved in practice by combining the two actions into one machine instruction that is completed with a single reference to the count word.

In describing algorithms involving parallel processes, it is convenient to declare certain quantities as private to a process. For this purpose the declaration

private x;

means that the quantity named x is to exist only so long as the process executing the declaration exists; that is, private data is lost when a process quits. At a **fork** the values of any quantities declared private to the main process are assigned as values of corresponding

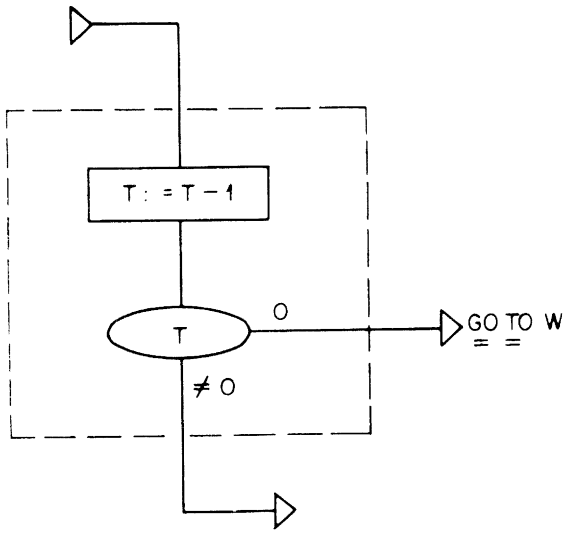

Figure 2. The **join** Procedure Step

quantities of the branch process. In practice, the state word of a process is the natural representation of private data. If there is more data declared private than can be represented in the state word, the system must create a segment for private data which is copied at each **fork** and lost upon reaching a **quit**.

LOCKOUT A provision whereby two processes may negotiate access to common data is a necessary feature of an MCS. Suppose a certain data object (which might be a word, an array, a list structure, a portion or all of a segment) may be updated asynchronously by several processes, which are perhaps members of different computations. Updating a data structure frequently requires a sequence of operations such that intermediate states of the data are inconsistent and would lead to erroneous computation if interpreted by another process.

The lockout feature proposed here presumes that all computations requiring access to the data object are well behaved. If it is desired to protect the data object from destructive manipulation by an untrustworthy computation, routines with protected entry points as described later in this paper must be employed.

We associate with the data object a one-bit lock indicator that is accessible to all processes requiring use of the data object. Two meta-instructions are introduced that operate on the lock indicator w.

lock w;

The effect of the **lock** meta-instruction is given in Fig. 3a. The lock bit is set to one just when the data object has been found unlocked by all other processes. Again, as indicated by the dashed box, the two references to w must not be separated by references to w from other processes. The meta-instruction

unlock w;

resets the lock indicator to zero as in Fig. 3b.

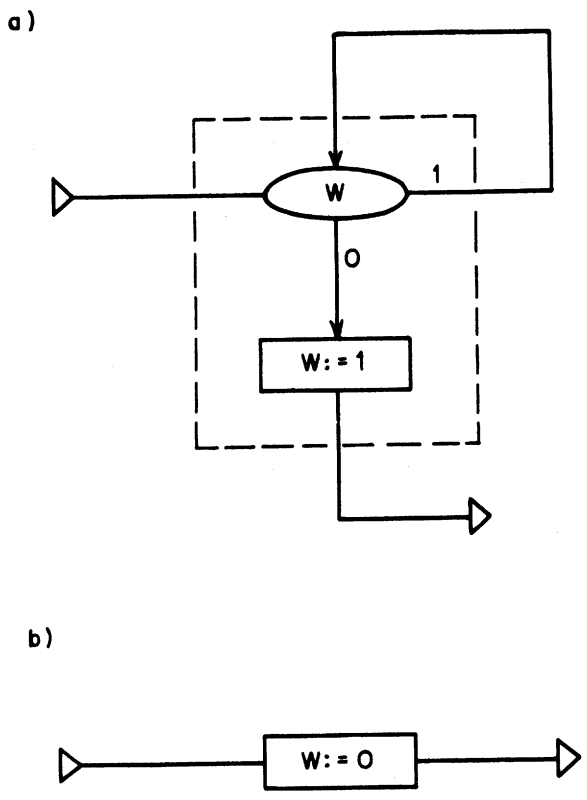

Figure 3. **Lock** and **unlock** Meta-Instructions

The use of these meta-instructions w[illegible] ing.

```
lock w;
- - -  }
- - -  }  update a sequence for data object associated
       }  with lock indicator  w .
- - -  }
unlock w;
```

In practice the execution time of a typical update sequence is quite small and the chance that a process will hang up on a lock instruction will be very low. However, a process may be removed from execution if a processor is preempted by a higher-priority computation. Thus, a data object could remain locked for a substantial time if such preemption occurred between a **lock/unlock** pair. Then hangup of other processes interrogating that lock indicator could be highly probable. A solution to this problem is to inhibit interruption of a process between execution of a **lock** and execution of the following **unlock**. Of course, this requires that a time limit be set on the separation of **lock/unlock** pairs.

AN EXAMPLE An elementary example of parallel programming that illustrates the use of these meta-instructions is the following program that evaluates that dot product of two vectors A and B .

```
begin    real array A[1:m] , B[1:n] ;
         Boolean w;   real S;   integer t;
         private integer i;
         t := n;
         for i := 1   step 1 until i > n  do
                      fork e;

         quit;
```

```
        e:              begin real X;
        substance:              X := A[i] x B[i];
                                lock w;
                                S := S + X;
                                unlock w;
                                join t, r;
                                quit;
                        end;
        r:
    end;
```

Obviously, this computation is too trivial for parallel programming to be of practical interest. If the algorithm expressed by the statement labelled "substance", instead of being a simple multiplication, involved the operation of a large, complex system of procedures (e.g., the compilation of a segment of procedure), the notation of parallel processing as used above would allow several instances of that algorithm to be in simultaneous execution, thus more effectively utilizing the presence of its procedure information in main memory.

INPUT/OUTPUT A basic power of computations in an MCS is the ability to communicate with peripheral (input/output) devices. Two classes of communication have evolved in terms of implementation in present day computer systems. In the simpler class a process requests the transmission of a unit of information (word or fraction of a word) to or from a peripheral device and waits in suspended status until the information is transmitted before continuing. (A processor, as contrasted with the process, may be executing other processes during the wait interval, however.) This form of implementation is appropriate for low data-rate situations, and also where a close interaction between the computation and the peripheral devices is required (e.g., quick response to brief inquiries from a remote console).

In the second form of input/output operation, a sequence of interactions between memory (i.e., a segment) and the peripheral device occurs in response to an initiation signal from a process. The process remains suspended until all interactions between memory and the peripheral device have been completed.

In either case a principal characteristic of the input/output operation is the elapse of time between initiation and completion. This input/output wait is generally long compared with the instruction execution time of a typical central processing unit. For our purposes we will not distinguish further between these two forms of input/output operations, and will call both by the term i/o function.

Since peripheral devices are part of the physical resources of a computer system, the use of i/o functions must be restricted to computations authorized to do so. It is natural to consider an i/o function as representing another class of capability that may be entered in the C-list that defines a sphere of protection. This capability is then exercised by the meta-instruction

execute i/o function i;

where i is the index number of an i/o function capability in the C-list of the computation. Performance of this procedure step by a process causes initiation of the i/o function represented by the ith entry of the C-list. The process then becomes suspended and remains so until the i/o function has completed. It then becomes active again to perform subsequent procedure steps.

Particular stress has recently been placed on ability to specify computations that may compute in parallel with input/output operations. Within the scheme presented here, this goal is easily achieved through the execution of **fork** meta-instructions prior to the execution of i/o functions.

MOTIVATION FOR PARALLELISM The motivation for encouraging the use of parallelism in a computation is not so much to make a particular computation run more efficiently as it is to relax constraints

on the order in which parts of a computation are carried out. A multi-program scheduling algorithm should then be able to take advantage of this extra freedom to allocate system resources with greater efficiency.

Moreover, the notation of parallel programming is a natural way of expressing certain frequently occurring operations of computations running in an MCS. Suppose, for example, we wish to program a computation to receive messages from any of a number of user consoles, where the messages are to arrive in some unknown and arbitrary order, and it is not known whether some consoles will ever send messages. Let listen(i, j) be an **integer procedure** that waits for a message to be received from console i and writes the message in the segment with index number j. The value of listen is set to the number of symbols in the message. Let analyze(i, j, n) be a **procedure** which scans a message of n symbols received from console i and written in segment j , and takes whatever action is necessary in response to the content of the message. Then the message-receiving computation described above may be programmed as follows.

```
begin private integer i;
      for i := 1 step i until i > n do
            fork e;
      quit;
e:    begin integer j, n;
            j := create segment RW;
            n := listen(i, j);
            analyze(i, j, n);
            quit;
      end;
end;
```

The **create segment** meta-instruction introduces a segment capability into the C-list of a computation and is discussed in a following section.

VI. SPHERES OF PROTECTION

It is useful to think of a computation's sphere of protection as having been established by another computation, that is, by the action of a process operating within another sphere of protection. A major reason for taking this view concerns the debugging of programs in some programming language system (PLS). However, other uses of this concept are also possible.

In connection with program testing (debugging), suppose that the processes of a PLS are carried out, as for any object computation, within some sphere of protection A. These processes must have access to all of the user's computing objects pertinent to the program under test, as well as to the procedure segments of the PLS. Since the program under test is likely to be faulty, it is desirable to protect both the user's permanent objects, and any objects created by the PLS on his behalf from unintentional use or destruction by the procedure being debugged.

INFERIOR SPHERES To allow the processes under test to be operated within a sphere of protection distinct from the one effective for the PLS, we define several meta-instructions.

i : = create sphere w; — Append an owned inferior sphere capability to the C-list with index number i. The word name w is the return point for exceptional conditions, as explained later.

The process executing this meta-instruction operates in a sphere we call the superior of the created sphere. Once in possession of an inferior sphere capability (Fig. 4), a process may grant some of its capabilities to the inferior sphere by the following meta-instruction.

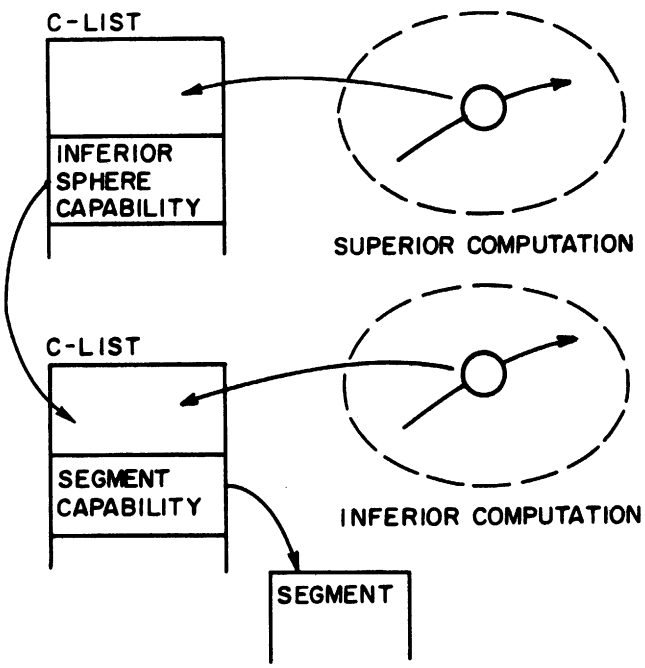

Figure 4. Control of an Inferior Computation

i : = grant $\left\{\begin{matrix}\lambda \\ O\end{matrix}\right\}$ $\left\{\begin{matrix}\lambda \\ X \\ R \\ XR \\ RW \\ XRW\end{matrix}\right\}$ j, k;

Grant capability j to inferior sphere k with index number i. Here j and k are index numbers in the current C-list, and i is an index number in the inferior C-list.

The granted capability is entered in the C-list of inferior sphere k and may be a segment capability, i/o function capability, entry capability, or directory capability. Entry and directory capabilities are discussed in later paragraphs. The braces mean that one of the strings within them must be selected to form part of the meta-instruction. Here λ stands for the null string. The string O indicates that the inferior sphere is to have ownership powers with respect to the granted capability. The other strings can be used only if j is the index number of a segment capability. In this case the capability is passed down with restricted access authority. For example,

i : = grant X j, k;

grants authority to execute the segment but not to read it, write it, or exercise ownership of it. The grant meta-instruction cannot be used to pass a capability that is not implied by a capability present in the higher sphere.

start i, w; Initiate a process at instruction word name w within inferior sphere i.

The new process commences with no private data, that is, a zero state word except for the instruction word name w.

EXCEPTIONAL CONDITIONS Next we ask what should happen if a process operating in an inferior sphere encounters an exceptional condition, that is, a procedure step requiring intervention by a higher level before the object process may continue in a sensible manner.

Some exceptional conditions call for action by the supervisor. These include:

1) Fault. A fault is a clear indication of hardware malfunction. A memory parity error is a good example. The supervisor is responsible for correct operation of processor and memory units.

2) Resource excess. A resource excess occurs if a process invokes resources in an amount exceeding the allotment to the principal responsible for its computation.

3) Addressing snag. An addressing snag occurs when a process generates a valid address, but the desired information is either not in main memory or a reference mechanism has not been set up. The supervisor must move the desired information into main memory from file storage and set up the necessary linkage.

Other exceptional conditions should be acted upon by the superior computation of the process in trouble, since only the procedures which established the process know how these conditions should be interpreted. These exceptional conditions are:

1) Sphere violation. A sphere violation occurs if a process refers to a capability that does not exist in the C-list of its computation, or makes invalid use of a capability (attempts to write in a segment for which only the execution capability is authorized, for example). A sphere violation also takes place if a reference is made beyond the limits of a segment.

2) Halt instruction. A halt means "terminate this process and notify superior" as contrasted with quit which means "terminate this process and forget it."

3) Breakpoint instruction. A breakpoint is substituted for other instructions by a debugging program in order to conduct a

breakpoint analysis of a program under test. A **breakpoint** has the same effect as **halt** except that a different indication is presented to the superior procedure.

4) Undefined instruction. A processor generates this condition when it is called upon to execute an undefined operation code.

5) Arithmetic contingencies. Such events as "divide check" call for action by a superior procedure when not explicitly handled by the inferior computation.

In any of these events, the process in which the exceptional condition occurred becomes suspended, and a new process is initiated in the superior sphere at the instruction word specified when the inferior sphere was created. The new process starts with two pieces of private data: a number indicating the reason for the interruption, and an index number of an owned suspended process capability that is appended to the C-list of the superior sphere at the time of interruption. This capability allows the superior computation to have access to the state word of the process in which the exceptional condition occurred. The following meta-instructions are defined with respect to a suspended process capability.

fetch status i, w;	Fetch the state word of suspended process i and write at word name w.
set status i, w;	Set the state word of suspended process i according to information at word name w.
continue i;	Reactivate suspended process i and delete from the C-list.

Notice that the **set status** meta-instruction must disallow a change in certain critical parts of the state word of the suspended process. For example, the superior sphere must not be able to cause the state word of the suspended process to point to a different C-list.

A debugging procedure needs primitive commands which allow it to "pick up the pieces" after a computation under test has malfunctioned. The following meta-instructions are useful under these circumstances.

stop k; Suspend all processes operating in inferior sphere k .

Execution of this meta-instruction causes each active process in inferior sphere k to be suspended. Corresponding to each inferior process a suspended process capability is created in the C-list of the superior sphere. Also, a process in the superior sphere is initiated to correspond to each inferior process, just as though the inferior process had encountered an exceptional condition.

Capability j in the C-list of inferior sphere i can be examined by the meta-instruction

examine i, j, w;

The information contained in the capability is copied into several words starting at word name w .

If the inferior computation has clogged its C-list with unneeded capabilities, the superior computation can remove them with

ungrant i, j;

which erases capability j from the C-list of inferior sphere i .

VII. PROTECTED ENTRY POINTS

An important class of situations arises when a peripheral device is operated or a data object is manipulated on behalf of several concurrent computations. Examples of this situation are:

1) A control routine for transferring messages between user computations and remote terminals of a given class.

Frequently, a system of remote terminals is coupled to a central processing system through a single i/o function (rather than one per terminal device).

2) A routine which updates a data base and may be called asynchronously by many separate user computations.

The planning of such a routine* requires that calling computations be protected from each other. If A and B are two computations using the routine S, it must not be possible for a malfunction of A's processes to cause incorrect execution of B's procedures. Clearly, neither A nor B should be able to modify the common data D used by S. Furthermore, A and B must be forced to initiate operation of S at a proper entry point, for erroneous transfer of control to an arbitrary instruction of S is likely to cause meaningless modification of the common data D. However, if D is to be written by S, then the processes executing S must have in their C-lists the capability to write in segment D as well as the capability to execute any instruction of S.

It follows that a modification or change of C-list must accompany transfer of control to S. A mechanism for accomplishing such restricted use of a procedure we call a protected entry point.

The mechanism we describe supposes that a process calling the protected procedure executes it in a distinct sphere of protection R, returning to the original sphere of protection A upon completion. The change of association of process with C-list implied here is

* Introduced as a "protected service routine" in ref. 4.

accomplished by the **enter** meta-instruction which requires an additional capability, the entry. An entry capability is created by the owner of a protected procedure through the use of the meta-instruction

h : = **create entry** w, n;

where h is the index number in the creator's C-list of the created capability. Here w is the word name [i, a] , and i must be the index number in the creator's C-list of an owned procedure segment. The entry capability thus created authorizes calls to be made to the word names [i, a] through [i, a+ n] inclusive. Also included in the entry capability is a pointer to the C-list of the creating computation. Once created, the entry capability can be copied into the C-lists of other computations, using mechanisms to be described.

The entry to and exit from a protected procedure is depicted schematically in Fig. 5. To enter a protected procedure a process gives

enter j, r, k;

where j is the index number of an entry capability. The calling process is suspended, and a new process is created. The C-list of this new process will be the C-list specified by the entry, with the addition of two new capabilities. One is a suspended process capability pointing to the state word of the calling process, and the other is a duplicate of the capability having index k in the caller's C-list. The index numbers of these capabilities are reported as private data in the state word of the new process. The new process is set to begin execution at word name [i, a+ r] , where i and a are quantities specified in the entry, as mentioned above. Notice that i is an index number with respect to the new C-list, not that of the caller, and also that r must satisfy

$0 \leq r \leq n$

where n is also specified in the entry. The remainder of the new state word is set equal to the corresponding parts of the caller's suspended state word. Finally the new process is made active. The protected procedure thus given control can use the **fetch status, set status,** and **continue** meta-instructions to communicate with the

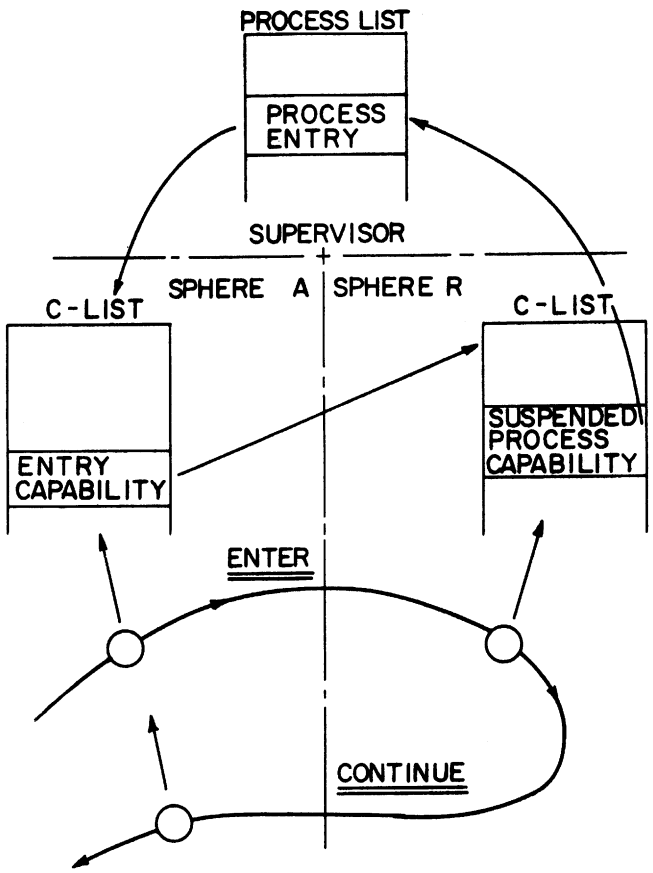

Figure 5. Entry to and Exit from a Protected Procedure

caller and reactivate its calling process whenever this is appropriate.

The capability transmitted to the protected computation (represented by index "x" above) can not only be a segment capability, i/o function capability or entry capability, but can also be a directory capability. As will be described in the next section, a directory consists of a collection of capabilities. Thus the **enter** meta-instruction provides a quite general, yet reasonably efficient, facility for passing to the protected procedure the capabilities that it needs to perform its service for the caller.

VIII. DIRECTORIES AND NAMING

Until now, we have been discussing those aspects of an MCS that deal with the active performance of computing tasks for the benefit of the system's users. Now consider the fact that in most MCS's, even if no active computing is taking place, each principal of the system is still represented passively in the system by a set of retained objects. Every retained object is either a segment, an i/o function, an entry, or a directory. Here we are letting the segment play a role which has been ascribed to something called a file in many MCS's, particularly in the MAC system. In the present formulation, a file is simply a long-lived segment.

SHARING OF RETAINED OBJECTS The possibility of rapidly and automatically controlling the sharing among principals of retained objects, chiefly procedure and data segments, is one of the main characteristics that distinguishes the MCS from other types of computing systems [3]. The importance of sharing is testified to by the fact that the file manipulating machinery of the MAC system has recently undergone a major revision, motivated in part by a desire to facilitate such sharing [15].

Besides being useful to individual users who wish to borrow each other's routines, a sharing mechanism is also useful to a group of users who wish to reference certain segments in common. Such segments might be a set of library routines, or a set of procedures making up a programming language system. It is natural to think of these segments as being owned by a principal associated with the group of users as a whole. A mechanism (such as the one to be described) is required for permitting an individual user to gain access to the directory of the group principal.

DESIDERATA FOR NAMES Through the capabilities in their C-lists, computations can, among other things, manipulate retained objects. In performing these manipulations, the processes of a computation must specify information that unambiguously

distinguishes each object of interest from all other retained objects in the computing system. Such information constitutes the name of the object.

Retained objects are created and deleted arbitrarily, and any particular object may remain in existence for an arbitrarily long time. There are two reasons why the name of an object can never be changed by the system throughout the object's entire existence. First, if a name is changed, then all usages of that name that are imbedded in other objects (e.g. segments) within the system must be updated. This alternative may be dismissed as being entirely impractical in a large MCS. The second reason why the system must leave all names unchanged is that every retained object is frequently referred to directly by people. People are used to thinking in terms of invariant names; to find that yesterday's "X" is suddenly today's "Y" would be disconcerting.

Another requirement which human usage places on the names of objects is that they should be alphanumeric and have mnemonic significance. Each principal should be able to choose freely the names by which he will identify the objects he retains, without regard to the choices of names made by other principals.

AMBIGUOUS NAMES If the names of two different objects have been freely chosen by two different principals, those names may possibly be identical. When this common string of characters is generated subsequently by a process, the computer system will not be able to determine which of the objects is being designated. Such a string of characters is said to form an ambiguous name.

The problem of ambiguous names also manifests itself in more traditional, non-multiprogrammed computing environments when groups of independently written subprograms are to be combined into one large program. One author has called for "an orderly corpus of symbology" designed to prevent name conflicts before they occur [16]. Others have offered a solution based on the loading-time definition of each subprogram's symbolic interface with its environment [17].

The most straightforward way of eliminating the possibility of name ambiguities within an MCS is to restrict each principal in his choice of names; a principal can be required to begin every one of the names of his objects with a string of characters that constitutes his principal name. The remainder of the name of an object, its chosen name, may then be freely selected by the principal retaining the object. This method of preventing name conflicts has been employed in the MAC time-sharing system [18].

FALSE NAMES In order to conserve storage, it is reasonable to embed within a procedure segment only the chosen names of the objects being referenced, with the understanding that the computer system can supply the principal name because it knows which principal initiated the process that is executing the procedure segment. Even if a principal has a complex program consisting of many procedure segments, each containing references to the others, the above scheme still insures that when the author principal operates the program the system will always supply the correct principal name to augment the chosen names embedded within the segments.

A serious problem arises, however, if this program is shared with a second principal and this principal attempts to execute the program. Intersegment references will evoke the name of the second principal, rather than that of the author. The names thus formed will be false names, because they will designate objects that are very different from those intended by the author. Such names will often designate no existing object at all, but occassionally they may designate objects of the second principal that are unrelated to the borrowed program.

PREVIEW The problem arises of simultaneously realizing the following four goals: (1) to avoid the creation of ambiguous names, (2) to provide reasonable freedom for a principal to choose some portion of the names of his objects, (3) to allow intersegment references to consist of parts of names rather than full names, and (4) to permit sets of objects to be shared without invalidating internal references.

The solution we propose stipulates that each reference to an object be derived from a partial name relative to some directory of objects, together with the index number of a capability pointing to that directory. Moreover, we allow the directories of the system to be organized into a hierarchical structure, as suggested by Daley and Neumann [19].

This approach has two major advantages:

(1) A whole subhierarchy of objects can be communicated among several computations or principals by passing a single pointer to the head directory of the subhierarchy.

(2) It is easy to design the MCS so that programs can be shared without the possibility that false names will be generated by their execution.

In the following paragraphs we define the proposed naming structure and introduce the meta-instructions necessary for computing within its framework.

DIRECTORIES A directory is a set of items, each being an association between a name component and a capability which points to a segment, i/o function, entry or another directory. Recall that each capability includes an ownership indicator (**O** for owned, **N** for not owned), and that a segment capability includes an indication (**R, W, X** or a combination) of the type of reference permitted. Each item of a directory also contains an access indicator (**P** for private, **F** for free). The interpretation of these indicators in directories is explained below.

Associated with each principal is exactly one directory called a root directory, which stands at the head of a hierarchy of the principal's retained objects. We allow perhaps many items to point to the same object, and in consequence, an object may be accessible through the directory structure from different root directories.

OWNERSHIP A principal always owns his root directory. Otherwise, an object is owned by a principal just if that principal owns a directory in which there exists an item with an O indicator that points to the object. Thus, a principal owns an object if and only if there is a path through the directory tree from his own root directory to the object such that each node of the path contains an O indicator.

When the supervisor creates a computation on behalf of a principal, it always places in the C-list of such a computation a directory capability with an O indicator that points to the principal's root directory. The principal is then said to own this computation and each of its processes. These processes are then permitted to exercise powers of ownership with respect to objects owned by the principal.

USING THE DIRECTORY STRUCTURE The powers of a computation with respect to the directory structure are embodied in meta-instructions as follows. We suppose that any process has at least one entry in its C-list giving it a directory capability.

$$j := \text{acquire} \left\{ \begin{array}{l} \lambda \\ \mathbf{X} \\ \mathbf{R} \\ \mathbf{XR} \\ \mathbf{RW} \\ \mathbf{XRW} \end{array} \right\} i, < \text{name component} >$$

Here i is the index number of a directory capability. This directory is searched for an association with < name component >, the corresponding capability is entered into the C-list of the computation to which the running process belongs, and its index number is reported as j. Capability j is tagged O if and only if directory i is tagged O in the C-list, and the capability being loaded is tagged O in directory i. A sphere violation results if the capability

referenced is tagged P in the directory item and directory capability i is not owned (i.e. contains an N indicator). In the case of a segment, the type of reference permitted may be changed from that permitted in the directory item, but an attempt to enlarge the class of reference permitted to a non owned segment is also deemed a sphere violation.

release i;

Remove the capability with index number i from the C-list of the running process.

Ownership of an object implies the ability to modify it, delete it, and grant access to the object by other principals.

place { P / F } i, < name component>, j;

Here i must be the index number of an owned directory capability. An item is inserted in directory i associating the capability having index number j with < name component>.

remove i, < name component >

The item associated with < name component > in owned directory i is removed from the directory.

CREATION AND DELETION OF RETAINED OBJECTS Segments, entries, and directories can come into existence upon execution of the following meta-instructions.

i := create { segment { X / R / XR / RW / XRW } / entry w, n; / directory }

A capability pointing to the created object is entered into the C-list of the process with an O indicator, and its index number is reported as i. Note that a name is not associated with the object at the time of its creation, but only when an entry is made for it in some directory by means of a **place** meta-instruction.

This illustrates the point that names are a convenience for principals. Different names may be convenient for different principals, and no name need be assigned unless a principal may need to select that object from the directory structure at a later time. Thus, for example, segments may be created by computations for temporary storage purposes without affecting the directory structure.

The owner of a segment, entry, or directory can cause it to cease to exist by using the following meta-instruction.

delete i, < name component >;

The owned object pointed to by the capability associated with < name component > in directory i is deleted so that it has no further existence. Any attempts to exercise capabilities pointing to a deleted object are treated as sphere violations.

The **release** and **remove** meta-instructions differ from **delete** in that the former meta-instructions simply remove capabilities from C-lists and items from directories, respectively, while the object itself continues its existence if there are other capabilities and items pointing to it.

We suppose then that the existence of a segment, entry, or directory extends from its time of creation until either specifically **delete**'ed by its owner or until **release**'ed from all C-lists and **remove**'ed from all directories. This convention yields the possibility of having a retained object with no owner. This seems quite reasonable because the following situation may occur frequently. An obsolete subroutine segment S is **remove**'ed from the directories of a library principal L but remains in use by principals A, B, and

C. The segment was previously owned by L, but now has no owner. The existence of S continues just until A, B, and C have abandoned use of it. Since we assume there can be no more than one owner of an object, the only alternatives are to assign ownership to one of A, B, or C (but how do we choose?), or to generate separate copies of S for each sharing principal.

THE STRUCTURE OF NAMES Since every computation initially has in its C-list at least one root directory capability, it is clear that by giving a series of **acquire**'s, a computation can make its way through the directory structure along any path, as long as it knows the correct series of name components to use. A series of name components leading from a directory to an object is called the partial name of the object with respect to that directory.

Because of the structure of the directories, an object can have many names, as well as many partial names with respect to any directory. For example, the directory structure in Fig. 6 shows a particular segment, owned by the principal FORTRAN, which has the following names.

FORTRAN, MATRIX, MULTIPLY

DENNIS, EXPERIMENT, SUBROUTINES, MATMULT

DENNIS, CIRCUITTHEORY, MAXPROD

VANHORN, DENNISEXP, SUBROUTINES, MATMULT

Notice that the item named DENNISEXP within the root directory VANHORN points to the directory whose full name is DENNIS, EXPERIMENT.

SHARING MECHANISMS Two mechanisms to allow the sharing of retained objects are described here. One mechanism gives blanket authority to all computations within the system to **acquire** the shared object. The other mechanism allows the owner of an object to specifically authorize each instance of its sharing.

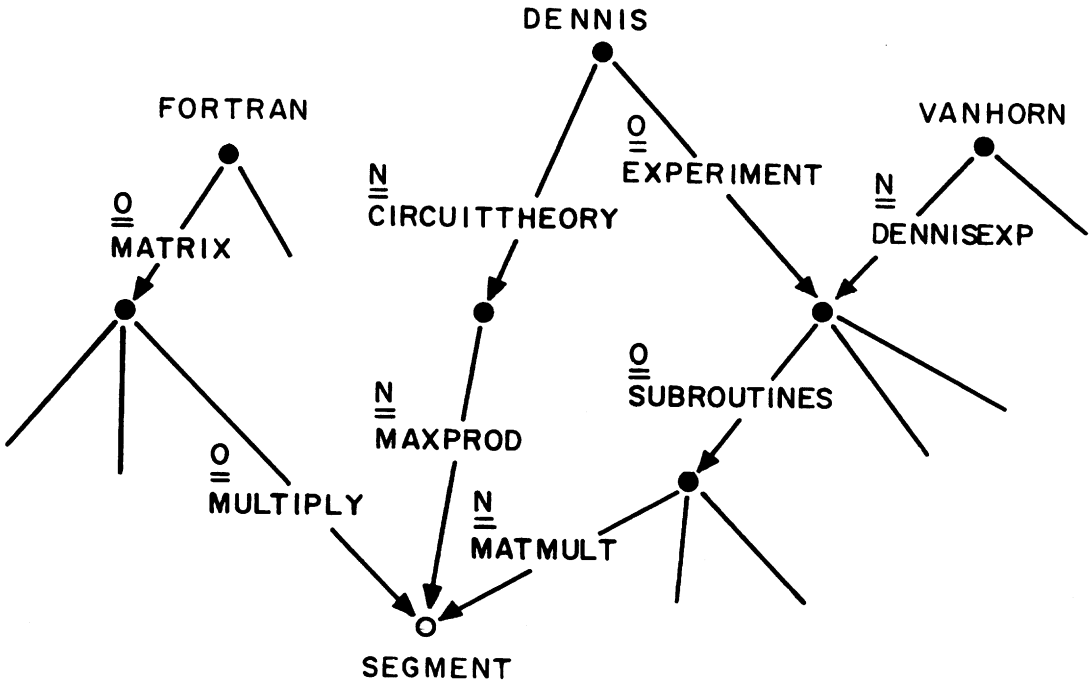

Figure 6. A Directory Structure

The meta-instruction

i : = **link** < principal name > ;

inserts into the C-list at index i a non-owned directory capability pointing to the root directory named < principal name >. Using the **acquire** meta-instruction, a computation can thus gain access to any object in the directory structure of any principal, provided that the directory items leading from the principal directory to the object all contain **F** indicators.

Any more selective sharing mechanism requires an explicit interaction between the borrower and the lender. We propose that the shared capability be passed between the C-lists of two computations that interact via the **enter** meta-instruction.

A typical interaction might proceed as follows. The lender first creates a free entry capability in one of its directories. The borrower then uses **link** and **acquire** to place this entry capability in its C-list. The borrower next creates a special entity in its C-list, called a receiver, by means of the meta-instruction

i : = **receive**;

Finally the borrower exercises the entry obtained from the lender by using **enter**. Parameters passed as private data provide to the lender the index i of the receiver in the borrower's C-list, as well as information identifying the capability desired to be borrowed.

The lender is thus given control, and proceeds to verify the right of the borrower to obtain the capability requested. In particular, the lender may wish to verify that the borrower computation is in fact owned by a certain principal. For this purpose the lender uses the meta-instruction

s : = **owner** j;

where j is the index in the lender's C-list of the suspended process capability generated by the **enter** operation, and s is a string giving the principal name of the owner of the suspended process.

Having completed its verification, the lender then **acquire**'s into its own C-list the owned capability it wishes to transmit. If this capability has index k , the meta-instruction

transmit j, i, k;

replaces receiver i in the C-list of suspended process j with the owned capability k , giving it an **N** tag.

Having modified the borrower's C-list, the lender then returns control to the borrower with **continue**. At this point the loan is complete; the borrower may now exercise the capability and **place** it in one of his own directories.

AN EXAMPLE - USING A PROGRAMMING SYSTEM Suppose a user wishes to use a programming system (PS). The retained objects (procedure segments, directories, entries, etc.,) of PS are on file in the hierarchical organization already outlined (Fig. 7a). The user has his objects organized in a private hierarchy (Fig. 7b). If the use of PS is only desired for one user then it is appropriate for an owned item in the user's directory structure to point to the directory structure of PS. If it is desired to make PS available to many or all principals at an installation, it is appropriate to place the directory hierarchy of PS under a principal of its own or as a subhierarchy within the domain of a common programming system principal. In either case, a computation for a user involving retained objects, both of his own and of the PS, would be carried out in the following manner:

1) The user initiates a process which acquires access capabilities for the two hierarchies of directories - one for his own files and one for PS — by executing the necessary sequence of meta-instructions. Suppose these capabilities have index numbers i and j respectively.

2) PS is called with i and j as parameters. PS does all addressing within the directory structure relative to the roots of their trees represented by entries i and j of the C-list (Fig. 7).

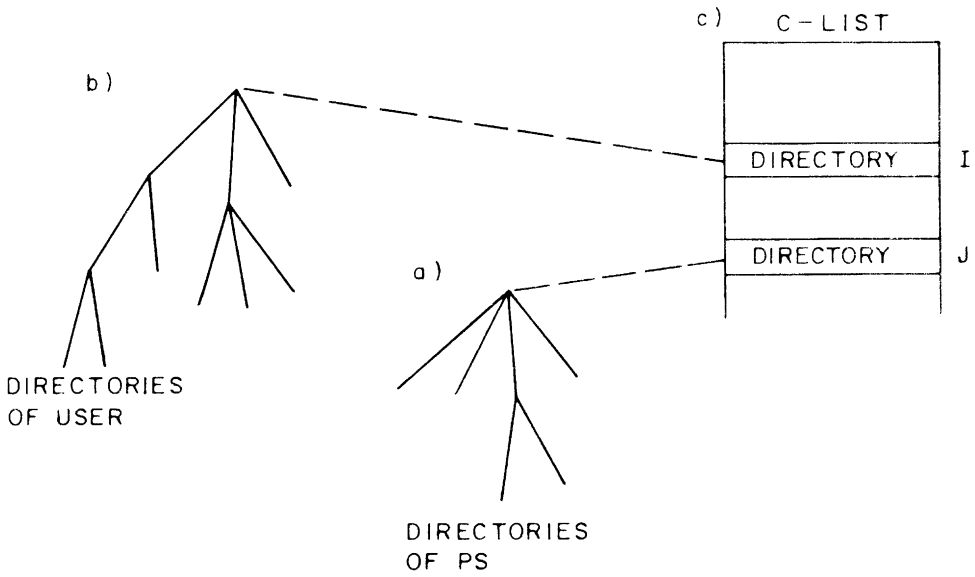

Figure 7. Using a Programming System

ACKNOWLEDGEMENT

We are indebted to Project MAC and the Compatible Time-Sharing System for the opportunity to make observations that have motivated much of the content of this paper. Our notion of the capability list stems from the "program reference table" idea first used in the Burroughs B5000 system. The value of duplicating private data at a fork was pointed out by H. Witsenhausen in an unpublished memorandum.

REFERENCES

1. Desmonde, W. H., Real-Time Data Processing Systems: Introductory Concepts, Prentice-Hall, Englewood Cliffs, N.J., 1964

2. Hamlin, J. E., "A General Description of the National Aeronautics and Space Administration Real-Time Computing Complex," Proceedings of the 19th National Conference, A2.2-1 to A2.2-22, Association for Computing Machinery, New York, 1964

3. Fano, R. M., "The MAC system: The Computer Utility Approach," IEEE Spectrum (January 1965), pp. 56-64

4. Dennis, J. B., and E. Glaser, "The structure of On-Line Information Processing Systems," Information Systems Sciences: Proceedings of the Second Congress, 1-11, Spartan Books, Baltimore, 1965

5. Iliffe, J. K., and J. G. Jodeit, "A Dynamic Storage Allocation Scheme," The Computer Journal (October 1962), pp. 200-209

6. Greenfield, M. N., "FACT Segmentation," AFIPS Conference Proceedings 21, Spartan Books, Baltimore, 1962, pp. 307-315

7. Holt, A. W., "Program Organization and Record Keeping for Dynamic Storage Allocation. Communications of the ACM (October 1961), pp. 422-431

8. Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems," Journal of the ACM (October 1965), Waverly Press, Baltimore, pp. 589-602

9. Glaser, E., J. Couleur and G. Oliver, "System Design of a Computer for Time-Sharing Applications," AFIPS Conference Proceedings 27, Spartan Books, Baltimore, 1965, pp. 197-203

10. Forgie, J. W., "A Time- and Memory-Sharing Executive Program for Quick-Response, On-Line Applications," AFIPS Conference Proceedings 27, Spartan Books, Baltimore, 1965, pp. 599-611

11. Comfort, W. T., "A Computing System Design for User Service," AFIPS Conference Proceedings 27, Spartan Books, Baltimore, 1965, pp. 619-626

12. McCullough, J. D., K. H. Speierman, and F. W. Zurcher, "A Design for a Multiple User Multiprocessing System," AFIPS Conference Proceedings 27, Spartan Books, Baltimore, 1965, pp. 611-619

13. Dennis, J. B., Program Structure in a Multi-access Computer, Project MAC, Technical Report MAC-TR-11, M.I.T., Cambridge, Mass., 1964

14. Conway, M., "A Multiprocessor System Design," AFIPS Conference Proceedings 24, Spartan Books, Baltimore, 1963, pp. 139-146

15. The Compatible Time-Sharing System: A Programmer's Guide, Crisman, P. (editor), second edition, M.I.T. Press, Cambridge, Mass., 1965, Section AD.2

16. Hosier, W. A., "Pitsfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," IRE Transactions on Engineering Management, EM-8 (June 1961), pp. 99-115

17. McCarthy, J., F. J. Corbato, and M. M. Daggett, "The Linking Segment Subprogram Language and Linking Loader." Communications of the ACM (July 1963), pp. 391-395

18. The Compatible Time-Sharing System: A Programmer's Guide, M.I.T. Computation Center Staff, first edition, M.I.T. Press, Cambridge, Mass., 1963

5. Dennis, R. C., and P. G. Neumann, "A General-Purpose File System for Secondary Storage," AFIPS Conference Proceedings 27, Spartan Books, Baltimore, 1965, pp. 213-251

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-23

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 52 (59-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOWING TITLE PAGE

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAPS (1-52) UNNUMBERED TITLE & BLANK PAGES	i-iv, 1-46
(53-59) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)	

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/10/96 Date Returned: 1/11/96

Scanning Agent Signature: Michael N. Cook

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE
Programming Semantics for Multiprogrammed Computations

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
A paper presented at the ACM Programming Conference, August 1965

5. AUTHOR(S) *(Last name, first name, initial)*
Dennis, Jack B., and Earl C. Van Horn

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1965	51	19

8a. CONTRACT OR GRANT NO.
Office of Naval Research, Nonr-4102(01)
b. PROJECT NO.
Nr-048-189
c.
d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC-TR-23

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D. C. 20301

13. ABSTRACT

The semantics are defined for a number of meta-instructions which perform operations essential to the writing of programs in multiprogrammed computer systems. These meta-instructions relate to parallel processing, protection of separate computations, program debugging, and the sharing among users of memory segments and other computing objects, the names of which are hierarchically structured. The language sophistication contemplated is midway between an assembly language and an advanced algebraic language.

14. KEY WORDS

Computer
Machine-aided cognition
Multiple-access computers
On-line computer systems
Programming semantics
Real-time computer systems
Time-sharing
Time-shared computer systems

DD FORM 1 JAN 64 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 11/10/1996
M.I.T. Libraries
Document Services