MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Machine Structures Group Memo No. 20 January, 1966

PROTECTED SERVICE ROUTINES AND INTERSPHERE COMMUNICATION

by

Peter J. Denning

INTRODUCTION

When one reads the literature in an attempt to learn about Protected Service Routines (PSR for short), he finds that over a period of several years the ideas about them have evolved little, despite the fact that other ideas have evolved freely. One wonders why a notion which is claimed to be fundamental has so many specialized provisions made for it. Then one asks: "Just what is going on here?" In this paper we attempt to answer this question. We wish to raise serious doubts concerning the fundamentality of the notion of "Protected Entry Points" [4]. We wish to show that the real problem is that of Intersphere Communication and by solving it we may dispense with "Protected Entry Points" *per se*. We start by discussing Spheres of Protection; then we give a survery of the ideas up to the present time on "Protected Entry"; then we introduce and discuss the notion of interdata communication, offered as an alternative to interprocess communication; then we propose some meta-instructions to implement the ideas; finally we show how a Service Routine might act in such an environment. Because we suggest that data be communicated, a Service Routine will always exist in an autonomous Sphere of Protection, with no possibility of "unauthorized entry" by other processes. Hence we will use the term "Service Routine" rather than "Protected Service Routine".

SPHERES OF PROTECTION

A computation must be denied access to any data objects it is unauthorized to use. It is useful to think of a computation as being confined by a Sphere of Protection, whose boundaries are defined by a List of Capabilities, or C-list [4]. A capability may be a segment reference capability, a process control capability, or a directory capability. [We will introduce a communication capability later.] A Sphere of Protection is created by another Sphere, and is said to be inferior to its creator. There is one Sphere with no superior, called the Master Sphere; it has supreme authority over the entire hierarchy of computations in a Multiprogrammed Computer System (MCS). A major reason for thinking of a computation's Sphere of Protection being created by another computation is program testing [4]. For example, a program under test may have to have access to a user's permanent objects and previously checked-out procedures; since it is likely to be faulty, it is desirable to protect these objects from unintentional use or destruction by the program being debugged.

There are other reasons for thinking of a hierarchy of computations. A particularly important one is that of naming. It is important that every computation have a unique name if intersphere communication is to be possible. A hierarchical structure is, like a family tree, ideal for specifying unique names: every branch can be uniquely labelled. We would like to point out that the name structure imposed by a file directory system [3,4] is ideal for naming computations. One can think of a computation as

being originated (or initiated) in some file directory; because the directory is uniquely named, and because the initial computation is the head of its own subhierarchy of spheres, it follows that each computation (sphere of protection) can be uniquely named. Consider for example Figure 1. The white nodes represent file directories, the black nodes represent computations associated with the directories. In particular, Sphere S can be specified exactly by the name "1;1;1;2" (using the notation of reference (3)]. Similarly S' can be named "2 1". The important point is that the "root sphere" of some programming language system originates in some file directory. Therefore every sphere of its subhierarchy can be uniquely named.

PROTECTED SERVICE ROUTINES

The notion of a Protected Service Routine was first introduced in a paper by Dennis and Glaser [2]. It is clear that there are certain service functions which must be shared, perhaps concurrently, by many distinct computations. Examples of such service functions are i/o functions or maintenance of a common data structure. Protection is necessary for three reasons. The Service Routine may be acting as the controller for some i/o function and a misbehaving computation must not be allowed to cause either the Service Routine or another computation to malfunction. Or, the Service Routine may be in charge of maintaining a common data base (for example, the usage tables for a disc file, or a linked data structure) for several computations, and meaningless modification of the data must be prevented. Finally, if the Service Routine itself is being debugged, the entire MCS

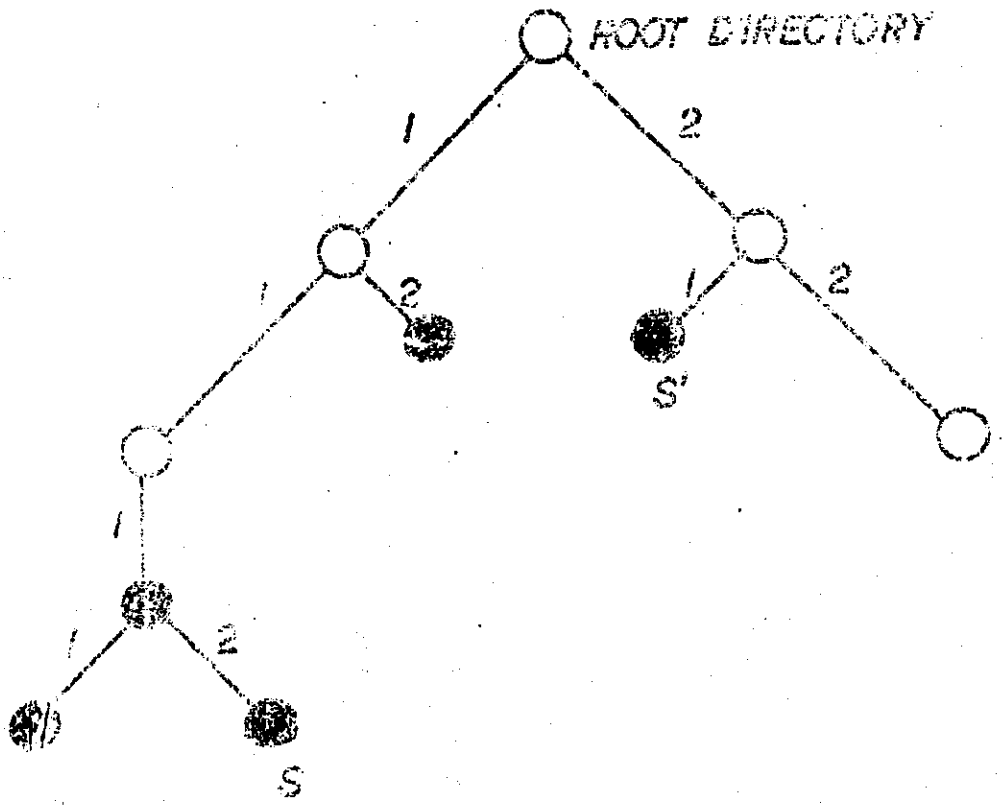


Figure 1. A hierarchy of names.

must be protected from catastrophic failure if the Service Routine is faulty.

The Service Routine cannot operate in any sphere inferior to any of the computation it serves, because that computation would have complete jurisdiction over the routine, and could, for example, delete it. Neither can the Service Routine operate in a sphere superior to any of the computations it serves, because a malfunction within itself might destroy the computations it serves. The Service Routine must therefore operate in an exterior sphere. That is, it must be autonomous, and neither superior nor inferior to any computation it serves.

Dennis, in an earlier paper [1], pointed out that special action must be taken for entering a Protected Service Routine. He spoke of "sphere entry" in the sense of the familiar subroutine entry of single-process computations. Entry was accomplished by an "enter sphere" meta-instruction, which essentially diverted the process from whatever it was doing, send it on a detour through the code of the PSR; control came back to the caller by means of "sphere return". The concept was that of a flow of control -- a process left its own sphere and crossed the boundary into that of the PSR, where it performed its function, and returned.

In a recent paper Dennis and Van Horn [4] again considered the problem, this time calling it a problem of "Protected Entry Points". Now the concept was generalized and refined. At its initiation the PSR would make available to interested computation, via the Supervisor, the capability to enter it at one or more selected entry points. Subscribing computations would then copy this entry capability into their own C-lists. When a process attempted

entry, the "calling process" would essentially fork a new process in the Sphere of the PSR (at the entry point) and would itself be suspended. The new process set up interlocks so that just before it quit it reinitiated its predecessor. The operation is illustrated in Figure 2. It is important to note that the notion of entry is still a flow of control concept, even though the service function is carried out by a new process, which is presumably well-behaved. There is an analogy with runners at a relay race: the first runs up and passes the other the baton (control), then stops and waits until it is returned.

There are some difficulties associated with this control-flow notion of entry. A certain amount of computation which cannot easily be made a hardware function must be performed upon entry and exit. For example, switching of processes must be accomplished upon entry. This requires new entries on the Process List. A capability to restart the suspended process must be added to the new process' C-list. A capability to modify the state word of the calling process must also be added to the new process' C-list. Acquisition of capabilities and state modification may be accomplished by Supervisor call only, and will therefore result in degradation of service if repeated often (as might well be the case in an interactive system).

In search of a better scheme for entering a PSR we came upon the ideas presented in the next few pages concerning Intersphere Communication. At the end of the paper we show how these more general ideas can be applied to Service Routines, and make it possible for interactions to take place quickly and respectably.

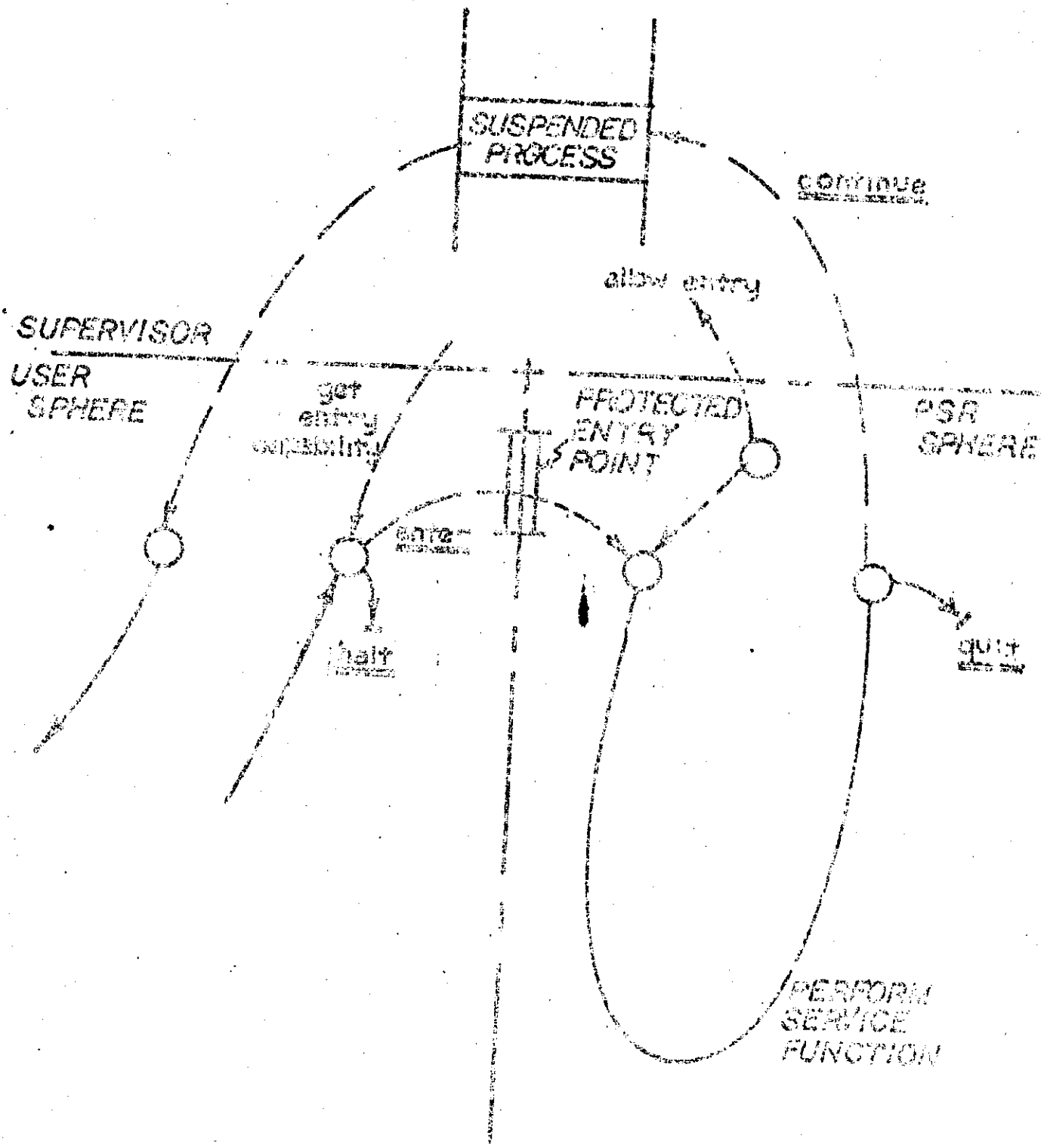


Figure 2. Protected Entry.

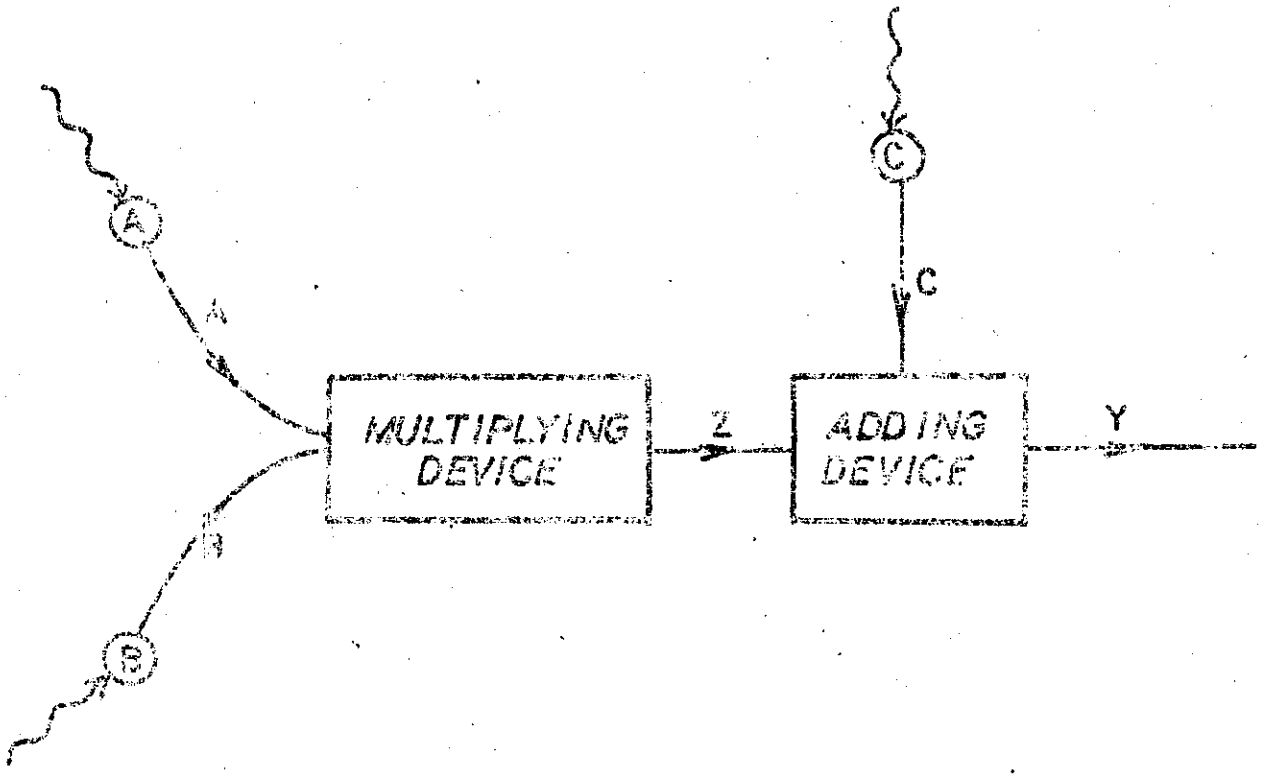
THE 'FLOW OF DATA' CONCEPT

In all the previous thinking about an MCS, a process has been thought of as a "flow of control". In fact Dennis and Van Horn [4] have defined a Process as "...a locus of control within an instruction sequence. That is, a process is an abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor." One may think of the history of a computation as being defined by 'where the control has been'. But one may also think of the history of a computation as being defined by the sequence of values taken on by all the operands within a computation. This leads us to the "flow of data" concept.

As an example of what is meant by "flow of data", consider the equation Y = (A x B) + C. We are used to programming this as

```
CLA  A
MUL  B
ADD  C
STO  Y
```

If we wished to emphasize the role that the data A, B, and C play in computing the data Y, we could think in the terms illustrated in Figure 3. When the value of A is defined (for example, it is read into the operand, or it changes value) that data A is said to be available. When A or B or both are available, the multiplication device makes the product Z available at its output. Whenever C or Z or both are available, the addition device produces the value Y. At some later time new values of A, B, or C might become available, and produce a new value of Y. There are two important points to observe: (1) it is the asynchronous availability of new data values that triggers a change in the network, and the propagation of this change can be thought of as data flow; (2) the computation can be defined by a sequence of values of the data, and large amounts of time might pass between such changes.



Arrows represent data flow, not control flow.

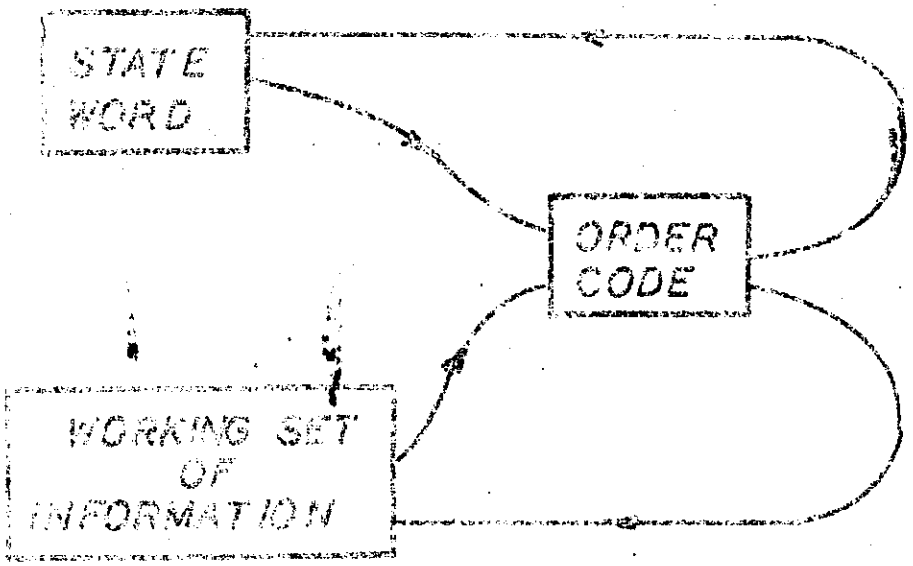
Arrows represent arrival of data from outside sources.

Figure 3. An Example of Data Flow.

Notice that the notion of control flow is implicit here in the data flow idea because execution cycles must have effected the changes in operand values. The notion of data flow, on the other hand, is implicit in that of control flow because changes in operand values affected the execution of processes. The two concepts are as intimately related as the "time" and "frequency" domains of linear system theory.

We might emphasize the duality between the control flow idea and the data flow idea. One normally thinks of the "state of a processor" as being given by the contents of its registers -- the state word. A state word can be thought of as a snapshot of the status of execution of a program. But a state word is an operand. The value of the operand named 'State Word' and the value of the data and procedure segments (which are also operands), together with the order code of the computer affect their own next values. This idea is illustrated in Figure 4.

It is now time to ask ourselves what we mean by a process. Is a process a "locus of control" [4]? Apparently not, for we can also think of it as the propagation of a wavefront of changing data through a network. Is a process then a "locus of data change"? Apparently not, else the idea of control flow would never have evolved. We propose that a process be considered a <u>locus of cause and effect</u>. In our new terms, then, a process is said to be <u>running</u> if its state word is loaded in a processor, so that changes are occurring in some region of the computation to which it belongs (that is, a processor is carrying out data changes); <u>ready</u> if it is in a queue awaiting the attention of a processor; <u>blocked</u> if it is in a period of indefinite duration, waiting for the availability



[illegible] A compiler viewed as a data-controlled device. Arrows represent data flow.

availability of data from either the outside world or another process.

Now we can point out that Intersphere Communication need not be accompanied by a transfer of control, but only by a transfer of data. Communication occurs between the data, not the processes. "Communication between data" means that communication is to be accomplished, at least conceptually, by one computation effectively "thrusting the data into the face" of another. Thus being jarred, the recipient computation takes action on the message so abruptly received. It is the arrival of the data that causes action in the receiver to take place, not a process entering into the recipient computation and "pulling the message in after it." In particular the "Protected Entry Point" is unnecessary. All that is necessary is an authorized flow of data between the service routine and the calling computation.

To provide for execution of the procedure code of the service routine, we may think of a caretaker process, permanently housed within the Sphere of Protection of the service routine. The caretaker process remains suspended until a piece of data from outside it is available; this triggers the process, which then performs its service. Figure 5 illustrates. It is interesting that this notion of a caretaker process has two important consequences. (1) Each i/o function is controlled by exactly one caretaker, which is well-behaved, and which by virtue of the fork need never divert attention from its vigil for new data. (2) A queue of the forked state words might form, blocked from completing a communication (more about this later).

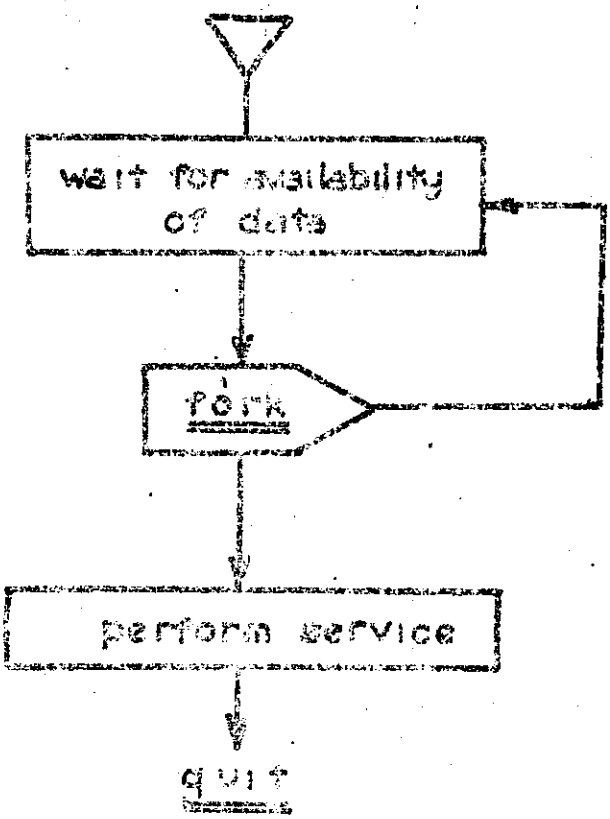


Figure 5. A caretaker process.

If an i/o function appears to require more than one caretaker, we believe that closer examination of the function can resolve it into more than one function.

We now turn attention to some meta-instructions which implement data communication between spheres. Then we will show how they may be used to call Service Routines.

INTERSPHERE COMMUNICATION

In the following paragraphs we will show how intersphere communication might be accomplished. The reader will recall that we mentioned earlier how every computation could be uniquely named because it originated in some file directory. Due to their intimate relationship with the file directory structure, Spheres of Protection can have communication in much the same way that file directories have links [3]. After all, a link is nothing other than a communication capability from one directory to another. Hence the ideas presented here can be applied to the problem of file directory linking.

Conceptually we can think of computations being sub-hierarchies hanging like fruit on the File Directory Tree*. In particular, the Master Sphere of the MCS can be thought of as being attached to the Master (or "root" [3,4]) Directory. When it wishes to communicate outside itself, a computation must first acquire a communication capability. Then it sets up a communication variable, which will exist outside all spheres of protection (a good place is in the file directory of the owner computation).

*It is hoped that the reader will excuse the analogy.

The originating computation owns the communication variable and can send reading or writing priviledges for that variable to other computations. It can also relinquish ownership, passing it to another computation, or giving it up entirely. Thus at most one computation can own the communication variable It none owns it, the variable will be deleted from existence by the Supervisor as soon as all outstanding reading and writing privileges have been exercised. Only one computation at a time has writing privileges. All non-owning computations linked to the variable will be called subscribers.

Since ownership may be passed around, all data concerning privileges and subscribers must be kept with the communication variable, and not in the C-lists of the subscriber computations. Refer to Figure 6 during the following discussion.

The Subscriber List is the list of subscribers permitted by the owner. In addition to naming each subscriber and pointing to one of his C-list entries, each entry in the Subscriber List contains a W (write permit), R (read permit), or B (blocked) bit. The owner's entry in the Subscriber List contains an inhibit count C, which is the total of outstanding W or R privileges. It is incremented by one each time a W or R privilege is sent, and is decremented by one each time a W or R privilege is either exercised or revoked. The owner's entry also contains a total B-bit count BC, which is the total of the B bits on in the Subscriber List.

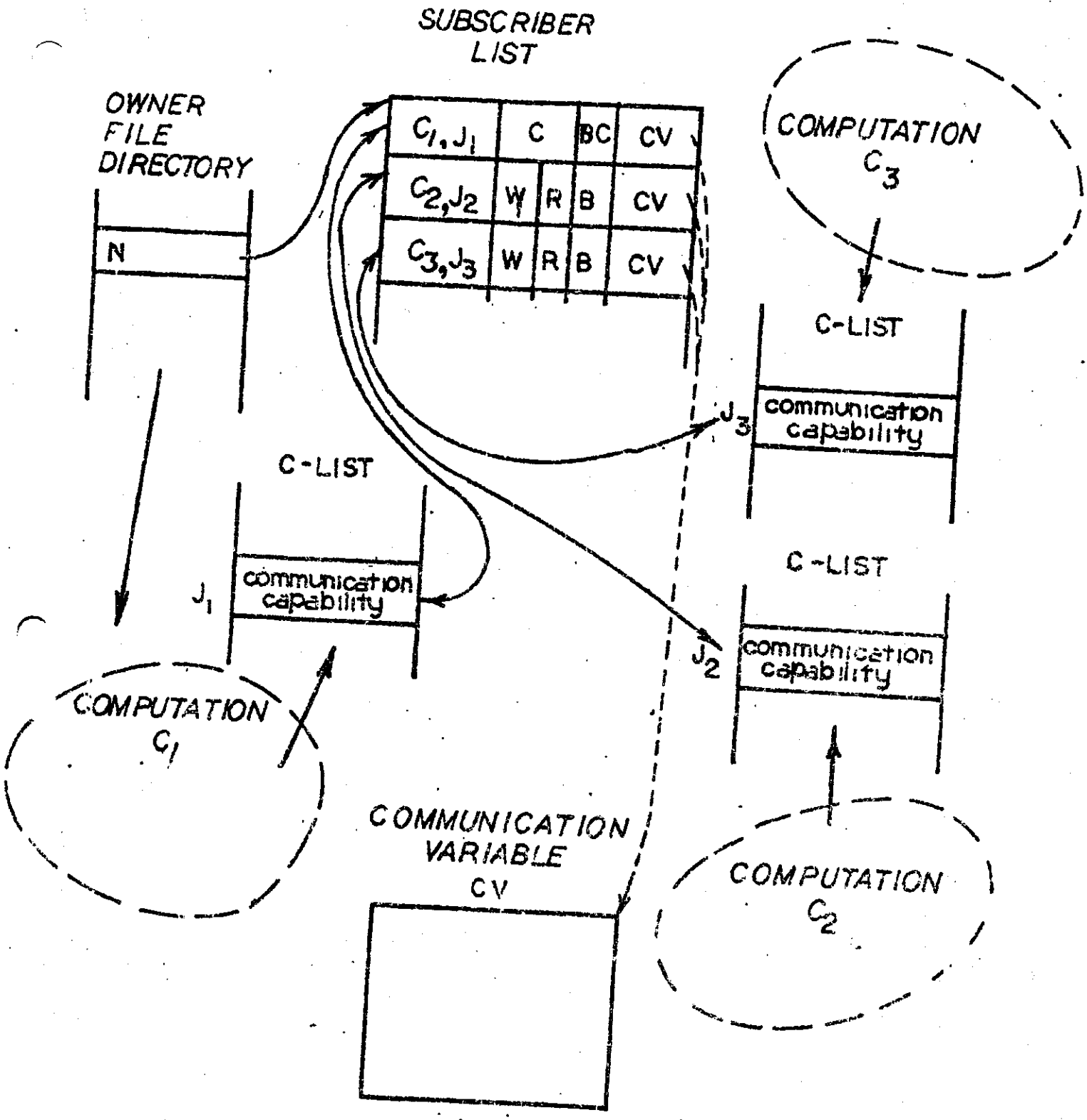


Figure 6. Organization of a communication capability.

A subscriber's B-bit acts as a flag to signal that he has attempted to read or write into the communication variable CV, but did not yet have the required R or W permission. This allows the Supervisor to detect that a process is to be restarted when it sets a W or R bit. If a process quits before exercising its privilege, its entry must be cleared, and necessary adjustments made to C and BC. The owner may write into CV whenever C = 0, and may read it whenever no W permission is outstanding.

When a computation decides to establish a communication variable, it issues the meta-instruction

j := establish communication variable N, 'n'

where j is the index number of the communication capability acquired by this meta-instruction. The system will make an entry of name N in the owner's file directory, pointing to the Subscriber list. The name N has been previously agreed upon by other communication between the computations, or by agreement among the Principals. The owner's name (with respect to the file directory structure) and j are entered as the first entry in the Subscriber List; finally there is a pointer in the Subscriber List to the name of the communication variable, CV, (which the System assigns) of the communication variable, a segment of at least 'n' words in length. The owner may now issue a series of one or more

i := permit j, ⟨name⟩

meta-instructions, which makes successive entries in the Subscriber List pointed to by capability j. ⟨name⟩ is the name of the designated Subscriber Computation. Eventually a back pointer will be inserted, pointing to the Subscriber C-list entry associated with this communication variable.

The integer i is the index in the Subscriber List given to the permitted computation name . A Subscriber computation blocked because its permission had not yet been granted will be restarted. The

revoke j,i

meta-instruction deletes the i^{th} entry from the Subscriber List pointer to by owner's capability j, and also removes the communication capability from the subscriber's C-list by using the back pointer. "revoke" can not take effect until the designated subscriber has exercised any outstanding W or R privilege.

A subscriber issues the meta-instruction

j := link communication N1,N2

where j is the index number of the communication capability acquired by this meta-instruction, N1 is the name of the owner computation, and N2 is the name in the owner's (N1's) file directory of the pointer to the Subscriber List, i.e., the name of the communication variable. The designated Subscriber List is searched to see if the linking computation's name appears there. If it is present the index number j is inserted there to act as a back pointer. If it is not present the process issuing the "link communication" meta-instruction is suspended until the owner places the name there. If no Subscriber List named N2 is present in the directory of N1,an error condition exists.

The owner may relinquish ownership by issuing

relinquish j,k

where j is the index number of a communication capability for which he desired to relinquish ownership, and k is an integer. If k = 0 ownership goes to on one; if k ≠ 0, ownership passes to the subscriber indexed by k in the Subscriber List. The old and the new owner's entries in the Subscriber List are interchanged. It is assumed that the old owner had made the new owner aware of his impending ownership by a previous communication. If the owner's inhibit count C is non-zero, no action is taken until the count becomes zero. That is, ownership cannot change hands until all outstanding privileges have been exercised. This implies that "relinquish j,0" will eventually result in the deletion of the communication variable from existence, and in the cancellation of all associated communication capabilities. Finally, the pointer to the Subscriber List is transferred to the new owner's file directory.

Any non-owner may find out who is the owner by

N := owner j

which places the owner's name (the first entry in the Subscriber List pointed to by capability j) into N.

Any non-owner may unlink by

unlink communication j

and the communication capability whose index number is j in his C-list is deleted, his name being removed from the Subscriber List. If any of his read

or write or blocked bits, R, W, or B, is one, it is turned off, and C and BC are decremented by one accordingly.

There is always a danger that some subscriber computation may fail to issue a "done" and so succeed in blocking further use of the communication variable. A possible solution to this problem would be for the Supervisor automatically to "revoke" if the communication variable falls out of the working set of the errant computation, or if some upper limit of time, to be specified by the owner, has elapsed.

One final meta-instruction will be extremely useful. Suppose that a computation has communication through one communication variable to several computations and wishes to take action just when a message appears from any of the subscribers. This would be a means by which a caretaker may "sense" an input from the outside, The meta-instruction is

i := wait for input j

where j is a communication capability, and i is the first entry in the Subscriber List with the B bit on. This meta-instruction causes the issuing owner process to wait at this instruction until the BC bits become non-zero; that is, until a subscriber has tried to exercise a privilege it does not have. A succession of "wait for input" and "send" meta-instructions could be used to read and clear the B-bits of the Subscriber List.

The owner sends a read or write permission by*

$$\underline{\text{send}} \left\{ \begin{matrix} W \\ R \end{matrix} \right\}, i, j$$

where j is the index number of a communication capability he owns, is is the index in the Subscriber List, and W or R is the permission being granted. The W or R bit of entry i in the Subscriber List is turned on. A Subscriber may read or write via his capability j into the communication variable as if it were his own segment, according to the permission W or R. When he is done he issues*

$$\underline{\text{done}}\ j$$

and the W or R bit of his entry is turned off, and the owner's inhibit count C is decreased by one. The owner can send write permission if and only if C = 0, but can send read ownership permission at will, provided the recipient has no outstanding W or R privileges. A send cannot take effect until the recipient finishes any previous permission with a done. Each send increases C by one, and each done decreases it by one. Only when all reads and writes are complete will C = 0.

Since all contact with the communication variable is made via the C-list in the same way a segment is referenced, contact will be fast, once established. Two-communication between computations is fast since W permission can be granted quickly.

---

*Patterned after similar meta-instruction proposed by E. Van Horn for reproducible inter-process communication within the same computation.

EXAMPLE 1. AN INPUT MESSAGE HANDLING SERVICE ROUTINE

Now is is easy to see how a Service Routine might be called. An "attach i/o function" meta-instruction would establish a buffer with the Service Routine as the owner; a communication capability is entered into the C-list of the caretaker process, for the buffer. Consider the case of a Service Routine receiving console messages and dispatching them to subscriber computations, depicted in Figure 7. Each subscriber has a buffer private to it and the Service Routine. When a character arrives it enables the blocked process to proceed. The caretaker executes a fork and returns to its vigil for characters. The forked process identifies the character to determine from which console it originated, and so determines to which buffer it must go. Then it checks to see if the character is a break character; if it is, a "send R" meta-instruction is issued, allowing the subscriber to read and empty the buffer; if it is not, the character is written into the appointed buffer. If the subscriber computation has not yet issued a previous "done", all the processes wanting to write characters into its buffer will be blocked, and will form successive entries in a Blocked Process List of the Supervisor. When the awaited "done" is received the blocked processes will proceed, write, then quit*. Also any "sends" will be blocked because the send cannot take effect until the designated subscriber has issued a done. Figure 8 illustrates conceptually what is taking place.

---

*It is interesting to note that the queue of waiting processes, each wanting to write a single character, can be thought of as a buffer. This is an argument in favor of small state words, which can be switched rapidly.

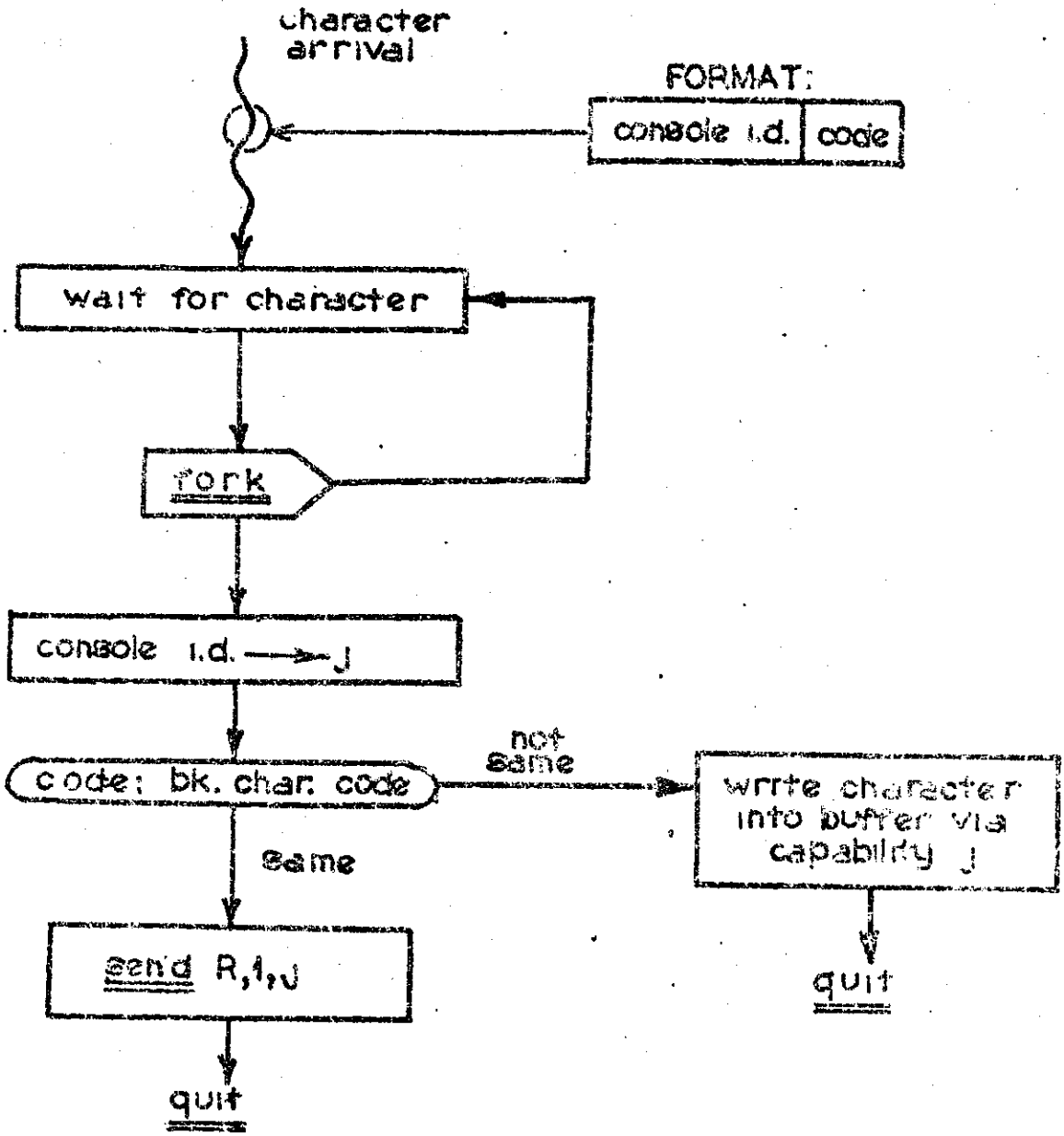


Figure 2. A console input message handling service routine.

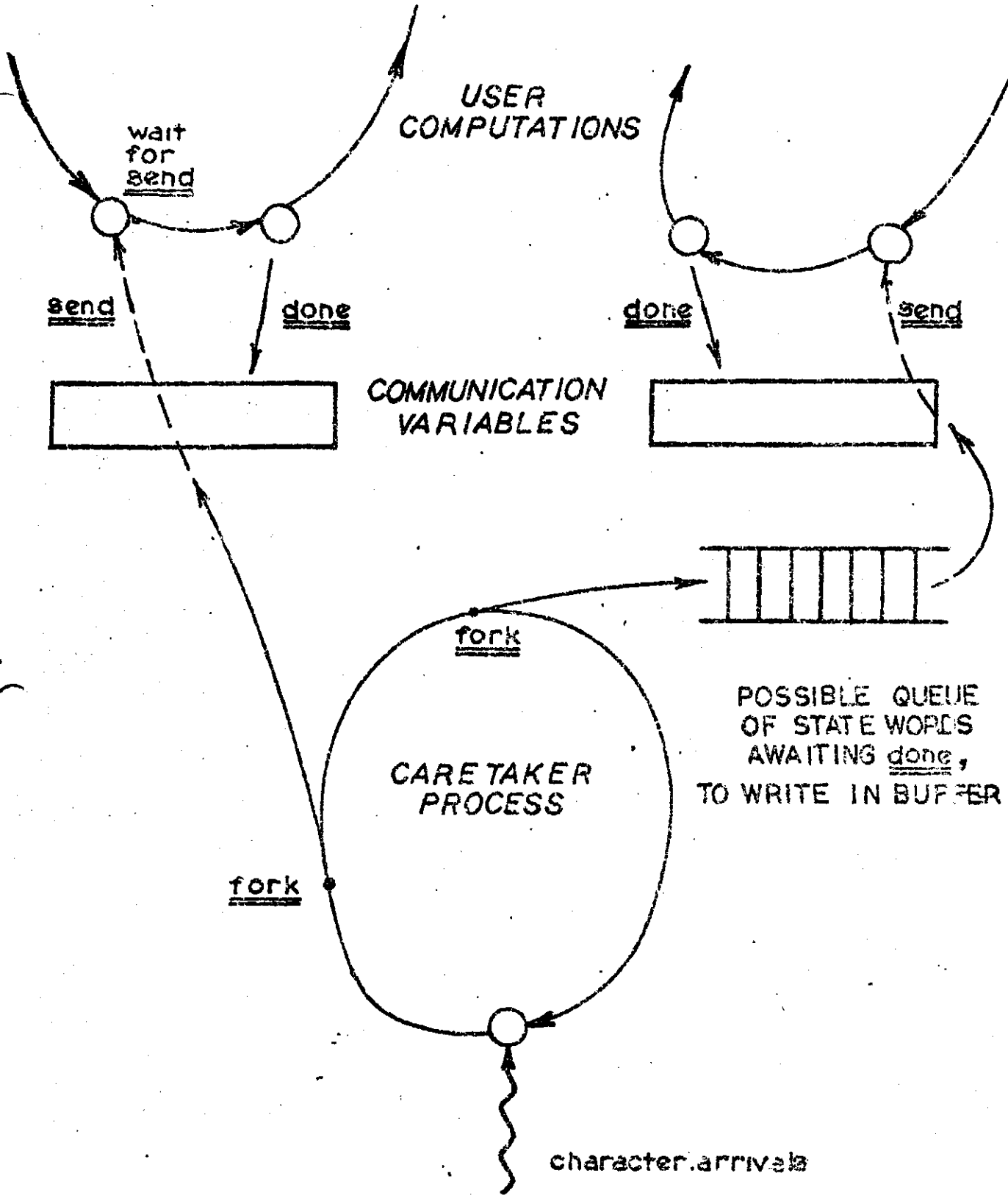


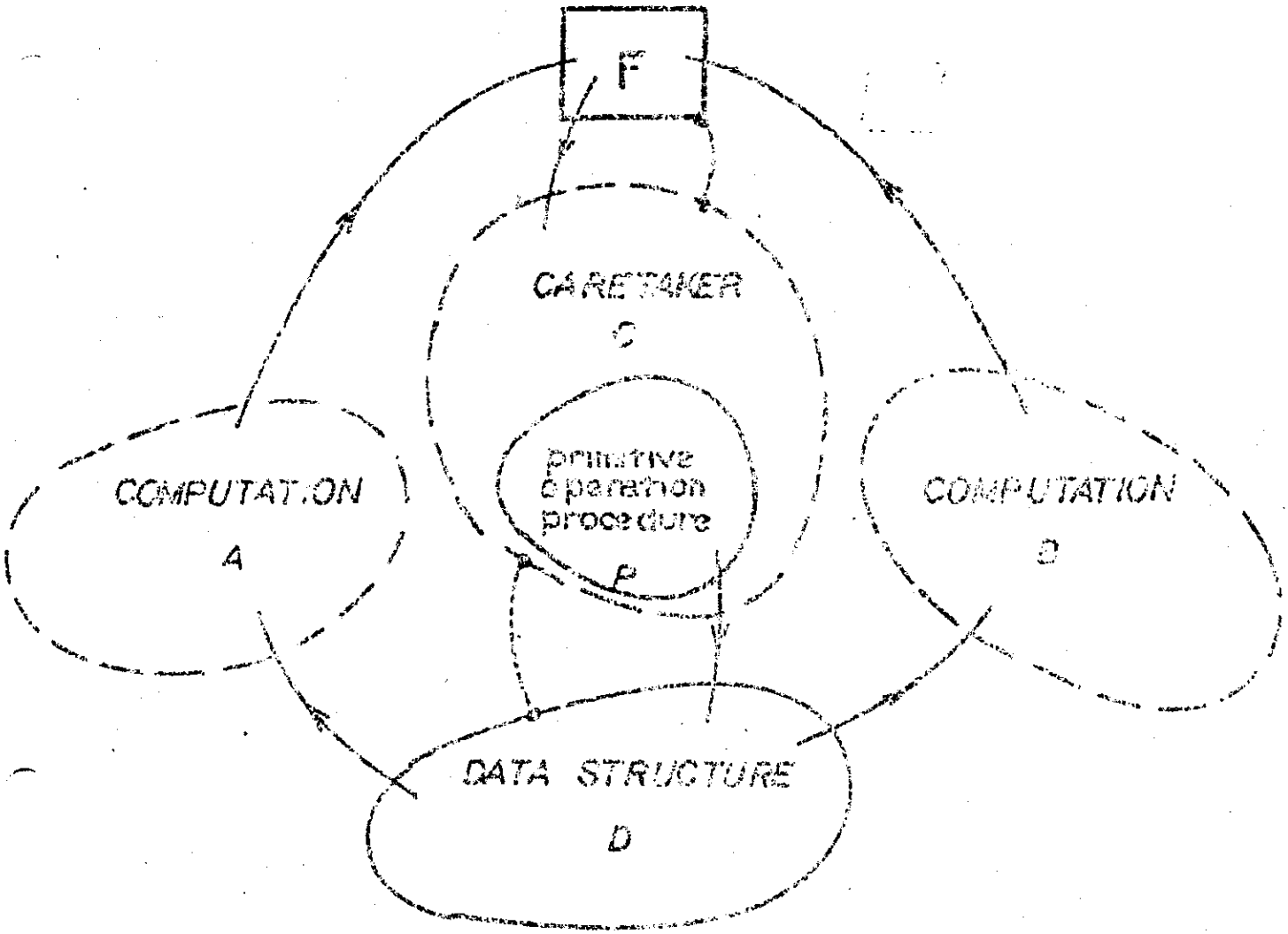
Figure 8. Using a Service Routine by means of communication variables.

EXAMPLE 2. A DATA BASE MANIPULATING SERVICE ROUTINE

Suppose A and B are computations working on some common data base D (for example, D might be a linked list structure). The primitive operations $\{P_i\}$ on D are contained in a block of procedure P. This situation might be handled as follows. Refer to Figures 9 and 10.

The procedure is embedded in a Caretaker Routine C. C is the owner of D and of a "flag" communication variable F, which is just several words long. Whenever A or B desires to read or modify D it attempts to write a code number and the modification information into F. Since it does not have W privileges with respect to F it is blocked and its B-bit is turned on. This awakens the caretaker process which forks, then sends a W privilege, and attempts to read F. When the caller writes the primitive operation code index i (i = 1,2,...) into F and issues a "done," the caretaker is allowed to read F. It then decodes i, sends an R privilege if i = 0, or goes to the proper primitive P_i for code index i ≠ 0.

The utility of the "wait for input" meta-instruction should be clear from this example, since it allows the caretaker process to sense the arrival of an event from the outside. Clearly general intersphere communication must be centered around some 'sense outside world' meta-instruction of this kind.



arrows represent data flow
dotted lines denote ownership

Figure 2. A common data base.

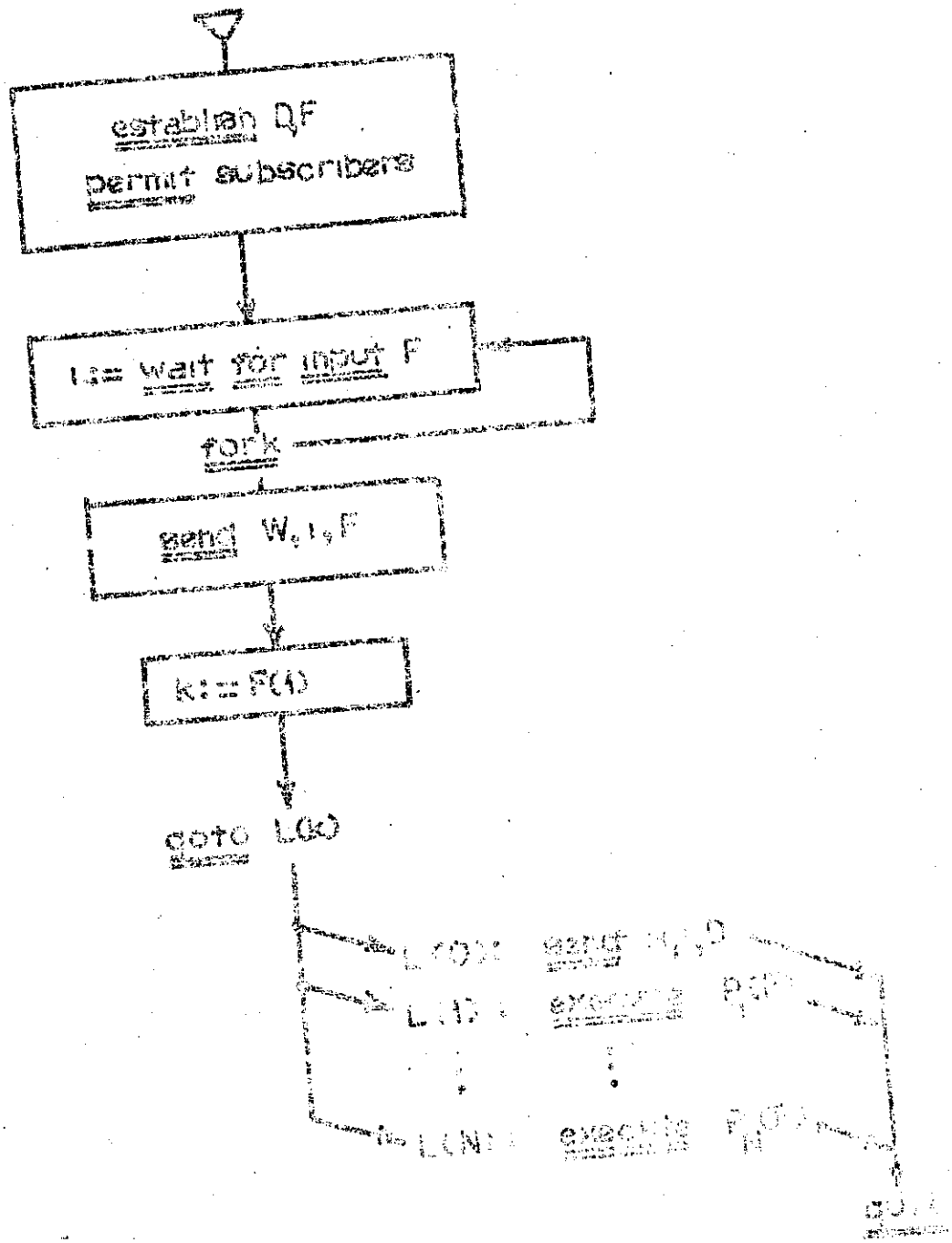


Figure 10. Caretaker for Data

SUMMARY AND CONCLUSIONS

In this paper we have investigated the history of thought behind Protected Service Routines and have discovered that "Protected Entry" is essentially a "flow of control" concept. Seeking to find a cleaner, more elegant solution, we found that "Protected Entry" is only an ad hoc solution to the most annoying aspect of a more basic problem: Intersphere Communication. "Protected Entry" is not fundamental. Intersphere Communication is. We investigated data flow briefly and found that a computation might be described by a sequence of operand values connected by a "locus of cause and effect", where the cause is the availability of a new operand value, and the effect is the next values being set. With the communication-of-data idea in mind we introduced a "communication variable", and a "communication capability" to use it, to act as the medium through which computations might communicate. Meta-instructions implementing the ideas were proposed. Special notice was taken of the fact that the permission about the vaiable must be kept with it, and not in the C lists of the subscriber computations. The communication variable was kept outside of the Spheres of Protection of the computations it linked, and was embedded in the file directory structure of the MCS. Examples showed how these concepts could be applied to understanding the use of a Service Routine, and to accomplishing fast, clear, yet protected communication.

The notion of a caretaker process was explored. It appears that exactly one caretaker process is needed per i/o function. Backup of state words of processes waiting to exercise communication capabilities is possible, forming in effect a variable length buffer. The implications of this last point are interesting -- it points out that a buffer can be thought of as a string of blocked state words waiting to communicate. This idea follows from the data communication notions, and shows how new viewpoints can arise

from this thinking.

One past point for emphasis: the flow of data notions can lead to better understanding of how computer systems work. We foresee asynchronous computers designed to operate on the arrival of data, rather than on the flow of control.

REFERENCES

[1.] Dennis, J. B. "An Example of Intersphere Communication and Asychronous Parallel Processing." Cambridge, M.I.T. Project MAC Memorandum MAC-M-189. September 25, 1964.

[2.] Dennis, J. B., and Glaser, E. L., "The Structure of On-Line Information Systems." Cambridge, M.I.T., Project MAC Memorandum MAC-M-181. October 1, 1964.

[3.] Daley, R. C., and Neumann, P. G. "A General-Purpose File System for Secondary Storage." AFIPS Conference Proceedings. Vol. 27, Part 1, 1965 Fall Joint Computer Conference, Spartan Books, pp. 213-229.

[4.] Dennis, J. B., and van Horn, E.C. "Programming Semantics for Multiprogrammed Computations." Cambridge, M. I. T., Project MAC Technical Report, December 1965.