COMMUNICATION WITHIN STRUCTURED OPERATING SYSTEMS*

Thomas J. Dingwall

TR 73 - 167

May 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

*This research was supported by NSF Grant GJ-28176.
# Table of Contents

I. Introduction 1

II. Basic Concepts and Definitions 3

III. Data Accessing 8

IV. Process Communication 20

V. An Example File System 27

References 30
I. Introduction

This paper discusses some desirable features in the organization of a multi-use computer system. In general, important considerations in the design of a computer system are flexibility, reliability, and speed. This paper will present mechanisms and conventions aimed at enhancing those properties.

The first desirable characteristic in a computer organization is modularity. Functionally distinct elements of a system should be defined and segregated as procedures. The advantages of a modular organization have been amply demonstrated, particularly in the designs of Dijkstra, Brinch Hansen, and Pelzer, and the project SUR system. Basically, modularity has important implications on the reliability and flexibility of a system.

Secondly, because of reliability considerations, the system must be secure. That is, distinct system components must be protected from one another, even while the various components are communicating with one another. For the purposes of this paper, Lampson's protection theories are sufficiently powerful. However, his models will only be applied to a communication device between otherwise isolated processes. Hence, some of his very general concepts will be given more restricted meanings in order to reduce the complexity of the protection system, while still maintaining completeness.

A very important consideration in designing such a communication system is efficiency. An increase in efficiency of communication in some given system can be utilized in one of two ways:

1. The system design could remain the same, with a resulting increase in speed.
2. The system could be modularized to a greater extent, providing greater protection and thus greater reliability, with no reduction in speed.

Unless the system were modularized as completely as possible, alternative 2 would probably be chosen because of the value of small, isolated modules.

Message passing systems such as Brinch Hansen's\textsuperscript{2} and project SUE\textsuperscript{5} have basically secure communication systems; however, they are not especially efficient for transferring large amounts of data. Shared storage systems such as MULTICS\textsuperscript{9} and ISPL\textsuperscript{1} provide greater efficiency; however, sharing on a segment basis means protected communication is inefficient or difficult for small amounts of data. Thus, a major goal of this paper is to provide a means of communication suitable for widespread application.

One basic strategy is used throughout the paper: Specify enough of the structure of a system so as to limit its possible complexity while still providing full generality. An analogy with programming languages may be drawn: Algol programs are more readable (understandable) than machine language programs because Algol defines more structure than a machine language. On the other hand, the class of Algol programs is as powerful (in the Turing sense) as any class of machine language programs.

In this paper, conventions are introduced for data accessing and process communication. These conventions will force all programs in the system to exhibit desirable properties, without limiting generality.

This section has outlined the basic goals of the paper. The following sections will develop the ideas introduced in this section.
II. Basic Concepts and Definitions

In this section, some basic elements in the organization of an operating system are described. Most terms used here have been previously given meaning in the literature. Thus this discussion is used to avoid confusion and to point out restricted uses of general terms. The following sections will give details of the basic elements.

A design is proposed for the kernel of an operating system. "Kernel" is used here in the traditional sense of a minimal operating system with multiprogramming and resource management capabilities. The kernel should specify system characteristics such as main memory organization (single fixed array or many variable length segments), number of processes maintained (a fixed number or variable, potentially infinite), and methods of interprocess communication (interrupts, messages, or shared memory). However, the kernel should not be responsible for making policy decisions such as which pages to replace in memory, how to schedule processes, and which processes should be allowed to communicate. The duty of the kernel is to provide means for specifying and enforcing these decisions. Balzer\textsuperscript{1} discusses the benefits of separating the decision-making elements of an operating system from the mechanisms for enforcing those decisions.

We will not consider how the kernel is to be implemented, in hardware, by microprogramming, or by conventional programming. Such decisions will have to be made on an individual system basis, considering efficiency requirements and cost constraints. Obviously some elements to be described, such as the memory addressing mechanism, will have to be realized in hardware.
The memory of the system is assumed to be organized in segments, from which more powerful organizations will be constructed. By segment is meant a linear address space of variable length consisting of fixed length words. It is assumed segments are implemented by a virtual memory mapping mechanism using fixed length pages. Such memory organizations are well-known, so that further elaboration should not be necessary.

The basic data objects in the system are structures. Structure is used here in the PL/I sense, and it may be imagined that each structure is superimposed on a memory segment. Structures provide more flexibility than homogeneous segments since access can be controlled at a finer level. Consider a data record in memory: if we can grant or deny access on an element level then we can allow a user program to read only certain fields in the record. If access is controlled on a segment basis, there are two ways to accomplish the desired masking: The record could be reformatted by copying the non-sensitive fields to a new area, or each field could be placed in a separate segment. Both methods lead to undesirable inefficiencies; the former in the time necessary for copying and the latter in storage overhead for small segments.

Procedures represent programs in the system and can be considered structures with extra properties. Ideally, we would like to think of microinstructions, machine instructions, I/O operations, and conventional programs as procedures, so that a procedure would consist of a series of calls on other procedures. Obviously, however, at some level we must have primitive, indivisible operations available. Furthermore, for efficiency, this definition of procedure may not extend
to as low a level as possible; we will only talk about procedures when access control is desired. Thus the arithmetic instructions will be considered primitive operations available to all procedures; whereas I/O operations will be considered procedures since we need to control their use.

A process is defined to be an activation of a set of mutually trusting procedures. The kernel has no control over interactions between procedures within a process. They all possess the same accessing abilities and may communicate in any manner desired. However, communication between processes is strictly controlled through a coroutine-type calling sequence. Thus processes form the boundary for system supervision of data accessing and communication. Processes are isolated in much the same way they are in a message passing system. Unlike a message passing system, however, processes are allowed to share structures in a controlled manner.

Following Dennis and Van Horn, a process carries capabilities for accessing objects. A capability is the name of an object which must be presented to the kernel in order to perform basic operations on the object. Naturally capabilities must not be generated or altered in an arbitrary way. Thus each process carries a special capability segment which is accessible only through operations in the kernel.

Capabilities for three types of objects are defined within the kernel: structures, procedures, and processes. The format of these capabilities and the types of operations they can be used for will be specified in the next two sections. However, application programs in the system will undoubtedly define and support other types of objects. Hence it would be very convenient
to allow user-defined capabilities. For such general capabilities it would be necessary to implement protection models similar to those proposed by Lamport[7,8] and Graham and Denning[6].

However, for the three kernel defined objects, such generality in unnecessary since the operations to be performed on the objects are known at the time of design of the kernel.

Capabilities are passed as arguments between processes, thus providing shared storage communication. It will be seen that ensuring data security incurs some overhead in memory accessing. Thus it can be seen that message passing systems, which are inherently secure, are efficient when small amounts of data are being passed. However, as the amount of communication between processes increases, the cost of moving information and managing buffers eventually outweighs the memory accessing overhead, thus making shared storage communication desirable. This is the basic motivation behind the concepts being introduced.

There is one notable restriction on structure capability passing: Only one process at a time may possess a capability for any given element of a structure. This is essentially Graham and Denning's transfer-only attribute[6]. The main reason for this convention is to ensure mutual exclusion. Thus, Dijkstra's P and V operations[4], are replaced by procedure calls to give and receive capabilities. Since a process must first receive a capability before accessing shared data, mutual exclusion is enforced.

Another reason for this convention is to reduce potential confusion. A process need not worry that some other process has secretly retained a data capability and is accessing information to which it is not entitled. Since the protection
structure is simpler, the chances are greater that it will be used correctly.

An example may clarify how Dijkstra's P and V operations can be replaced by capability means. Suppose n processes wish to share a table. A manager process is created which initially possesses capabilities for all entries in the table. Whenever one of the n processes requires a table entry, it sends a request to the manager process and waits for the proper capability to be returned. The manager process can accept only one request at a time, so calls are queued up automatically by the system. Whenever the manager receives a request for a capability which it possesses, it gives it to the calling process. When this process is finished with the table entry, it returns the capability to the manager so that other processes may access that entry.

There is a problem with this restriction in that a rogue process may gather data capabilities and never return them. (Only capabilities which are transfer-only present this problem.) This is not really a new problem, however, since in any system processes must wait for completion signals from other processes, creating the same sort of dependency. Thus a capability being passed back and forth really represents a communication device, and the failure of a process to return a capability is equivalent to a refusal to return from a call. It is also equivalent to a process failing to issue a V operation in Dijkstra's system.

This concludes the preliminary discussion of the proposed system. The next sections will describe in detail the data accessing and process communication mechanisms.
III. Data Accessing

Data objects in the system are represented by structures. A structure is a memory segment which has been broken up into a nested set of subsegments, called elements. A segment is divided into an integral number of fixed length pages. We can thus picture two separate and independent formats overlayed on a given segment: the first is the structure which is used to control data accessing, and the second is the page layout, used for efficiency in the transfer of data between main and auxiliary storage. In this section we discuss a possible implementation of such a concept.

A structure can be represented by an ordered tree, with each node of the tree corresponding to an element of the structure. Each node $x$ of such a tree is associated with a scalar, $\text{length}(x)$, which gives the length of the corresponding memory block. The $\text{length}$ function satisfies the following constraint: for each node $x$ with a nonempty set $A$ of offspring, relation (1) holds.

\[
\text{length}(x) = \sum_{y \in A} \text{length}(y)
\]

Fig. 1 shows several ways of representing a structure. Fig. 1a shows the memory organization of an example structure. Fig. 1b gives the ordered tree representation of the same structure, along with its corresponding length function. It is also possible to
1a. memory organization of a typical structure.

1b. tree representation of the structure in 1a.

1c. nested paranthesis representation of tree in 1b.

1d. representation of the structure in 1a. as used in the system.

Figure 1. Structure Representations
Figure 2. Address Translation
represent an ordered tree by listing the nodes in preorder, and
using nested parantheses to define the shape of the tree. In
fig. 1c (i) we see the nested paranthesis representation of the
tree in 1b. If node names are unimportant we can uniquely
specify the structure by replacing the node names with their
corresponding lengths. Fig. 1c (ii) shows this.

It is easy to show that for any such nested paranthesis
representation, if the node names and parantheses are separated
into two lists, the structure is still uniquely determined. This
is true because there is a rule for remeshing the two lists: put a
node name at the front and after every ( and , . It can be shown
that this is the only possible reconstruction of the nested
paranthesis representation.

Thus in fig. 1d, we see how the example structure would be
represented in the system. Fig. 1d (i) is a list of the node
names and lengths, while fig 1d (ii) is an encoding of the
paranthesis structure.

Next the primitive operations available in the kernel
for operating on structures are considered.

1. CREATE(format,structure), DESTROY(structure) - The CREATE
operation accepts the definition of a structure in the form of
fig. 1d, and returns a structure of the appropriate form. The
process issuing the instruction is given full read/write access
to all elements in the structure, and is charged for the storage
used by the structure. The DESTROY operation eliminates a given
structure and may be issued only by a process possessing read/write
access to all elements in the structure. A more restrictive
alternative would be to allow only the creating process to
destroy a structure.
2. EXTEND, CONTRACT (structure, length) - These functions are used to change the length of a structure. The length of a structure can be changed only at the end, thus affecting only the final structure elements. Thus in the structure of fig. 1, only the lengths of A and D will be changed by an EXTEND or CONTRACT. Also an element may not be contracted to a point where its length is less than or equal to zero.

3. GET DEFINITION(structure, definition) - This operation returns the description (lengths and parenthesis structure) of a given structure. It is used when a process needs to know the layout of a structure which it has been given.

4. READ, WRITE(structure, element, displacement, value) - These are the operations used to access memory, and every machine instruction which uses main memory must use them. They are given the address of a word in memory and perform the indicated operation on that word. A detailed description of the execution of these operations will be given below.

5. ADD DATA, REMOVE DATA(structure, element, read/write) - These operations allow a process to give another process any of its data capabilities. They are not directly executed; rather they are performed with any process synchronization command which includes an argument list. Details of the capability passing mechanism are given below. The process communication commands are described in Section IV.

A process supplies a data address consisting of three parts for the READ and WRITE commands. The first part specifies the structure being accessed and consists of an index into the processes' capability list. The second part gives the structure
element and is the rank of the element in the preorder listing.
The final part is the displacement within the structure element.

When a process gives such an address to the kernel, two things
must be done: the right of the process to access the specified
element within the specified structure must be checked, and the
physical location of the word specified by the logical address
must be found. Three different tables are used to perform
these operations. These tables are:

1. The Capability List - Each process carries with it a
list of capabilities for accessing structures and procedures.
The capabilities for structures contain two fields:

   a. A read/write access mask is used to specify accessing
      abilities on a structure element level. Two bits are used for
      each element: one to represent read authorization and the other
to indicate write authorization. The two bit fields are arranged
      corresponding to the preordered listing of the structure.

   b. A pointer to the structure definition for the structure
      being referenced.

Note that each entry in the capability segment is fixed
length, in order to ensure rapid access to any specified entry.
This implies a fixed length for the read/write mask and hence
a maximum number of elements per structure. The choice of entry
length then requires a compromise: a small entry length reduces
flexibility because of the limit on the number of structure
elements, whereas a large entry length implies much wasted
space. This conflict could be resolved somewhat by allowing
nesting of structures, which will be described later.
2. The Structure Definition - Each structure in the system carries a description of itself which essentially contains the information in fig. 1d. The structure definition contains three parts, the first of which is an encoding of the nesting of the structure elements as in fig. 1d (ii). The second part is a table consisting of the length and offset (from the start of the structure) of each structure element. Finally a pointer to the page table for the segment is given.

3. The Page Table - Each segment carries with it a page table, used for managing pages in main and auxiliary storage. The table has one entry for each page in a segment. An entry consists of status bits with information such as whether the page is in main or secondary memory, whether the page has been accessed, etc. Also included is the address of the page frame containing the page or the page location in auxiliary storage.

The table accesses necessary for address translation are now discussed. Fig. 2 gives the layout of the tables just described and the operations involved in the address translation. One access of each type of table is required for each logical memory access:

1. The segment portion of the logical address is used as an index into the processes' capability segment and the read/write mask and the structure definition address are retrieved. The element portion of the address is used as an offset into the read/write mask in order to verify the processes' ability to make the requested access.

2. The structure definition address from step 1 is used as a base, and the element part of the logical address is used as an offset into the length/offset table of the definition.
The length retrieved is compared with the displacement part of the logical address and an error occurs if the displacement is greater than the length. The offset is added to the displacement to give the position of the accessed word relative to the start of the segment. This sum is then broken into two parts: the high-order bits which specify the logical page in which the word occurs, and the low-order bits which give the displacement into that logical page. At the same time, the page table address is accessed in the structure definition.

3. The page table address and logical page address from step 2 are used to retrieve an entry from the page table. This entry is then used as in conventional virtual memory systems to generate the desired physical address: if the status bits indicate that the page is in main memory, then the page frame address is concatenated with the low-order bits generated in step 2, giving the physical address of the requested word.

Naturally to make such a scheme feasible, we must avoid making three extra memory accesses for every memory reference. An associative memory can be used with slight modifications from conventional virtual memory systems. One entry in the associative store corresponds to the intersection of a structure element and a page in the segment. The following fields are content-addressed in the table: The segment and element fields are checked for equality with the corresponding fields in the logical address. The minimum and maximum displacement fields are used to restrict one associative memory element to a single page of the segment. The displacement field of the logical address must lie between the two values.
If an entry is found which satisfies those four fields, the final three fields of the entry are returned. The read/write flags are used to indicate what type of access is allowed. The bit corresponding to the access requested must be on. The pana frame is used as the high-order portion of the physical address, while the offset is added to the displacement in the logical address (overflow is ignored) to give the low-order bits of the physical address.

Thus, it is possible to implement this data addressing mechanism with small increases in time and storage over current virtual memory systems. However, we must still consider the operation of the ADD DATA and REMOVE DATA operations. These instructions are not directly available to processes, but are instead executed as part of the process communication instructions. The full process communication system, which allows the passing of both procedure and structure capabilities, will be described in section IV. Here we consider only the conventions exclusively used for transferring data capabilities.

The ADD DATA and REMOVE DATA commands are non-trivial because changing the access capability for a structure element affects the surrounding and contained elements. Thus in order to give a process the capability for a certain structure element these operations must be done:

1. Turn on the proper access bit in the read/write mask for the structure and element specified.

2. Using the element number, scan the structure's parenthesis encoding to find the position where the element occurs.

3. Scan the parenthesis encoding to the right, counting the number of elements contained in the given element.
turn on the corresponding access bits in the capability entry.

4. Scan the parenthesis encoding in both directions to find the sibling elements of the given element. If all their access bits are on, turn on the access bit of the parent element.

5. If step 4 turned on the parent access bit, repeat steps 4 and 5 on that parent element.

The REMOVE DATA operation must perform steps 1-3 with "on" replaced by "off". It then performs the following operation:

4'. Scan the parenthesis encoding to the left, searching for the ancestor elements of the given element. Turn off the access bit on all such elements.

These operations are rather specialized and probably cannot be efficiently coded in most machine languages. Thus, it would be desirable, but not necessary, to microprogram the operations.

In certain situations it may be convenient to nest structures; i.e., to superimpose a substructure upon some structure element. Consider the following situation when this would be desirable:

A file contains records which contain fields, some of which a user is permitted to see and some of which he is not. The logical records are blocked into larger physical records. The file system wishes to avoid moving logical records to unblock the physical records. It does this by giving the user program only capabilities for the non-sensitive fields in the proper logical record. If nesting of structures is not allowed, and there are no records per block, a structure will be required with
n copies of the logical record layout.

On the other hand, a two level structure could be used. The outer structure would be an n element array, defining the blocking of the logical records. The second structure would define the format of the logical record, and would be associated with only one element of the outer structure at any time. The file system would perform unblocking by merely moving the logical record structure to the proper physical record structure element. Since this operation only involves moving pointers, it is more efficient than moving whole records. Also this scheme requires fewer structure elements than the non-nesting scheme above. This is a significant advantage in light of the fixed length restriction on capability entries (see p.11).

Some restrictions on nesting are required for security reasons. It is essential that if two elements have a non-null intersection, then one must be a subelement of the other. If this were not true, whenever the capability to a given element is removed it would be difficult or impossible to determine what other capabilities must be removed. Thus these restrictions are made:

1. Only one structure may be associated with any given structure element.
2. A structure may be associated only with the lowest level elements (leaves) of a structure.

A natural restriction on creating or removing substructures is that a process may perform one of these operations only when it possesses both read and write capabilities for the element in question.
The implementation of nested structures requires only one change in the address translation tables. Another field is added to each entry in the length/offset table for each structure definition. This field is used to point to the structure definition for a possible associated substructure.

The three field address described previously is denoted as a simple address. An address for referencing a nested structure then consists of one simple address for each level. The simple addresses for all but the lowest level structure are called preliminary addresses; the lowest level address is called the terminal address. The displacement field is needed only in the terminal address; hence the displacement fields in preliminary addresses can be used to link all the simple addresses together, ordered from outermost to innermost structure. Finally, an additional bit is required on each simple address to differentiate preliminary addresses from the terminal address.

Next the address translation process for these nested structures is considered. The preliminary addresses are used to determine the offset of the terminal structure from the start of the segment. The terminal address is then used for checking access authorization and to find the physical address of the requested word.

In the following description these conventions are used: P is a pointer to the current simple address, S, E, and D are the segment, element, and displacement portions, respectively, of the simple address pointed to by P. \( R/W(S,E) \) and \( \text{DEF}(S) \) are the capability table entries determined by S and E. Finally, \( \text{LENGTH}(S,E) \), \( \text{OFF}(S,E) \), and \( \text{DEF'}(S,E) \) are the table entries for
the Eth element in the structure definition of structure S.

The following algorithm is followed to evaluate an address
for a nested structure:

1. \( P \leftarrow \text{first simple address. (for outer structure)} \)
   \( \text{OFFSET} \leftarrow 0. \)
2. \( \text{Get DEF}(S) \) from capability segment.
3. \( \text{Using DEF}(S), \text{get LENGTH}(S,E), \text{OFF}(S,E), \text{DEF}'(S,E) \) from
   structure definition.
4. \( \text{OFFSET} \leftarrow \text{OFFSET} + \text{OFF}(S,E). \)
5. \( \text{If this is a terminal address, go to step 10.} \)
6. \( \text{TEMP} \leftarrow \text{DEF}'(S,E). \)
7. \( P \leftarrow D. \) (D is really the link field to the next simple
   address.)
8. \( \text{Get DEF}(S). \text{ If DEF}(S) \neq \text{TEMP then error. (This ensures}
   \text{the substructure specified in the address is the proper}
   \text{one.)} \)
9. \( \text{Go to step 3.} \)
10. \( \text{Get R/W}(S,E). \text{ Error if proper bit is not on.} \)
11. \( \text{Error if D} > \text{LENGTH}(S,E). \text{ OFFSET} \leftarrow \text{OFFSET} + D. \)
12. \( \text{Use the page table as in step 3 of the simple address}
    \text{translation algorithm to calculate the physical}
    \text{address of the word.} \)

It is also very easy to augment the associative memory
described above to handle nested structures. One access of the
associative store is required for each simple address in the list,
and a slightly different entry is required for the preliminary
addresses. This new entry type is matched only by the segment
and entry fields, and provides the offset of the element, and
structure definition pointers. These pointers are used for
checking that the proper sequence of structures is used, and
for proceeding with in-core tables if a further entry is missing
from the associative store.
Finally, the ADD DATA and REMOVE DATA operations must be expanded to scan containing and contained structures, so as to maintain this property. Given any structure element and any contained element, the capabilities a process possesses for the subelement must be a subset of those possessed for the containing element.

As is readily apparent, the flexibility of nested structures is offset by extra time and space requirements. Thus whenever a fixed overlay on storage is sufficient, a simple non-nested structure should be used. However, if an application requires dynamically changing an overlay, then nested structures may be necessary.

This section has discussed in detail the nature of structures and the operations on them. Section IV will describe process communication in much the same manner. Finally, Section V will present an example to show how all these mechanisms interact in a real application.
IV. Process Communication

A process is specified by a set of procedure segments and a capability segment. The procedure segments, which contain code to be executed by the processor, must trust each other since they will all possess the same access abilities. Since the system has no control over or knowledge of relations between procedures within a process, any sort of protection conventions are unenforceable within a process. This also implies that any type of communication conventions may be used within a process.

On the other hand, communication between processes follows a system-defined convention, which is enforced by the kernel. Also, accessing of objects is controlled on a process level, so that processes can protect themselves from one another. Thus the concept of process forms a basis both for protection and for a standardized communication mechanism.

By combining these functions a simple system results: Lampson's models do not assume conventions for interprocess communication and do not directly associate accessing abilities with processes. As a result his models are very complex. By introducing conventions which do not reduce the power of the system, the potential complexity is reduced.

The capability segment carried by each process contains all the status information (registers, wait/active state, user identification, etc.) plus the list of capabilities for the process. There are three types of capabilities defined by the kernel: structures, procedures, and processes. Structure capabilities were the subject of Section III and need no further explanation. A procedure capability may be used for two operations:
1. To initiate a process to execute the procedure.

2. To include the procedure in the set of procedures which constitute the given process.

A procedure capability contains the address of the procedure segment and flags indicating which of the above operations may be performed.

A process capability is used by one process to communicate with another process. It merely consists of a pointer to the capability segment of the given process.

As was mentioned in Section II, it might be convenient for application systems to maintain capabilities for objects they define. This would really not be difficult to implement since the capability segment is already implemented. Some general capability mechanism such as Lampson’s would be necessary, however.

There are several possible attributes for procedures. A reentrant procedure may be associated with several different processes at the same time, whereas a serially-reusable procedure may be part of only one process at a time. A trusted procedure may be used within any process in the system. It is essentially some primitive operation, such as a cosine function, which has been implemented in software rather than hardware. In order for a procedure to be trusted, it must be shown that the procedure will not abuse the accessing abilities it will receive as a component in any process. Essentially a procedure must be certified correct and benevolent before it may circumvent the system protection mechanisms.

Suppose process P wishes to execute some procedure A. P
calls on the loader process, which holds procedure capabilities for all procedures in the system, to ask for the capability for procedure A. The loader checks a directory to see if P is allowed to talk to A, and if this is permissible, gives the capability to P. Then P executes an OPEN command, passing the capability for A to the kernel. If the procedure is either reentrant or serially-reusable with no other associated process, the kernel creates a capability segment and starts procedure A executing at the beginning. It then places the new process capability in the capability segment of P. If the procedure is serially-reusable with an associated process, the kernel merely places the existing process capability in P's capability segment.

The processes can then communicate by a fairly standard asynchronous coroutine-type of discipline. The communication operations are essentially those of the project SUE system⁵ and ISPL¹.

1. SEND(process, capability list). This operation passes to the named process the capabilities listed and the name of the calling process.

2. WAIT(process, parameter list),(process,parameter list),... This operation suspends execution of the process until a SEND command is directed to it from one of the listed processes. The associated parameter list provides a template for inserting the passed capabilities into the receiving processes' capability segment.

A calling process issues a SEND command, giving the process capability and the list of capabilities it wishes to pass. When the calling process cannot continue without the results from
the called process, it issues a WAIT, listing the processes it is waiting for, and the capabilities it expects returned.

All SEND commands directed to a process are queued by the kernel. Whenever a process desires input, it issues a WAIT command and lists parameters to be bound with capabilities. The kernel then takes the first entry in the queue which matches a process in the WAIT list, transfers the specified capabilities, and restarts the called process. It also gives the called process the name of the calling process. When the called process is finished, it issues a SEND command to return capabilities and unblock the calling process.

If a process wishes to wait for a message from any process, it can specify an "ALL" parameter on the WAIT operation rather than list every possible calling process. The most flexible implementation of WAIT would allow the process to supply a decision algorithm with a WAIT to determine what processes would satisfy the WAIT. This ability would allow processes to use the built-in queueing provided by the system. For instance, in the table-sharing example in Section II, the table manager could accept calls only from processes requesting available capabilities. Without a selective ability like this, the table manager will have to maintain queues of its own.

In the case of a process executing a serially-reusable procedure, an OPEN by some other process acts like a SEND command in that it satisfies the processes WAIT, even if the issuing process is not on the waiting processes list. It is necessary to do this since a serially-reusable procedure cannot know when some process is going to start communicating with it. In the
reentrant case, an OPEN starts a new process at the beginning of the specified procedure.

The form of the argument and parameter lists must still be specified. A process argument merely gives an index into the calling processes capability list, whereas a structure argument specifies the structure, and the read/write mask of access privileges to be given. A procedure argument specifies the entry in the calling processes capability list and the type of operations which the receiving process may perform on the procedure. Each parameter in a list specifies whether a process, procedure, or structure is expected, and where in the called processes capability list to put the received argument. In addition, a structure parameter can require the incoming argument to refer to the same structure which is currently in the capability entry specified in the parameter.

A process terminates execution by issuing a CLOSE command. It is assumed when a process terminates that it has returned all the capabilities it has received. In case it hasn't, and in the case when a process must be forcibly terminated because of an error, the sticky problem arises as to what to do with the left-over capabilities. The problem is important mainly for structure capabilities since there is only one capability for any structure element. The easiest solution would be to return capabilities to the processes from which they came by means of a SEND command with some indication of process termination. However, this does not take care of the possibility that a process receives a capability, gives it to some other process, and then expires before it gets the capability back
and returns it to where it came from. One could envision the kernel simulating a terminated process so that capabilities would be properly returned; however, the complexity of such a scheme seems too great to make it worthwhile. Hopefully, not too much harm would come from throwing away capabilities which are returned to a deactivated process.

A process uses the above SEND, WAIT mechanism to obtain all of its capabilities. When a process is first started, it holds only a capability for the loader process, and in the case of a serially-reusable procedure, a capability for reading and writing its procedure. The process can then get procedure capabilities from the loader, and structure and process capabilities from the kernel. After establishing communication with other processes it may give and receive capabilities as it wishes.

An important system function is debugging. The natural restriction is made that only serially-reusable procedures may be debugged. When a procedure is first brought into the system, it is treated as a data segment by the loader. When a serially-reusable procedure is first opened, it is given read/write access to itself. A process must give its consent to be debugged, and it does this by giving those read/write capabilities to some debugging process. It is also necessary for a debugging system to modify some of the status information of the process. To accomplish this, it seems necessary to define a new capability and a set of kernel operations which accept the capability and effect changes in a process's status. Then, just as with the read/write capability, a process would initially possess its own debugging capability which it could give away if it desired.
This section has stressed that the SEND and WAIT operations are sufficient for all system communication. This is true, however, in many situations an interrupt capability is desirable. In cases where an event occurs rarely, such as error conditions, it is much better to allow an interrupt when the event occurs than to check for every possible occurrence. Nothing said previously eliminates the possibility of adding interrupts to the system. One could very easily imagine a setup similar to PL/I where explicit calls and waits are the main communication device, but interrupts (in the form of ON conditions) are available. Naturally capabilities could be passed through the interrupt mechanism.

The last sections have described the system elements and their interactions in detail. The final section presents an example to show how these mechanisms would be used in practice.
V. An Example File System

This section describes an example file system which could be constructed under the organization described. The purpose is to show how the various mechanisms interact to produce a secure and efficient system.

We wish to provide an environment for user programs to access information on auxiliary storage devices. Each user process communicates with a file manager process for each file it accesses. All file manager processes talk to a single channel manager which queues channel operations. The channel manager then gives operations to various channel processes which execute the I/O operations.

We now consider the interfaces between these various processes. Whenever a user process wants to access a file, it opens a process on a file manager procedure. The file manager provides some access method by converting logical file operations to physical I/O operations. The user processes' first call on the file manager is used to open the file. The user program passes the file name and a data record area. The file manager checks to see if the user process is allowed to use the file, and which fields it is allowed data independent access. The file manager then checks to see if the structure for the data area is sufficient to mask the disallowed fields. If all these conditions are met, the file manager saves a record of which structure elements can be accessed data-independently and which cannot. Then the file manager gives the data area back to the user process, and
Calls on the file manager from then on are used to request the record sequentially from the file. On each call the user program must pass the same data area as on the first call.

Since the file manager has access to the entire record area at that time, it calls on the channel manager to get a record and put it in the data area. The file manager then returns only the capabilities for data independent fields.

If the user program wishes to read a field for which permission is data dependent, it must call the file manager to request access to the given field. The file manager will then check the data and decide whether access should be allowed. If it is, the file manager returns the field capability. Since all field capabilities must be returned to get the next record, the field can be accessed only in the record for which the permission was granted.

The file manager-channel manager interface is very simple from the file manager side. As it generates channel operations, it calls the channel manager, passing it capabilities for the data areas, and waiting for the channel manager to return.

However, the channel manager talks to many file managers, and hence has a more complicated calling structure. The channel manager queues the capabilities it receives from file managers in its capability segment. It does this by changing its parameter list so that successive argument lists are placed in different positions in the capability segment. Whenever the channel manager receives a call from a file manager, it queues the arguments and then searches its entire queue to see if there are channel operations which can be started. Whether it initiates channel action or not, it then waits for a message from either
a file manager or a channel.

If it receives a completion signal from a channel, it
removes the corresponding capabilities from its capability
queue by sending them back to the proper file manager. The
channel manager then completes its cycle by looking for more
channel operations to start. It then waits for more input.

It would be possible to let file managers call channel
operations directly and thus eliminate the queueing operation
in the channel manager. However, this example shows how the
automatic fifo queueing implemented in the kernel can be
replaced with some other discipline designed to optimize
channel usage. It is also safer to have only one procedure
communicating with I/O channels, since the channel manager can
screen the channel operations for errors.

Finally, the channel's view of the channel manager-
channel interface is very simple. The channel accepts an
operation and capabilities for the I/O areas, performs the
operation, returns the capabilities, and waits for another
operation.

This concludes the description of the example file system.
It was designed to show how a record could be moved from
auxiliary storage to the user program in one transfer, with
no reformatting, and in an entirely secure manner. As was
described in the introduction, the goal of the entire paper
was to structure operating systems so application systems
can be easily constructed which are modular, secure, and
efficient.
References


