Lecture Notes in Computer Science, 60,
"Operating Systems - An Advanced Course"
R. Bayer, R. M. Graham, and G. Seegmüller (eds.)
Springer-Verlag, 1978, pp. 99-208

CHAPTER 3.A.

J. H. Saltzer

Massachusetts Institute of Technology
Cambridge, Mass., USA

Naming and Binding of Objects

© 1978 by J. H. Saltzer. All rights reserved.

Overview

A property of a computer system that determines its ease of use and its range of applicability is the way it creates and manages the objects of computation. An important aspect of object management is the scheme by which a system names objects. Names for objects are required so that programs can refer to the objects, so that objects can be shared, and so that objects can be located at some future time. This chapter introduces several rather general concepts surrounding names, and then explores in depth their applicability to two naming structures commonly encountered inside computer systems: addressing architectures and file systems. It examines naming functions that are usually implemented (or desired) in these two areas, and some of the design tradeoffs encountered in a variety of contemporary computer systems. It ends with a brief discussion of some current research topics in the area of naming.

Glossary

bind	– to choose a specific lower-level implementation for a particular higher-level semantic construct. In the case of names, binding is choosing a mapping from a name to a particular object, usually identified by a lower-level name.
catalog	– an object consisting of a table of bindings between symbolic names and objects. A catalog is an example of a context (q.v.).
closure	– abstractly, the mechanism that connects an object that refers to other objects by name with the context in which those names are bound.
component	– an object that is contained by another object.
context	– a particular set of bindings of names to objects: a name is always interpreted relative to some context.
indirect entry	– in a naming network, an entry in a catalog that binds a name, instead of to an object, to the path name of some catalog entry elsewhere in the naming network.
library	– a shared catalog (or set of catalogs) that contains objects such as programs and data to which several users refer. A computer system usually has a system library, which contains commonly used programs.

limited context - a context in which only a few names can be expressed, and therefore names must be reused.

modular sharing - sharing of an object without the need to know of the implementation of the shared object. From the point of view of naming, modular sharing is sharing without need to know of the names used by the shared object.

name - in practice, a character- or bit-string identifier that is used to refer to an object on which computation is performed. Abstractly, an element of a context.

naming hierarchy - a naming network (q.v.) that is constrained to a tree-structured form.

naming network - a catalog system in which a catalog may contain the name of any object, including another catalog. An object is located by a multi-component path name (q.v.) relative to some working catalog (q.v.).

object - a software (or hardware) structure that is considered to be worthy of a distinct name.

path name - a multiple component name of an object in a naming network. Successive components of the path name are used to select entries in successive catalogs. The entry selected is taken as the catalog for use with the next component of the path name. For a given starting catalog, a given path name selects at most one object from the hierarchy.

reference name - the name used by one object (e.g., a program) to refer to another object.

resolve - to locate an object in a particular context, given its name.

root - the starting catalog of a naming hierarchy.

search - abstractly, to examine several contexts looking for one that can successfully resolve a name. In practice, the systematic examination of several catalogs of a naming network, looking for an entry that matches a reference name presented by some program. The catalogs examined might typically include a working catalog, a few other explicitly named catalogs. and a system library catalog.

shared object - 1) a single object that is a component of more than one other object. 2) an object that may be used by two or more different, parallel activities at the same time.

synonym - one of the multiple names for a single object permitted by some catalog implementations.

tree name - a multiple component name of an object in a naming hierarchy. The first component name is used to select an entry from a root catalog, which selected entry is used as the next catalog. Successive components of the tree name are used for selection in successively selected catalogs. A given tree name selects at most one object from the hierarchy.

unique identifier - a name, associated with an object at its creation, that differs from the corresponding name of every other object that has ever been created by this system.

unlimited context - a context in which names never have to be reused.

user-dependent binding - binding of names in a shared object to different components depending on the identity of the user of the shared object.

working catalog - in a naming network, a catalog relative to which a particular path name is expressed.

A. Introduction

1. Names in computer systems

Names are used in computer systems in many different ways. One of these ways is naming of the individual variables of a program, together with rules of scope and lifetime that apply to names used within a collection of programs that are constructed as a single unit. Another way names are used is in database management systems, which provide retrieval of answers to sophisticated queries for information permanently filed by name and by other attributes. These two areas are sufficiently specialized that they have labels of their own: the first is generally studied under the label "semantics of programming languages" and the second is studied under the label "database management".

Yet another use of names, somewhat less systematically studied, is the collection together of independently constructed programs and data structures to form subsystems, inclusion of one subsystem as a component of another, and use of individual programs, data structures, and other subsystems from public and semi-public libraries. Such activity is an important aspect of any programming project that builds on previous work or requires more than one programmer. In this activity, a systematic method of naming objects so that they may contain references to one another is essential. Programs must be able to call on other programs and utilize data objects by name, and data objects may need to contain cross references to other data objects or programs. If true modularity is to be achieved it is essential that it be possible to refer to another object knowing only its interface characteristics (for example, in the case of a procedure object, its name and the types of the arguments it expects) and without needing to know details of its internal implementation, such as to which other objects it refers. In particular, use of an object should not mean that the user of that object is thereafter constrained in the choice of names for other, unrelated objects. Although this goal seems obvious, it is surprisingly difficult to attain, and requires a systematic approach to naming.

Unfortunately, the need for systematic approaches to object naming has only recently been appreciated, since the arrival on the scene of systems with extensive user-contributed libraries and the potential ability easily to "plug together" programs and data structures of distinct origin*. As a result, the mechanisms available for study are fairly ad hoc "first cuts" at providing the necessary function, and a systematic semantics has not yet been developed†. In this chapter we identify those concepts and principles that appear useful in organizing a naming strategy, and illustrate with case studies of contemporary system naming schemes.

2. A model for the use of names

We shall approach names and binding from an object-oriented point of view: the computer system is seen as the manager of a variety of objects on

---

* Examples include the Compatible Time-Sharing System (CTSS) constructed at M.I.T. for the IBM 7090 computer, the Cambridge University System, the Honeywell Information Systems Inc. Multics, IBM's TSS/360, the TENEX system developed at Bolt, Beranek and Newman for the Digital Equipment PDP-1O computer, the M.I.T. Lincoln Laboratory's APEX system for the TX-2 computer, the University of California (at Berkeley) CAL system for the Control Data 64OO, and the Carnegie-Mellon EYORA system for a multiprocessor Digital Equipment Company PDP-11, among others.

† Early workers in this area included A. Holt, who was among the first to articulate the need for imposing structure on memory systems [Holt, 1961] and J. Iliffe, who proposed using indirect addressing (through "codewords") as a way of precisely controlling bindings [Iliffe and Jodeit, 1962]. J. Dennis identified the interactions among modularity, sharing, and naming in his arguments for segmented memory systems [Dennis, 1965]. A. Fraser explored the relation between naming in languages and naming in systems [Fraser, 1971].

which computation occurs. An active entity that we shall call a program interpreter* performs the computation on these objects. Objects may be simply arrays of bits, commonly known as segments, or they may be more highly structured, for example containing other objects as components. There are two ways to arrange for one object to contain another as a component: a copy of the component object can be created and included in the containing object (containment by value) or a name for the component object may be included in the containing object (containment by name).

In containment by value, an object would be required to physically enclose copies of every object that it contains. This scheme is inadequate because it does not permit two objects to share a component object whose value changes. Consider, for example, an object that is a procedure that calculates the current Dow-Jones stock price average. Assume that this procedure uses as a component some data base of current stock prices. Assume also that there is another procedure object that makes changes to this data base to keep it current. Both procedure objects must contain the data base object. With containment by value, each procedure object must include a copy of the data base. Then, however, changes made by one procedure to its copy will not affect the other copy, and the second procedure can never see the changes.

A fundamental purpose for a name, then, is to accomplish sharing, and the second scheme is to include a name for a component object in a containing object. When names are used, some way is then needed to associate the names with particular objects. As we shall see, it is common for several names to be associated with the same object, and for one name to be associated with different objects for different purposes. In examining these various possibilities, we shall discover that they all fit

* In various systems, the terms "execution point", "processor", "process", "virtual processor", "task", and "activity", have been used for this active entity. For the present discussion we shall adopt the single term "program interpreter" for its mnemonic value, and assume that there are potentially many active sites of computation (and thus many active program interpreters) at the same time, as in typical time-sharing and multiprocessing systems.

into one abstract pattern. This abstract pattern for containment by naming is as follows: a context is a partial mapping from some names into some objects of the system*. To employ a component object, a name is chosen for the object, a context that maps that name into that component object is identified or created, the name is included in the containing object, and the context is associated with the containing object. At some later time, when the containing object is the target of some computation, the program interpreter performing the computation may need to refer to the component object. It accomplishes this reference by looking up the name in the associated context. Arranging that a context shall map a name into an object is called binding that name to that object in that context. Using a context to locate an object from a name is called resolving that name in that context. Figure 1 illustrates this pattern.

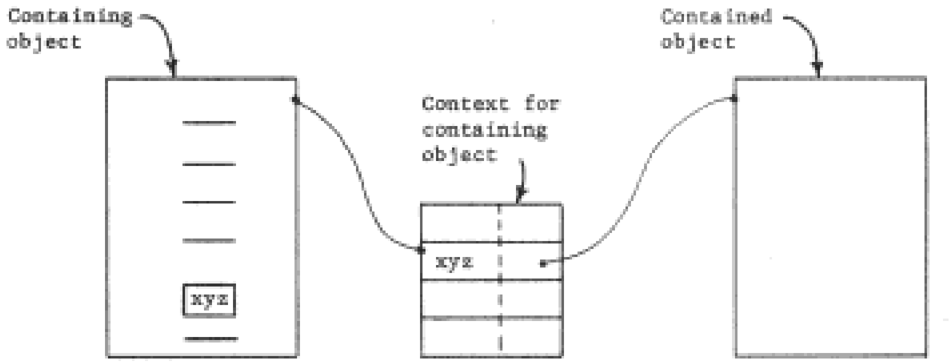


Figure 1 -- Pattern for use of names. The containing object includes a use of the name "xyz". The containing object is somehow associated with a context. The context contains a mapping between the name "xyz" and enough information to get to the contained object. Because the contained object has not been copied into the containing

* In the study of programming language semantics, the terms universe of discourse, context, and environment are used for a concept closely related to the one we label context. Usually, the programming language concept is a mapping with the possibility of duplicate names, a stack or tree structure, ant a set of rules for searching for the correct mapping within the environment, Our concept of context is simpler, being restricted to an unstructured mapping without duplicates. The names we deal with in this chapter correspond to free variables of programming language semantics, and we shall examine a variety of techniques for binding those free variables. Curiously, we use a simpler concept because in systems we shall encounter a less systematic world of naming then in programming languages.

object, it is possible for some third object also to contain this object; thus sharing can occur.

In examining figure 1, two further issues are apparent: 1) the context must include, either by value or by name, the contained object; 2) the containing object must be associated with a context. Figure 2 illustrates the handling of both these issues in the familiar example of a location-addressed memory system in a simple computer that has no sophisticated addressing machinery at all. Electrical wiring in effect places a copy of the contained object in its context and also places a copy of the context in the containing object. (In both cases, the "copy" is the only copy, the original.)

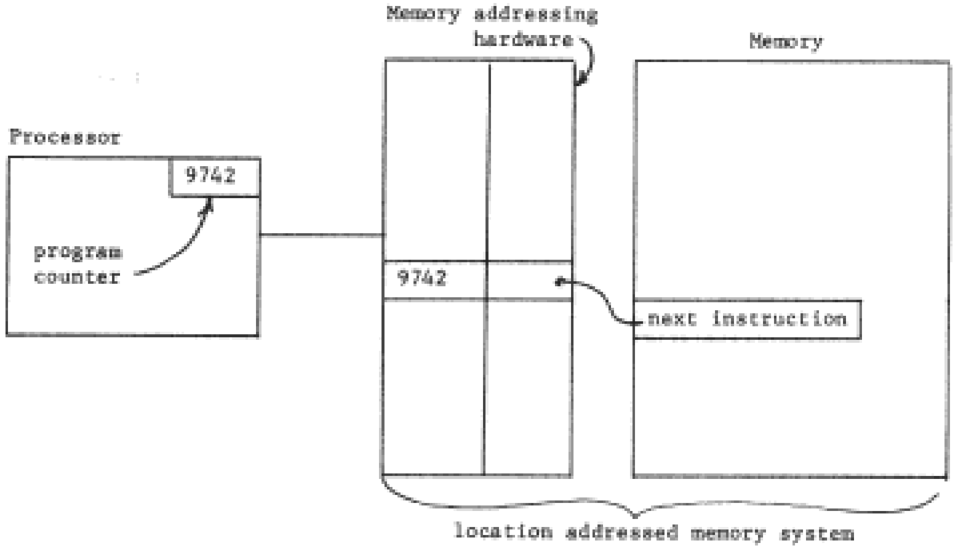


Figure 2 -- Instruction retrieval as an example of naming. In this simple computer the processor program counter names the next instruction to be interpreted. The processor is associated with a context, the memory addressing hardware, by means of an electrical cable. The context maps the name "9742" into the physical location in memory of some particular word of information, again using electrical cable to form the association. (Note that, except in the simplest microprocessors, one does not usually encounter a processor that actually uses such a primitive scheme.)

The alternative approach for handling the connection between the context and the contained object is for the context to refer to the contained object with another name, a lower-level one. This lower level name must then be resolved in yet another context. Figure 3 provides an example in which an interpreter's internal symbol table is the first, higher-level context, and the location-addressed memory of figure 2 provides the lower-level context. A more elaborate example could be constructed, with several levels of names and contexts, but the number of contexts must be finite: there must always be some context that contains its objects by value (as did the location-addressed hardware memory) rather than naming them in still another context. Further, since a goal of introducing names was sharing, and thus avoiding multiple copies of objects, each object ultimately must be contained by value in one and only one context.

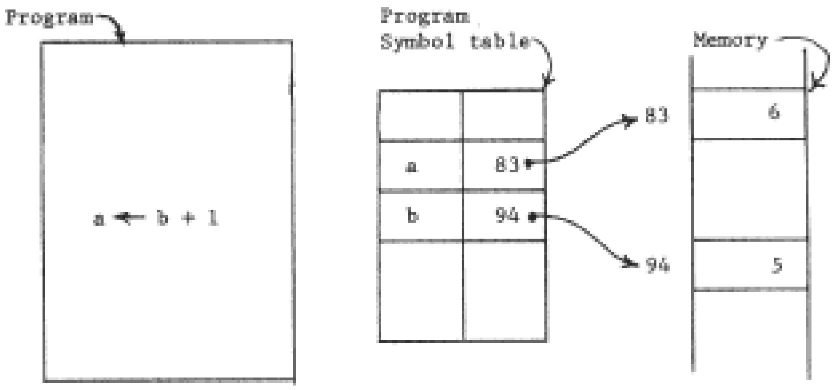


Figure 3 -- A two-level naming example. An interpreter executes a program containing the names "a" and "b". The interpreter resolves these names using the context represented by a symbol table that maps the names "a" and "b" into lower level names, which are addresses in the memory. These lower level names might be-resolved as in figure 2 .

Returning to figure 1, it is also necessary for a containing object to be associated with its context. If the context happens to be implemented as an object in its own right (a common strategy) this association may be provided by creating a new object that contains (using either lower-level names or copies, as appropriate) both the original containing object and the appropriate context as components. A mechanism that exists for the purpose of associating some name-containing with its context is known as closure, and an object that performs this function is a closure object. In many cases, the closure is implicitly supplied by the program interpreter rather than being implemented as an explicit object. For example, in figure

3, the interpreter automatically uses the program's symbol table (which might be a data object contained in the interpreter itself) as a context. For another example, in many systems the user's catalog is an automatically provided context for file names. Yet another example is the context associated with each virtual processor in a system for resolving the addresses of words in memory; this context is called the virtual processor's address space, and in a paged system is represented by a page map. The concept of a closure is fundamental to naming, but explicit closure objects will not appear to be of much interest until we consider the problem of changing contexts when calling from one procedure to another.

3. Problems in the use of names

This simple model for the use of names seems straightforward in that it allows objects to be shared. However, there are several more objectives usually wanted in a naming system: modularity of sharing, multiple contexts, and user-dependent bindings. Failure to meet one or more of these objectives shows up as an awkward problem. These troubles may arise from deliberate design compromises or from unintentional design omissions.

One common problem arises if the wrong implicit context is supplied by the program interpreter. This problem can occur if the interpreter is dealing with several objects and does not fully implement closures. Such an interpreter may not keep distinct the several contexts, or may choose among available contexts on some basis other than the object that contained the name. For example, file names in many systems are resolved relative to a "current working catalog"; yet often the working catalog is a static concept, unrelated to the identity of the object making the reference.

Names permit sharing, but not always in the most desirable way. If use of a shared object requires that the user know about the names of the objects that the shared object uses (for example, by avoiding use of those names) we have not accomplished the goal of modularity. We shall use the term modular sharing to describe the (desirable) situation in which a shared object can be used without any knowledge whatsoever of the names of the objects it uses.

Lack of modular sharing can show up as a problem of name conflict. in which for some reason it seems necessary to bind the same name to two or more objects in one context. This situation often occurs when putting together two independently conceived sets of programs in a system that does not provide modular sharing. Name conflict is a serious problem since it requires changing some of the uses of the conflicting names. Making such changes can be awkward or difficult, since the authors of the original programs are not necessarily available to locate, understand, and change the uses of the conflicting names.

Sharing should also be controllable, in the following apparently curious way: different users of an object (that is, users with distinct, simultaneously active program interpreters) should be able to provide private user-dependent bindings for some of its components. However, one user's private bindings should not affect other users of the shared object. The most common example of a user-dependent binding is the association between arguments to a function and its formal parameters, but in modular systems other examples abound also. When a single subprogram is used in different applications, it may be appropriate for that subprogram to have a different context for each application. The different contexts would be used to resolve the same set of names, but some of those names might resolve to different objects. There are three common situations in which the users of an object might need different contexts for different applications:

1. When the object is a procedure, and its operation requires memory private to its user. The storage place for the private memory can be conveniently handled by creating a private context for this combination of user and program and arranging that this private context be used whenever the program serves this user. In the private context, the program's name for the memory area is bound to a storage object that is private to the user. A concrete example might be the

storage area used as a buffer by a shared interactive text editor in a word processing system.

2. When a programmer makes a change to one part of a large subsystem. and wants to run it together with the unchanged parts of the subsystem. For example, suppose a statistics subsystem is available that uses as a component a library math routine. One user of the statistics subsystem has a trouble, which he traces to inaccuracy in the math routine. He develops a specialized version of the math routine that is sufficiently accurate for his use, and wants to have it used whenever he invokes the statistics subsystem. Copying the entire subsystem is one way to proceed, but that approach does not take advantage of sharing, and in cases where writeable data is involved may produce the wrong result. An alternative is to identify those contexts that refer to the modified part, and create special versions that refer to the new part instead of the original.
3. Two multimodule subsystems (for example a theatre ticket and an airline reservation system) might differ in only one or two modules (for example the overbooking policy algorithm). Yet it may be desirable to maintain only one copy of the common modules. To handle those cases where a common module refers to a non-common module by name, user-dependent bindings are required.

In each of these situations some provision must be made for a name-using object to be associated with different contexts at different times, depending on the identity of the user. This provision is usually made by allowing the establishment of several closures, each of which associates the name-using object with a different context, and providing some scheme to make sure that the name interpreter knows which closure to use for each different user.

Yet another problem in using names is <u>unstable bindings</u>; that is, bindings that change unpredictably between definition and use. For example, file system catalogs often serve as contexts, and usually those catalogs permit names to be deleted or changed. Employing one object in another by using a name and a changeable context can make it impossible to ensure that when the time comes to use that name and context the desired object will be obtained.

Sometimes, these naming troubles arise because a system uses a single compromise mechanism to accomplish naming and also some other objective such as economy, resource management, or protection. A common example is a limitation on the number of names that can be resolved by a single context. Thus, the limited size of the "address space" of a location-addressed memory system often restricts which subprograms can be employed together in forming a program, producing non-modular sharing, name conflicts, or sometimes both. For example, some operating systems allow several users to share a text editor or compiler by assigning those programs fixed locations, the same in every user's address space. In such a system if a single user wants to construct a subsystem that uses both the editor and the compiler as components, they must have been assigned different fixed locations. If more than a handful of shared programs are required, name conflict will occur, and restrictions must be placed on which sets of programs any one user can invoke as part of a single subsystem. What is going wrong is simply that with a limited number of names available, one cannot make the universally usable name assignment needed to accomplish modular sharing.

4. Some examples of existing naming systems

Most existing systems exhibit one or more of the problems of the previous section. Two types of naming systems are commonly encountered systems growing out of a programming language, and operating systems with their own, language-independent naming systems.

FORTRAN language systems are typical of the first type [IBM, 1961]. For purpose of discussion here, separately translated subprograms play the part of objects*. Each subprogram is given a name by its programmer, and may contain the names of other subprograms that it calls. When a set of

---

* The names of individual FORTRAN variables and arrays are handled by the compiler using another, distinct naming system.

subprograms is put together (an activity known as "loading"), a single, universal context is created associating each subprogram with its name. Uses of names by the subprograms of the set, for example where one subprogram calls another by name, are then resolved in this universal context. The creator of the set must be careful that all of the objects named in an included object are also included in the set. The set of loaded subprograms, linked together, is called a "program".

Because a universal context is used for all subprograms loaded together, two subprograms having the same name are incompatible. The common manifestation of this incompatibility is name conflicts discovered when two collections of subprograms, independently conceived and created, are brought together to be part of a single program.

Loading subprograms involves making copies of them. As discussed in the previous section, this copying precludes sharing of modifiable data among distinct programs. Some systems provide for successive programs to utilize data from previous programs by leaving the data in some fixed part of memory. Such successive programs then need to agree on the names for (positions of) the common data.

Loading a set of subprograms does not create another subprogram. Instead, the resulting program is of a different form, not acceptable input to a further loading operation, and not nameable. This change of form during loading constrains the use of modularity, since a previously loaded program cannot be named, and thus cannot be contained in another program being created by the loader.

In contrast with FORTRAN, APL language systems give each programmer a single context for resolving both APL function names and also all the individual variable names used in all the APL functions [Falkoff and Iverson, 1968]. This single context is called the programmer's "workspace". APL functions are loaded into the workspace when they are created, or when they are copied from the workspace of another programmer.

Problems similar to those of FOKTRAN arise in APL: name conflicts lead to incompatibility, and in the case of APL, name conflicts extend to the level of individual variables. The programmer must explicitly supply all contained objects. Copying objects from other workspaces precludes employing shared writeable objects.

In an attempt to reduce the frequency of name conflicts, APL provides some relief from the single context constraint by allowing functions to declare private variables and placing these variables in a name-binding stack. thus creating a structured naming environment. Stacking has the effect that the names in a workspace may be dynamically re-bound, leading to unreliable name resolutions. When a function is entered, the names of any variables or other functions defined in that function are temporarily (for the life of that function invocation) added to the workspace stack, and if they conflict with names already defined they temporarily override all earlier mappings of those names. If the function then invokes a second function that uses one of the remapped names, the second function will use the first function's local data. The exact behavior of a function may therefore depend upon what local data has been created by the invoking function, or its invoker, and so forth. This strategy, named "call-chain name resolution," is a good example of sharing (any one function may be used, by name. by many other functions) but without modularity in the use of names.

Consider the problem faced by a team of three programmers creating a set of three APL functions. One programmer develops function A, which invokes both B and C. The second programmer independently writes function B, which itself invokes C. The third programmer writes function C. The second programmer finds that a safe choice of names for private temporary variables of B is impossible without knowing what variable names the other two programmers are using for communication. If the programmer of B names a variable "X" and declares it local to B, that use of the name "X" may disrupt communication between procedures A and C in the following scenario: suppose the other programmers happened to use the name "X" for communication. B's variable "X" lies along the call chain to C on some--but not all--invocations of C. Each programmer must know the list of all names used for intermodule communication by the others, in violation of the definition of modular sharing.

LISP systems have extremely flexible naming facilities, but the way they are conventionally used is very similar to APL systems [Moses, 1970]. Each user has a single context for use by all LISP functions. Functions of other users must be copied into the context of an employing function. Call-chain name resolution is used.

LISP is usually implemented with an internal cell-naming mechanism that eliminates naming problems within the scope of a single user's set of functions. The atoms, functions, and data of a single user are all represented as objects with unique cell names. When an object is created, it is bound to this cell name in a single context private to the user. (The implementation of this mechanism varies among LISP systems. It usually is built on operating system main-memory addressing mechanisms and a garbage collector or compactor.) These cell names usually cannot be re-bound, although they are a scarce resource and may be reallocated if they become unbound. Cell names are used by LISP objects to achieve reliable references to other LISP objects.

LISP permits modular sharing, through explicit creation of closure objects, comprising a function and the current call-chain context. When such a function is invoked, the LISP interpreter resolves names appearing in the function by using its associated context. The objects and data with bindings in the context contained in the closure are named with internal names. Internal names are also used by the closure to name the function and the context.

In many LISP systems the size of the name space of internal names is small enough that it can be exhausted relatively quickly by even the objects of a single application program. Thus potential sets of closures can be incompatible because they would together exhaust the internal name space.

As far as name conflicts are concerned, however, two closures are always compatible. Closures avoid dynamic call-chain name resolution. So within the confines of a single user's functions and data, LISP permits modular sharing through exclusive, careful use of closures*.

Most language systems, including those just discussed, have been designed to aid the single programmer in creating programs in isolation. It is only secondarily that they have been concerned with interactions among programmers in the creation of programs. A common form of response to this latter concern is to create a "library system". For example, the FORTRAN Monitor System for the IBM 709 provided an implicit universal context in the form of a library, which was a collection of subprograms with published names [IBM,1961]. If, after loading a set of programs, the loader discovered that one or more names was unresolvable in the context so far developed, it searched the library for subprograms with the missing names, and added them to the set being loaded. These library subprograms might themselves refer to other library subprograms by name, inducing a further library search. This system exhibited two kinds of problems. First, if a user forgot to include a subprogram, the automatic library search might discover a library subprogram that accidentally had the same name and include it, typically with disastrous results. Second, if a FORTRAN subprogram intentionally called a library subprogram, it was in principle necessary to review the lists of all subprograms that that library subprogram called, all the subprograms they called, and so on, to be sure that conflicts with names of the user's other subprograms did not occur. (Both of these problems were usually kept under control by publishing the list of names of all subprograms in the libraries, and warning users not to choose names in that list for their own subprograms.)

A more elaborate form of response to the need for interaction among programmers is to develop a "file system" that can be used to create catalogs of permanent name-object bindings. Names used in objects are resolved automatically using as a context one of the catalogs of the file

* This particular discipline is not a common one among LISP programmers, however. Closures are typically used only in cases where a function is to be passed or returned as an argument, and call-chain name resolution would likely lead to a mistake when that function is later used.

system. The names used to indicate files are consequently called "file names".

However, because all programmers use the same file system, conflict over the use of file names can occur. Therefore it is common to partition the space of file-names, giving part to each programmer. This partition is sometimes accomplished by assigning unique names to programmers and requiring that the first part of each file name be the name of the programmer choosing that file name.

On the other hand, so that programs can be of use to more than one programmer, file names appearing within a program and indicating objects that are closely related may be allowed to omit the programmer's name. This omission requires an additional sophistication of the name resolution mechanisms of the file system, which in turn must be used with care. For example, if an abbreviated name is passed as a parameter to a program created by another programmer, the name resolution mechanisms of the file system may incorrectly extend it when generating the full name of the desired object. Mistakes in extending abbreviated names are a common source of troubles in achieving reliable naming schemes.

As a programmer uses names in his partition of the file names, he may eventually find that he has already used all the mnemonically satisfying names. This leads to a desire for further subdivision and structuring of the space of file names, supported by additional conventions to name the partitions*. Permitting more sophisticated abbreviations then leads to more sophisticated mechanisms for extending those abbreviations into full file names This in turn leads to even more difficulty in guaranteeing reliable naming.

Many systems permit re-binding of a name in the file system. However, one result of employing the objects of others is that the creator of an object may have no idea of whether or not that object is still named by

* For example, Multics provided a tree-structured file naming system [Bensoussan, 1972].

other objects in the system Systems that do not police re-binding are common; in such systems, relying on file names can lead to errors.

The preceding review makes it sound as though systems of the kinds mentioned have severe problems. In actual fact, there exist such systems that serve sizable communities and receive extensive daily use One reason is that communities tend to adopt protocols and conventions for system usage that help programmers to avoid trouble. A second reason is that much of the use of file systems is interactive use by humans, in which case ambiguity can often be quickly resolved by asking a question.

In the remainder of this chapter, we shall examine the issues surrounding naming in more detail, and look at some strategies that provide some hope of supporting modular sharing, at least so far as name-binding is concerned.

5. The need for names with different properties

A single object may have many kinds of names, appearing in different contexts, and more than one of some kinds This multiple naming effect arises from two sets of functional requirements:

1) Human versus computational use:

   a) Names intended for use by human beings (such as file names) should be (within limits) arbitrary-length character strings. They must be mnemonically useful, and therefore they are usually chosen by a human, rather than by the computer system. Ambiguity in resolving human-oriented names is often acceptable, since in interactive systems, the person using the name can be queried to resolve ambiguities.

   b) Names intended for computational use (such as the internal representation of pointer variables) need not have mnemonic value, but must be unambiguously resolvable. They are usually chosen by the system according to some algorithm that helps avoid ambiguity. In addition, when speed and space are considered, design optimization leads to a need for names that are fixed length. fairly short, strings of bits (for example, memory addresses).

2) Local versus universal names:

   a) In a system with multiple users, every object must have a distinct unique identity. To go with this unique identity, there is often some form of universal name, resolvable in some universal context.

   b) Any individual user or program needs to be able to refer to objects of current interest with names that may have been chosen in advance without knowledge of the universal names. Modifying (and recompiling) the program to use the universal name for the object is sometimes an acceptable alternative, but it may also be awkward or impossible. In addition, for convenience, it is frequently useful to be able to assign temporary, shorthand names to objects whose universal names are unwieldy. Local names must, of course, be resolved in an appropriate local context.

Considering both of these sets of requirements at once leads to four combinations, most of which are useful. Further, since an object may be referred to by many other objects, it may have several different local names. As one might expect, most systems do not provide for four styles of names for every object. Instead, compromise forms are pressed into service for several functions. These compromises are often the root cause of the naming troubles mentioned in the previous section.

A further complication, especially in names intended for human consumption, is that one may need to have synonyms. A synonym is defined as two names in a single context that are bound to the same object or lower level name*. For example, two universal names of a new PL/I compiler might be "library.languages.pl1" and "library.languages.new-pl1", with the intent being that if a call to either of those names occurs, the same program is

* Note that when a higher-level name is bound, through a context, to a lower-level name, the higher and lower level names are not considered synonyms.

to be used. Synonyms are often useful when two previously distinct contexts are combined for some reason.

Finally, a distinction must be made between two kinds of naming contexts: unlimited, and limited. In an unlimited naming context, every name assigned can be different from every other name that has ever been or ever will be assigned in that context. Character string names are usually from unlimited naming contexts, as are unique identifiers, by definition. In a limited context the names themselves are a scarce resource that must be allocated and, most importantly, must be reused. Addresses in a location addressed physical memory system, processor register numbers, and indexes of entries in a fixed size table are examples of names from a limited context.

One usually speaks of <u>creating</u> or <u>destroying</u> an object that is named in an unlimited context, while speaking of <u>allocating</u> or <u>deallocating</u> an object that is named in a limited context*. Names for a limited context are usually chosen from a compact set of integers, and this compactness property can be exploited to provide a rapid, hardware-assisted implementation of name resolution, using the names as indexes into an array.

Because of the simplicity of implementation of limited contexts, the innermost layers of most systems use them in preference to unlimited contexts. Those inner layers can then be designed to implement sufficient function, such as a very large virtual memory, that some intermediate layer can implement an unlimited context for use of outer layers and user applications.

6. <u>Plan of study</u>

Up to this point, we have seen a general pattern for the use of names, a series of examples of systems with various kinds of troubles in

---

* Both the name for the object and resources for its representation may be allocated (or deallocated) at the same time, but these two allocation (or deallocation) operations should be kept conceptually distinct.

their naming strategies, and a variety of other considerations surrounding the use of names in computer systems. In the remainder of this chapter, we shall develop step-by-step two related, comprehensive naming systems, one for use by programs in referring to the objects they compute with (an addressing architecture,) and one for use by humans interactively directing the course of the programs they operate (a file system). We shall explore the way in which these two model naming systems interact, and some implementation considerations that typically affect naming systems in practice. Finally, we shall briefly describe some research problems regarding naming in distributed computer systems.

B. An architecture for addressing shared objects

An addressing architecture is an example of a naming system using computation-interpretable names, in which the program interpreter is usually a hardware processor. Although we shall see points of contact between these machine-oriented names ant the corresponding human-oriented character string names, those contacts are incidental to the primary purpose of the addressing architecture, which is to allow flexible name resolution at high speed. Typically, the interpretation of a single machine instruction will require one name resolution to identify which instruction should be performed and one or more name resolutions to identify the operands of the instruction, so the addressing architecture must resolve names as rapidly as the hardware processor interprets instructions in order not to become a severe bottleneck.

Figure 2 illustrated an ordinary location-addressed memory system. Sharing is superficially straightforward in a location-addressed memory system: an object is named by its location, and that name can be embedded in any number of other objects. However, using physical locations as names guarantees that the context is limited. If there exist more objects than will fit in memory at once, names must be reused, and reuse of names can lead to name conflict. Further, since selective substitution requires multiple contexts, the single context of a location-addressed memory system appears inherently inadequate. To solve these problems, we must develop a more hospitable (and unfortunately more elaborate) addressing architecture.

The first step in this development is to interpose an object map between the processor and the location-addressed memory system, as in figure 4, producing a structured memory system. Physical addresses of the location-addressed memory system appear only in the object map, and the processor must use logical names--object numbers--to refer to stored objects. The object map acts as an automatically supplied context for resolving object numbers provided by the processor; it resolves these object numbers into addresses in the location-addressed memory to which it is directly attached. We assume that this one object map provides a universal context for all programs, all users, and all real or virtual processors of the system, and that the range of values is large enough to provide an unlimited context; the object numbers are thus unique identifiers. To simplify future figures, we redraw figure 4 as in figure 5, with the unique identifiers directly labeling the objects to which they are bound. We can now notice that the procedure has embedded within itself the name of its data object; the context in which this name is interpreted is the same universal context in which the processor's instruction address is interpreted, namely the object map of the structured memory system. We shall occasionally describe this name embedded in the procedure as an outward reference, to distinguish it from references by the procedure to itself.

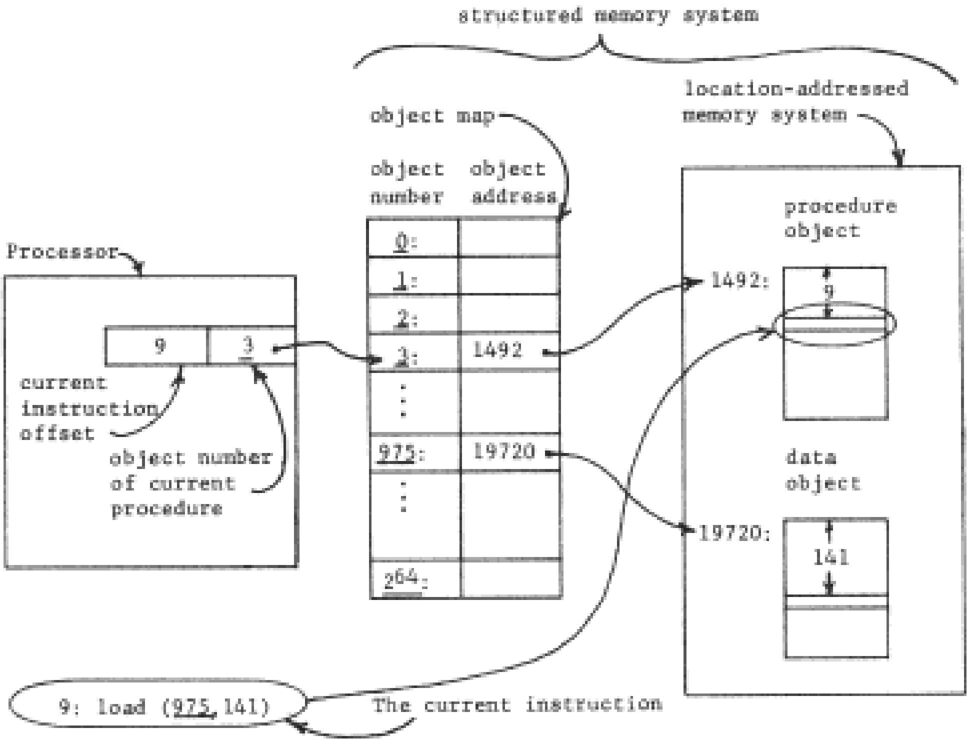


Figure 4 -- The structured memory system. The processor is executing instruction 9 of procedure object 3, located at address 1501 in the memory. That instruction refers to location 141 of data object 975, located at address 19861 in the memory. The columns of the object map relate the object number to the physical address. In a practical implementation, one might add more columns to the object map to hold further information about the object. For example, for a segment object, one might store the length of the segment, and include checking hardware to insure that all data offsets are of values within the length of the segment.

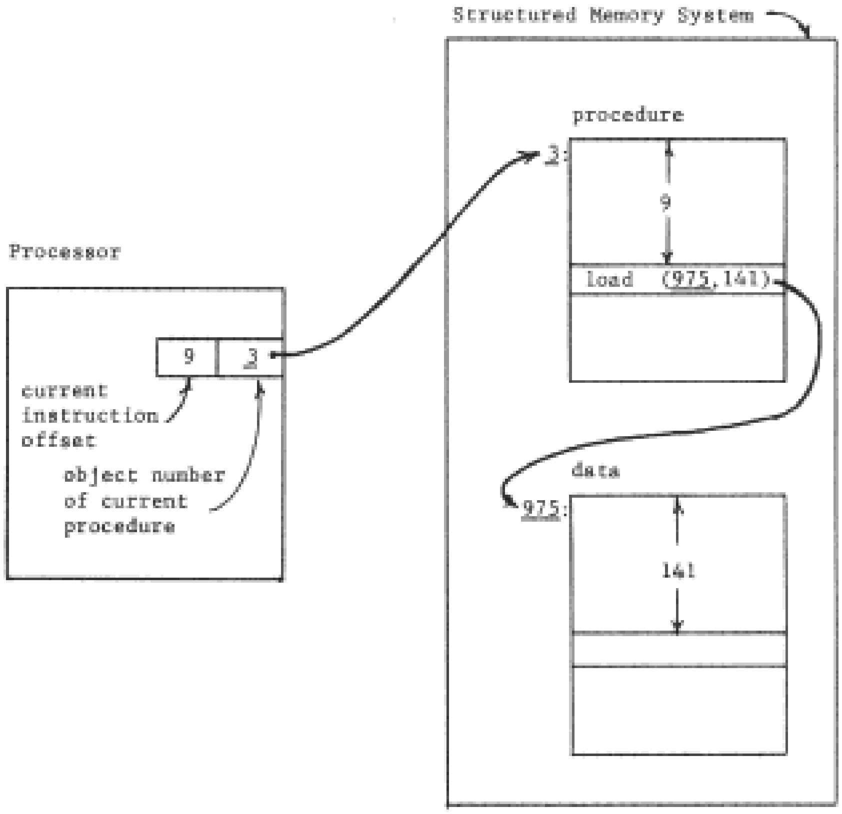


Figure 5 -- The structured memory system of figure 4 with the object map assumed and therefore not shown. Note that the procedure object contains the name of the data object, 975. To emphasize the existence of the context that the now-hidden object map implements, all object numbers in this and the following figures are italicized (underlined).

Since the structured memory system provides an unlimited context, the procedure can contain the name of the data object without knowing in advance anything about the names of objects contained in the data. Further, if the location-addressed memory system is small, one set of programs and data can be placed in it at one time, and another set later, with some objects in common but without worry about name conflict. We have provided for modular sharing, though with a minor constraint. The procedure cannot choose its own name for the data object, it must instead use the unique identifier for the data object previously assigned by the system. Table I will be used as a way of recording our progress toward a more flexible addressing architecture. Its first two columns indicate the effect of adding an object map that allows unique identifiers as object names*. Its later columns and lower rows are the subjects of the next few sections.

Table I -- Naming objectives and the addressing architecture

Naming Objective	architectural feature					
	Location Addressed Memory System	Structured Memory System	SMS with pointer register context	SMS with context objects	SMS with closure table	SMS with closures and name source register
sharing of components	yes	yes	yes	yes	yes	yes
sharing of component objects without knowing subcomponents	no	yes	yes	yes	yes	yes
sharing procedure components with user-dependent binding of subcomponent names	no	no	yes	yes	yes	yes
ability to easily change contexts on procedure calls	no	no	no	yes	yes	yes
automatic change of context on procedure calls	no	no	no	no	yes	yes
sharing data objects with user-dependent binding of subcomponent names	no	no	no	no	no	yes

* Although unique-identifier object maps have been proposed [Radin and Schneider, 1976; Redell, 1974] there seem to be formidable problems in implementing unlimited contexts in hardware (a very large map may be needed, thereby producing interactions with multilevel memory management) and most real object addressing systems provide limited contexts that are just large enough to allow short-lived computations to act as though the context were unlimited. Multics [Bensoussan et al., 1972] was a typical example.

1. User-dependent bindings and multiple naming contexts

As our system stands, every object that uses names is required to use this single universal context. Although this shared context would appear superficially to be an ideal support for sharing of objects, it goes too far; it is difficult to avoid sharing. For example, suppose that the data object of figure 5 should be private to the user of the program, and there are two users of the same program. One approach would be to make a copy of the procedure, which copy would then have a different object number, and modify the place in the copy where it refers to the data object, putting there the object number of a second data object. From the point of view of modularity, this last step seems particularly disturbing since it requires modifying a program in order to use it. What is needed is a user-dependent binding between the name used by the program and the private object.

Improvement on this scheme requires that we somehow provide a naming context for the procedure that can be different for different users. An obvious approach is to give each user a separate processor, and then to make the context depend on which processor is in use*. This approach leads to figure 6, in which two processors are shown, and to provide a per-user context each processor has been outfitted with an array of pointer registers, each of which can hold one object number. The name-interpreting mechanics of the processor must be more elaborate now, since interpretation of a name will involve going through two layers of contexts.

* In the usual case that there are not enough real hardware processors to go around, one would implement virtual processors in their place. This discussion will continue to use the term "processor" for the program interpreter, since from the point of view of naming, it is of no concern whether a processor is virtual or real.

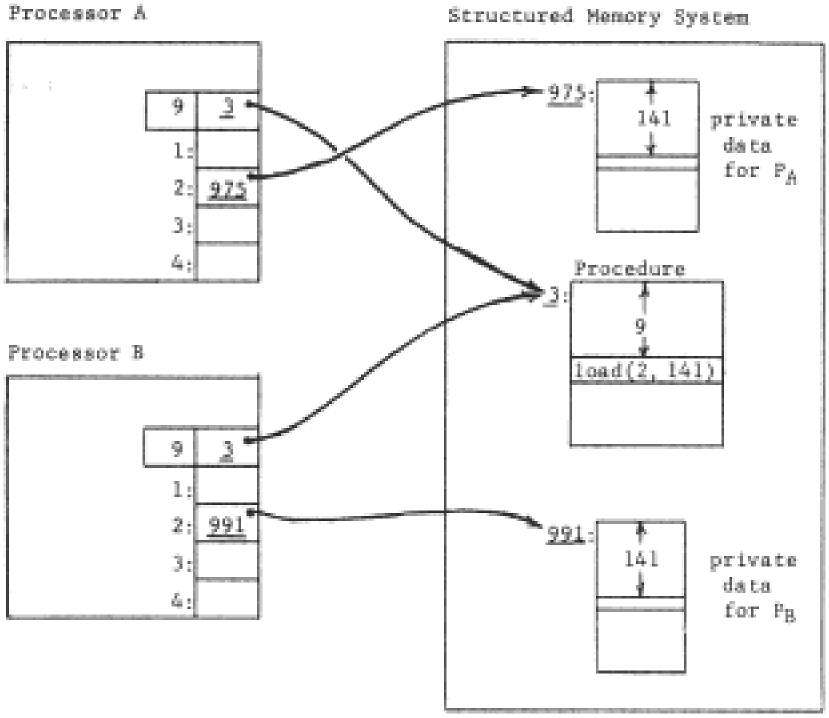


Figure 6 -- Addition of pointer registers to the processor, to permit a single procedure to have a processor-dependent naming context.

This more elaborate name interpretation goes as follows: the pointer registers are numbered, and the processor interprets an operand reference, which used to be an object number, as a register number instead. The register number names a register, whose contents are taken by the processor to be an object number in the context of the structured memory system.

Thus, in figure 6, the current instruction now reads "load (2,141)" with the intent that the name "2" be resolved in the context of the processor registers. If processor A resolves "2", it finds object number <u>975</u>, which is the name of the desired object in the context of the structured memory system. Thus when processor A interprets the operand reference of the instruction (<u>3</u>,9) it will obtain the 141st item of object <u>975</u>. Similarly, when processor B interprets the same operand reference, it will obtain the 141st item of object <u>991</u>.

We have thus arranged that a procedure can be shared without the onerous requirement that everything to which the procedure refers must also be shared–we are permitting selective user-dependent bindings for objects contained by procedure objects.

The binding of object numbers to particular objects was provided by the structured memory system, which chose an object number for each newly created object and returned that object number to the requester as an output value. We have not yet described any systematic way of binding register numbers to object numbers. Put more bluntly, how did register two get loaded with the appropriate object number, different in the two processors? Suppose the procedure were created by a compiler. The choice that register name "2" should be used would have been made by the compiler so in accordance with the standard pattern for using names, the compiler should also provide for bindings of that name to the correct object In this case, it might do so as in figure 7, by producing as output not only the procedure object containing the "load" instruction, but also the necessary context binding information. If the procedure uses several pointer registers, the context binding information should describe how to set up each of the needed registers. As shown, the context binding information is a high level language description of the context needed by the procedure; this high level description must be reduced to a machine understandable version of the context for the program to run. The combination of the program and its context binding information is properly viewed as a prototype of a closure*.

The same technique can be used by the compiler to arrange for the procedure to access a shared data object, too. Suppose, for example, the compiler determines from declarations of the program that variable b is to be private (that is, per-processor) while variable b is to be shared by all users of this procedure. In that case it might create, at compilation time, an object to hold variable a (say in location 5 of that object) and include

* In the terms of programming language semantics, the compiler is a function that produces as its output value another function: this output function contains free variables planted in it by the compiler and that should be bound in a way specified by the compiler. Thus the compiler should return not a function, but a closure that provides for binding of the free variables of the enclosed function.

its object number with the context binding information as in figure 8 . The result would be the pattern of reference shown in figure 9.

Translation from the high level context description of figure 8 to the register context of figure 9 is accomplished by a program known here as a <u>context initializer*</u> and most such programs permit a wider variety of object interlinking possibilities than illustrated in figure 8. Before getting into that subject, we should first consider three elaborations on the naming conventions already described.

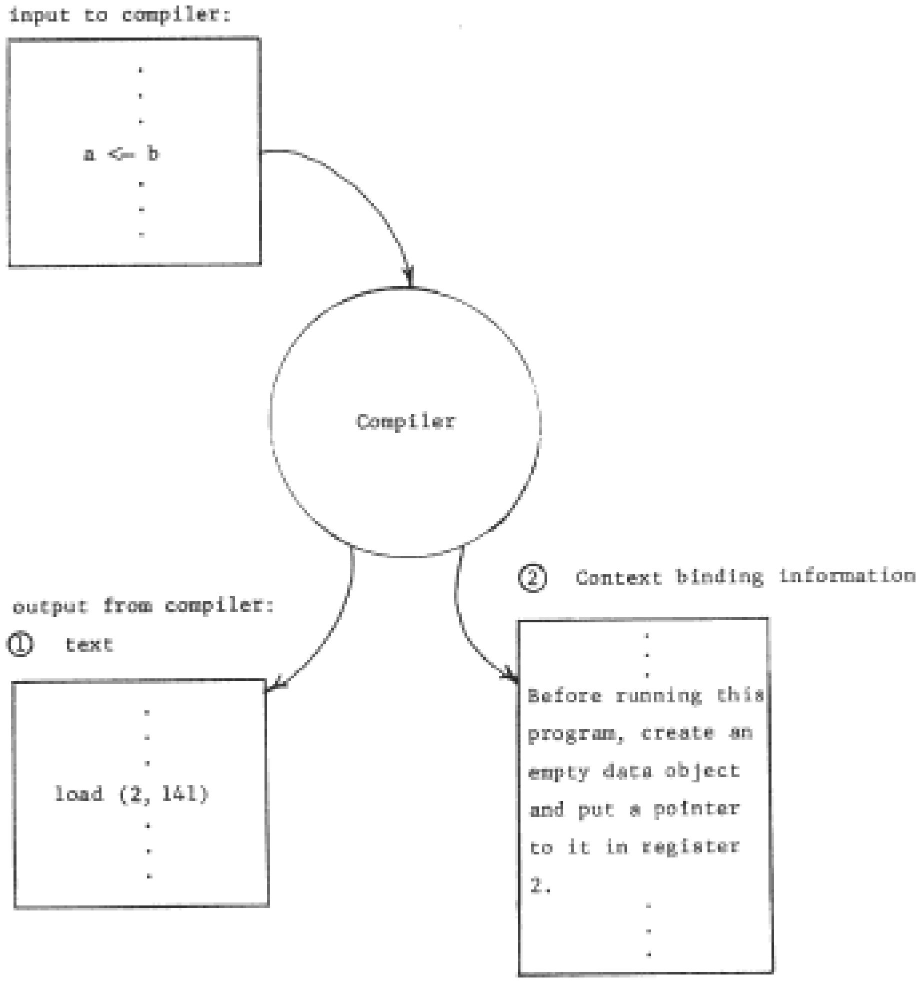


Figure 7 -- When a per-processor addressing context is used, one of the outputs of the compiler is information about the bindings needed to create that context, In this example, the empty data object should have had some values placed

* Various other names for this program are <u>loader</u>, <u>linker</u>, <u>link</u>-<u>editor</u>, or <u>binder</u>.

in it (by earlier instructions in this program) before the load instruction is encountered.

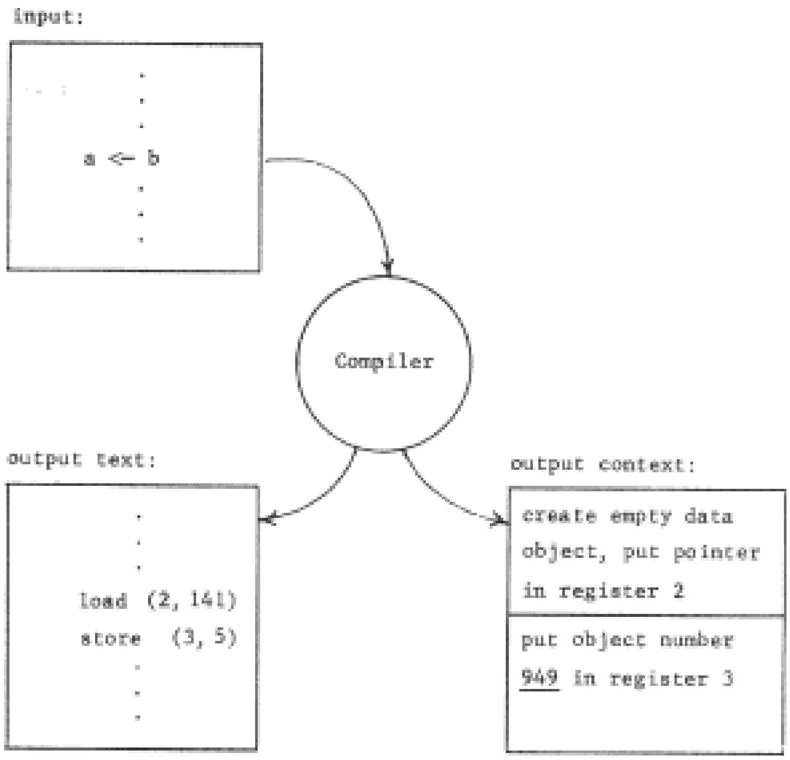


Figure 8 -- Shared data objects can be handled by appropriate entries in the context part of the compiler's output. This output context produces the reference pattern of figure 9.

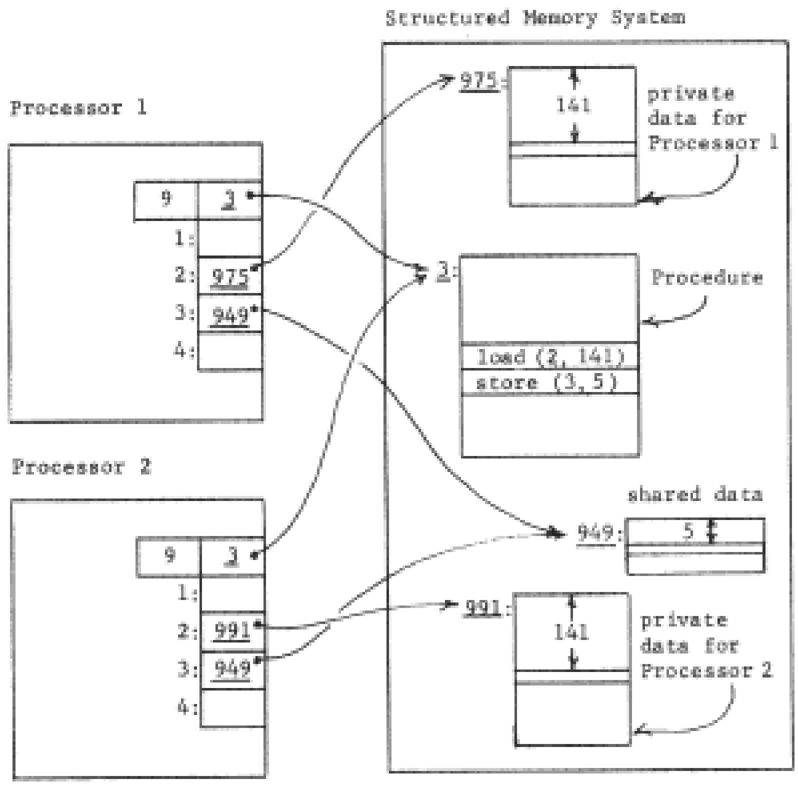
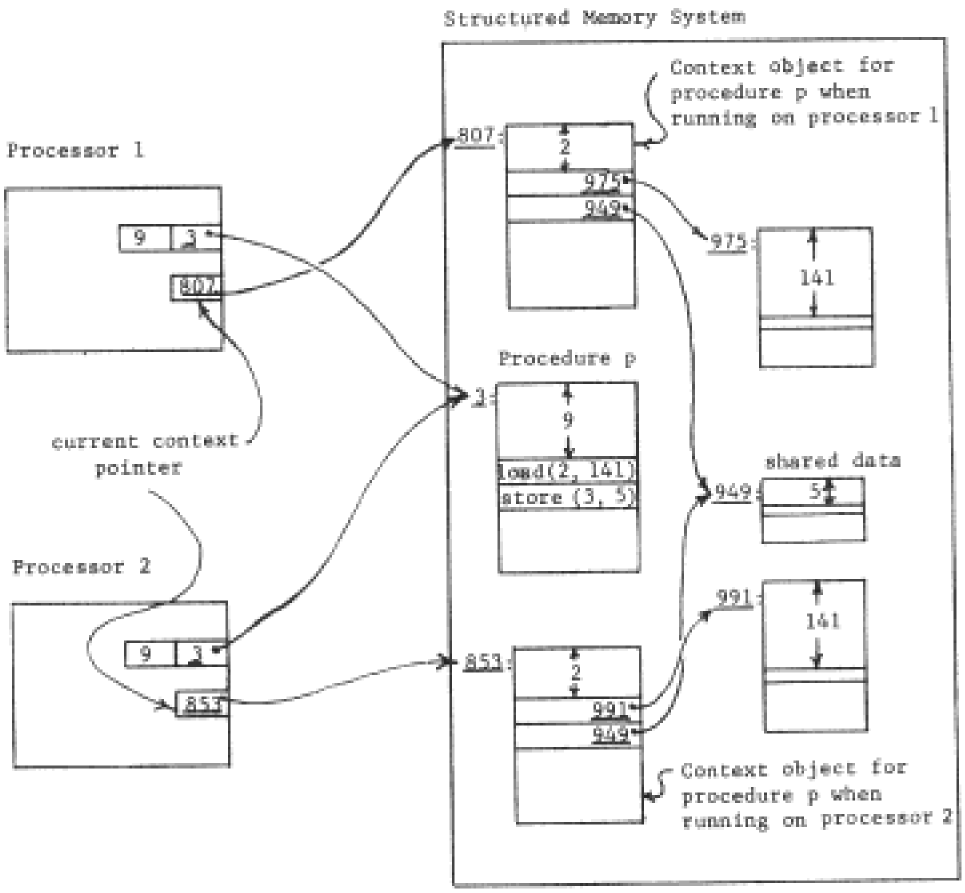


Figure 9 – A shared procedure, using both per-processor private data and shared data, with a context in the processor registers and bindings supplied by the compiler of figure 8.

2. Larger contexts and context switching

In order to achieve user-dependent bindings, we have arranged that each procedure has its own private context, so the first of these elaborations is to arrange for switching from one context to another when calling from one procedure to another. We encounter an interesting implementation dilemma: how many pointer registers should be provided? If there are only a few, some procedure will undoubtedly need to refer to more objects than there are available pointer registers. (Recall that a limited context is a common naming problem) On the other hand, if there are a large number, context switching will require reloading all of them, which could be time-consuming. This dilemma can be resolved by moving the processor context into memory, in a data object, and leaving behind a single processor register, the <u>current context pointer</u>, that points to this context object. Now, a single register swap will suffice to change contexts, at the cost of making the name interpreter more complex and, maybe, slower. Figure 10 illustrates this architecture, figure 11 shows the corresponding changes needed in the context-establishing information that the compiler must supply, and Table I continues to chart our progress*. With this addition to the addressing architecture, in preparation for context switching, we should note that we have quietly introduced explicit closures. The processor now contains a pair of pointers, to a procedure and to a context for the procedure: this pair of pointers can be considered to be an object in its own right, a closure. (In a moment we shall take the final step of placing these explicit closures in memory.)



* For an example of this form of architecture, in Multics the linkage section played the role of the context object a linker initialized it, and compilers routinely produced prototype linkage sections as part of their

Figure 10 -- By placing the per-processor context in a data object in memory, and adding a current context pointer register to the processor, the context is not limited to the number of processor registers. Instead, all addresses are assumed to be interpreted indirectly relative to the segment named by the current context pointer. For example, the instruction at location (3,9) in procedure p contains the address (2,141). The name "2" is resolved by referring to the second location of the object named by the current context pointer. All of the context objects for a given procedure have the same layout, as determined by the compiler, but the bindings to other objects can differ. Note that the combination of the current context pointer and the current instruction pointer in any one processor represents an object, the current closure.

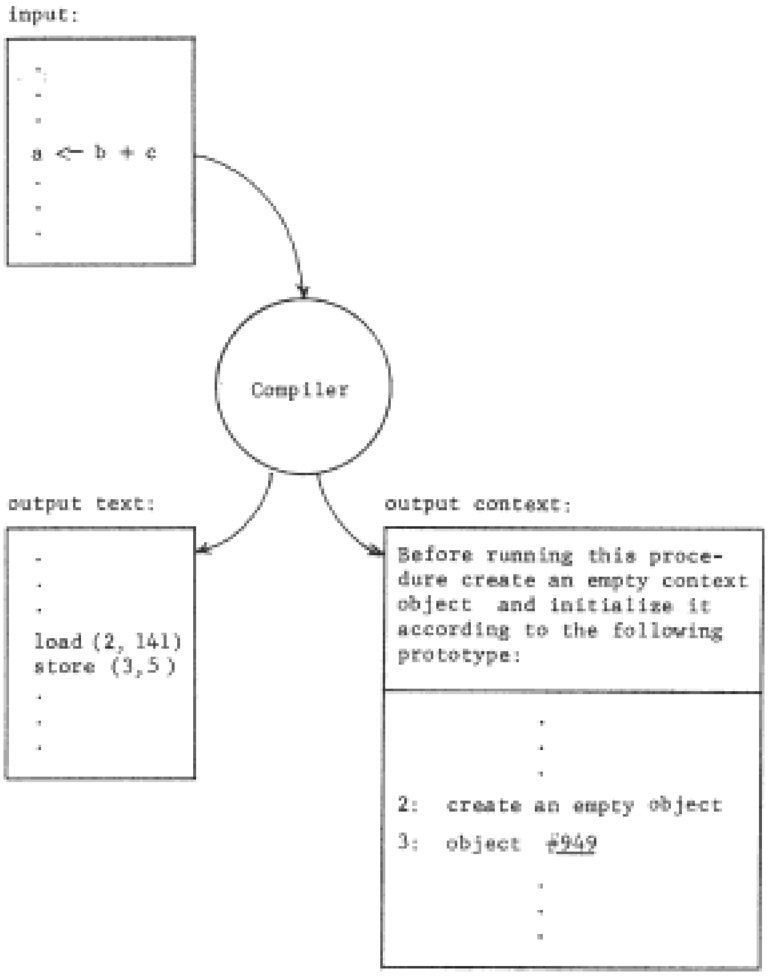


Figure 11 - Compiler output needed to initialize the context objects of figure 10. In addition to the instructions provided by the compiler, one further step is needed:

---

output. One of the processor base registers, known as the linkage pointer, played the part of the current context pointer. [Daley and Dennis, 1968].

just before calling procedure "p", the object number of its context object for this processor must be loaded into the current context pointer register.

The establishment of the context for resolving names of the procedure spans three different times:

1) compile time, when names within the context object are assigned, and the compiler creates the context-establishing information with the aid of declarations of the source program,

2) just before the program is first run, when the context initializer creates and fills in the context object and creates any private data objects,

3) just before each execution of the program, when part of the calling sequence loads the current context pointer register with a pointer to the context object.

We have distinguished between the second and the third times in this sequence on the chance that the program will be used more than once, without need for reinitialization, by the same processor. In that case, on second and later uses of the program, only the third step may be required.

The second elaboration of our per-procedure context scheme is to provide for automatically changing the context when control of the processor passes from one procedure to another. Suppose, for example, the procedure "p" calls procedure "q". In that case, as control passes from "p" to "q" the current context pointer of this processor should change from the processor’s context object for "p" to the processor's context object for "q"; upon return of control, the context pointer should change back. In terms of the naming model, the meaning of a call is that the processor should switch its attention from one closure to another.

Mechanically, we may accomplish these changes by adding one more per-processor object: a closure table, which contains a mapping from procedure object number to the private context object number for every procedure used by this processor. At the same time, we replace the current context pointer with a processor register that contains the object number of the closure table. The name interpreting part of the processor must once again be made more complex. To interpret a name, found as the operand part of an instruction, the processor first uses the closure table pointer and the object number of the next instruction register to look up in this processor's closure table the object number of this procedure's context. It can then proceed as before to interpret the name in that context. Figure 12 illustrates this new closure table pointer register, and a typical object layout just before a call. The call instruction, after resolving the name "4" in the current context to be object number 98, inserts that number in the object number part of the instruction location counter. From then on, the processor will automatically use the context object for procedure "q" in resolving names. When procedure "q" returns to "p", the context automatically is restored to that of "p" when the object number part of the instruction location counter is reset to the object number of "p"*. Table I again identifies the additional function gained.

* We have not specified the way in which procedure "p" tells procedure "q" where its arguments are located or where to return, because it would lead to a distracting discussion of calling mechanisms. Both the return point and the arguments should be viewed as temporary bindings to names already in some naming context of "q", and some machinery is needed both to effect those temporary bindings and to reverse them when "q" returns. For example, the argument addresses and the return point might be pushed onto a stack by "p" and popped off the stack by "q" when it returns. The top frame of the stack is then properly viewed as a distinct naming context for "q". Automatic hardware to perform all the functions of a procedure call is becoming commonplace, as in Multics [Schroeder and Saltzer, 1974] and the Cambridge Capability System [Needham, 1972]. Both of these systems had versions of the closure table in some form.

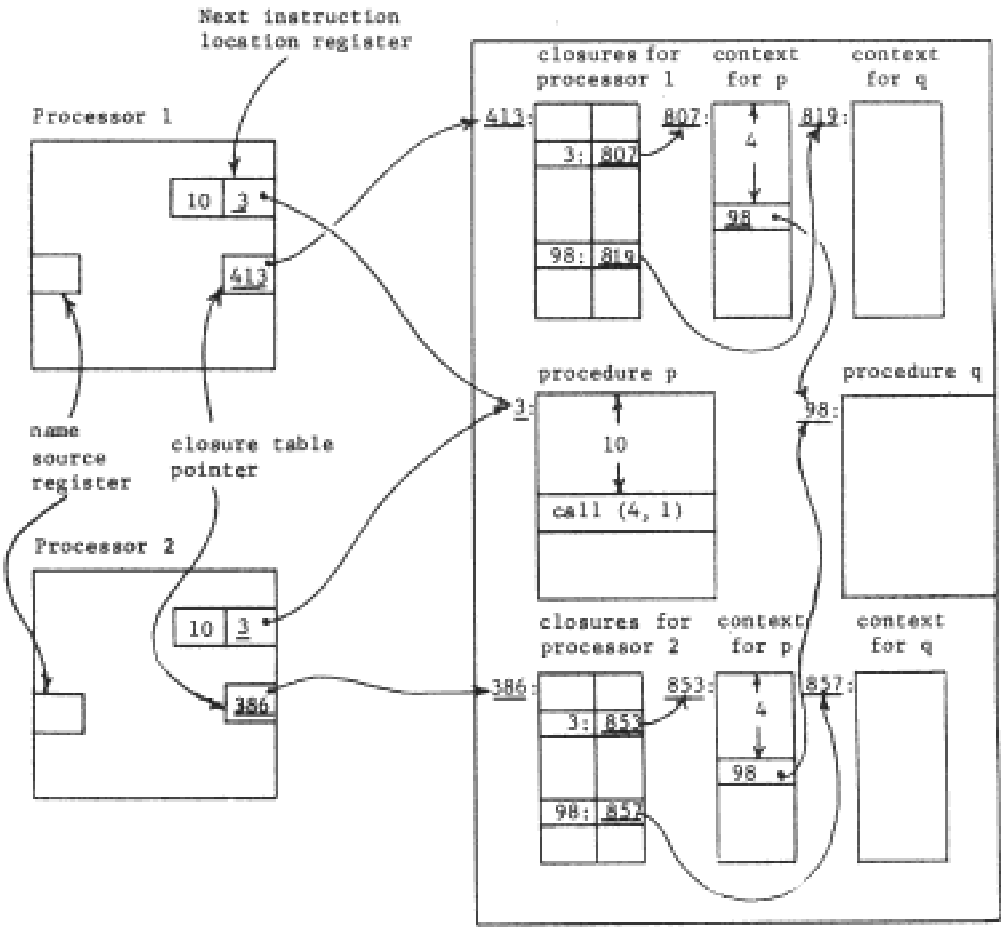


Figure 12 -- Context switching. Procedure "p" is just about to call procedure "q", in both processors. The current context for resolving names is located in two steps, starting with the closure table pointer and the object number of the current procedure. This pair (for processor 1, the pair is (413,3)) leads to a location containing the context for "p" (for processor 1, p's context is object 807). Note that the call instruction in procedure "p" refers to its target using a name in the context for "p" exactly as was the case for data references. The name source register enters the picture when data objects refer to other data objects, as described in the text.

The final elaboration, which is actually omitted in most real systems, is to provide for the possibility that a shared data object should have a user-dependent context. This possibility would be required if it is desired to share some data object without sharing all of its component objects. Such user-dependent binding of course requires that the data object have a per-processor context, just like a procedure object, and one's initial reaction is that figure 12 seems to apply if we are careful to create a context for each such data object and place a pointer to it in the appropriate closure table. A problem arises, though, if we follow a reference by a procedure to a data object and thence to a component named in that object, an operation that may be called an indirect reference. Consider first the direct data reference that occurs if the instruction

get (19,7)

is executed. The number "19" is a name in the current procedure's context object, which selects a pointer to the data object, and the number "7" is an offset within that object; the result would be to retrieve the 7th word and perhaps put it in an arithmetic register. Now suppose the instruction

get (19,7)*

is executed, with the asterisk meaning to follow an indirect reference. Presumably, location 7 of the data object contains, instead of an arithmetic item, an outward reference (say (4,18)) that should be interpreted relative to the per-processor context object associated with this data object by the closure table. If we are not careful, the processor may get the wrong context, for example, the context of the current procedure. To be careful, we can explicitly put in the processor a name source register that the processor always automatically loads with the object number of the object from which it obtained the name it is currently resolving. To obtain the correct context, the processor always uses the current value of the name source register as the index into the closure table. Since it obtains most names from instructions of the current procedure, the name source register will usually contain the object number of the current procedure. However, whenever an indirect reference chain is being followed, the name source register follows along, assuring that for each indirect reference evaluated, the correct context object will be used*.

Another interesting problem arises when a program stores a name into an object. The program must also ensure that the name is correctly bound in the context where that object's names are resolved. Such an operation would occur if dynamic rearrangement of the internal organization of a partially shared structured object were required. Again, since most programming languages do not permit partially shared structures, they also do not provide semantics for rearranging such structures.

We should also note that when a procedure or data object refers to an object by name, we have constructed a fairly elaborate mechanism to resolve the name, to wit:

1) The closure table pointer and the name source register are read to form an address in the closure table.
2) The closure table is read to retrieve the current context object number.
3) The current context object number and the originally presented name of the object are used to form an address within the current context object.
4) The current context object is read to obtain the object number of the desired object.
5) The object number of the object is combined with the offset to form an address for the data reference.
6) The data is read or written.

Thus for each data access, three accesses to the structured memory subsystem (steps 2, 4, and 6) are required. And we might expect that inside the structured memory subsystem, a single access may require three accesses

* As mentioned, the elaboration of a pointer source register is rarely required, because per-processor data contexts are rarely implemented in practice. (One example of a pointer source register appeared in the Honeywell 68/80 protection ring hardware [Schroeder and Saltzer, 1974].) Most programming languages have no provision at all for describing data objects that have per-user private contexts. The TENEX copy-on-write feature can be interpreted as an example of per-user data contexts [Murphy, 1972].

to the location-addressed memory system, if both an object table and also block allocation (paging) are used. Thus it appears that we could be requiring a nine to one expansion of the rate of memory accesses over that required for a single data reference. One solution to this problem lies in speed-up tricks of various kinds, the simplest being addition to the front of the structured memory system of a small but very fast buffer memory for frequently used data items. Since the processor's name interpreter refers to the current context object once for every instruction that has an operand reference, its object number would almost certainly remain in even the smallest buffer memory. A similar observation applies to frequently resolved names within the context. Thus, although there are many memory references, most of them can be made to a very fast memory.

A second approach is to reduce the number of object-to-object cross references. Depending on the level of dynamics that a set of programs actually uses, it may be feasible to prebind many references, and thereby avoid exercising much of the addressing architecture. For example, in figure 12, a "prebinder" may be able to take procedure p and procedure q, and construct from them a single procedure object with a single, combined context, and with the call from p to q replaced by an internal reference. In effect, the prebinder replaces containment by name with containment by copy. This kind of prebinding would be appropriate if p and q are always used together, and there are no name conflicts in their outward references. Doing prebinding irrevocably commits the connection between p and q; a user of the combined procedure cannot substitute a different version of q. If an application is sufficiently static in nature, and does not share writeable objects with other applications, one could in principle prebind all of the objects of that application into a single big object containing only self-references, and thereby completely avoid use of any special hardware architecture support at all. Such a strategy might be valuable in converting a system from development to production. The development system might over use a software interpreted version of the addressing architecture. The extreme slowness of such software interpretation would be less important during development, and prebinding would eliminate the interpretation when the production system is generated.

3. Binding on demand and binding from higher-level contexts

Figure 12 illustrates a static arrangement of contexts surrounding procedures, but does not offer much insight into how such an arrangement might come into existence. Since there are two levels of contexts, there are now two levels of context initialization. The creation of a new virtual processor must include the creation of a new, empty closure table and the placement of the object number of that table in the closure table pointer of the new virtual processor. The filling in of the closure table, and the creation and filling in of individual contexts, may be done at the same time, by the creator of the virtual processor. Alternatively the creator may supply only one entry in the closure table, and one minimally completed context for the context initializer procedure itself, and expect that procedure to fill in the remainder of its own context and add more context objects to the closure table as those entries are needed.

This latter procedure we shall term binding on demand*, and it is usually implemented by adding to the processor the ability to recognize empty entries in a context or the closure table. When it detects an empty entry, the processor temporarily suspends its normal sequence, saves its current state, and switches control to an entry point of the context initializer program. (The processor may have a special register that was previously set to contain a pointer to the context initializer. Alternatively it may just transfer to a standard address, in some standard context, with the assumption that that address has been previously set to contain an instruction that transfers to the context initializer.) The context initializer examines the saved processor state, to interpret the

* The term dynamic linking was used in Multics, one of the few systems that actually implemented this idea (Daley and Dennis, 1968].

current address reference so as to determine which entry in which context is missing, and proceeds to initialize that entry.

Binding on demand is a useful feature in an on-line programming system, in which a person at a terminal is interactively guiding the course of the computation. In such a situation it is frequently the case that a single path through a procedure, out of many possible paths, will be followed, and that therefore many of the potential outward references of the procedure will not actually be used. For example, programs designed for interactive use often contain checks for typing errors or other human blunders, and when an error is detected, invoke successively more elaborate recovery strategies, depending on the error and the result of trying to repair it. On the other hand, if the human user makes no error, the error recovery machinery will not be invoked, and there is no need for its contexts to have been initialized. For another example, consider the construction of a large program as a collection of subprograms. It can be important for one programmer to begin trying out one or a few of the subprograms before the other programmers have finished writing their parts. Again, if the programmer can, by adjusting input values, guide the computation through the program in such a way as to avoid paths that contain calls to unwritten subprograms, it may be possible to check out much of the logic of the program. Such partial checkout requires the ability to initialize partially a context (leaving out entries for non-existent subprograms)*.

A second idea related to context initialization stems from the suggestion, made earlier, that a context initializer can perform more elaborate operations than simply creating empty data objects or copying object numbers determined at compile time. Returning to figure 11, the compiler creates a prototype context object for use by the context initializer. If the context initializer were prepared for it, this

---

* Binding on demand is an idea closely related to a programming language technique named _lazy_ _evaluation_, in which either binding or calculation or both are systematically postponed until it is apparent that the result is actually needed. [Henderson and Morris, 1976].

prototype could also contain an entry of the form "look for an object named 'cosine' and put its object number in this entry of the context". This idea requires that the name "cosine" be a name in some naming context usable by the context initializer, and it is really asking the context initializer to perform the final binding, by looking up that name at run time, discovering the object number, and binding it in the context being initialized. There are several situations in which it might be advantageous £o do such binding from a higher-level context at context initialization time rather than at compile time:

1) The program is intended to run on several different computer systems and those different systems may use different object numbers for their copies of the contained objects.
2) At the time the program is compiled, the contained object does not yet exist, and no object number is available. However, a symbolic name for it can be chosen. (This situation arises in the large-system programming environment mentioned before. It also arises when programs call one another recursively.)
3) There may be several versions of the contained object, and the programmer wants control at execution time of which version will be used on a particular run of the program.

Bindings provided at compile time are created with the aid of declaration statements appearing inside the program being compiled. As we shall see, bindings created at run time must be created with the aid of declarations external to, but associated with, the program. It is exactly because the declarations are external to the program that we obtain the flexibility desired in the three situations described above.

Most computer operating systems provide some-form of highly structured, higher-level, symbolic naming system for objects that allows the human programmer or user of the system to group, list, and arrange the object with which he works: source programs, compiled procedures, data files, messages, and so on. This higher-level context, the file system, is designed primarily for the convenience of people, rather than programs. Among typical features of a file system designed for interactive use by humans are synonymous names, abbreviations, the ability to rename objects, to rearrange them, and to reorganize structures.

A program, in referring to computational objects, usually does so in an addressing architecture like that developed up to this point, using names that are intelligible to hardware, and explicitly attempting to avoid potential troubles such as uncertain name resolutions, name conflict, and incorrect expansion of abbreviations. Thus the machine-oriented program addressing architecture is usually made as distinct and independent as possible from the human-oriented file system. The program context initializer, however, acts as a bridge between these two worlds, prepared to take symbolic names found in the program execution environment, interpret them in the context of the file system, and return to the program execution environment an object binding that is to match the programmer's intent.

Development of the higher-level file system and that part of the program context initializer that uses it is our next topic. First, however, it may be helpful to review, in Table II, all of the examples of naming and name binding that occur in our addressing architecture alone. This table emphasizes two points. First, in even a simple naming system there are many examples of naming and name binding. Second, in the course of implementation of appropriate name-binding facilities for modular programming, there are many places in which naming is itself used as an internal implementation technique. This internal use of naming and binding is conceptually distinct from the external facility being implemented, but real implementations often blur the distinction, as an implementation shortcut or out of confusion. These two points should be pondered carefully, because in developing a higher-level naming concept in the next sections, we will utilize the naming contexts of the addressing architecture, create intermediate levels of contexts, and have several opportunities to confuse the problem being solved (building up a naming context) with the method of solution (using naming concepts).

Table II -- Examples of naming and binding in the model of addressing architecture.

	object that contains the name	form of the name	closure implementation	context
1.	location-addressed memory system	none	none	none (objects are directly included)
2.	object map	absolute address of object	name interpreter has only one choice	location-addressed memory system
3.	virtual processor retrieving instructions	procedure object number	name interpreter has only one choice	object map
		instruction location counter	procedure object number register	procedure object
4.	procedure context object	object number	only one choice	object map
5.	procedure object	offset within context object	closure table	procedure context object
6.	closure table	object number of context object	only one choice	object map
7.	processor retrieving operand, step 1 of name interpretation	procedure object number	closure pointer	closure table
8.	processor retrieving operand, step 2 of name interpretation	offset within context	dynamically constructed in step 1	procedure context
9.	symbolic name in prototype context	symbolic object name	supplied by context initializer	file system
10.	symbolic name in source program	name resolvable to compile time	supplied by compiler	file system
		name not resolvable at compile time	containment in procedure text	prototype context

C. Higher-Level Naming Contexts, or File Systems

1. Direct-Access and Copy Organizations

Higher-level naming contexts, or file systems. are provided in computer systems primarily for the convenience of the human users of the systems. In on-line systems, a file system may assume a quite sophisticated form, providing many features that are perceived to be useful to an interactive user*.

The foremost property of a file system is that it accepts names that are chosen and interpreted by human beings--ideally arbitrarily chosen, arbitrary length strings of characters. The context used to resolve these user-chosen names is called a catalog†, which in its simplest form is an object containing pairs: a character string name and a unique identifier of the object to which that character-string name is bound‡. The unique identifier names the object in the context of the underlying storage system. Normally, the name-resolution mechanism of the file system is sufficiently cumbersome that it is not economically feasible for a running program to use it for access to single words. Therefore, some mechanism must be provided for making sure that most references are to a high-speed addressing architecture. There are two commonly-found ways of organizing a file system's relation to this addressing architecture: the copy file system, and the direct-access file system.

---

* A similar, but typically less-sophisticated file system is often used for batch processing job control languages.

† In many systems, the term directory is used.

‡ In many systems, catalogs are also used as repositories for other things of interest about an object, such as details of its physical representation, measures of its activity, and information about who is authorized to use the object. Such use of a catalog as a repository as well as a naming context tends to confuse naming issues with other problems, so we shall assume for our present discussion that the underlying storage system provides for the repository function and that catalogs are exclusively naming contexts. In a later discussion of implementation considerations, some of the effects of mixing these ideas will be examined.

In a <u>copy</u> file system, the catalog manager operates quite independently of the addressing architecture. A program may call upon the catalog manager to create an object with a given file system name. When the program wishes to read or write data from or to the object, it again calls the catalog manager, at a read or write entry point, giving the character-string name of the object, and the address of a data buffer in the addressing architecture. The catalog manager looks up the character-string name, finds the object identifier, and then performs the read or write operation by copying the data from its permanent storage area to the addressing architecture or back. Thus, in a copy file system, use of an object by name is coupled with the kind of data movement usually associated with multilevel memory management; usually the file system uses secondary storage devices that have their own addressing structure, but this addressing structure is hidden from the user of the file system. Figure 13 illustrates a copy file system.

In a <u>direct</u>-<u>access</u> file system, the catalog manager also creates a higher-level naming context, but it uses the addressing architecture itself as the mechanism for data access. Instead of performing copying read and write operations for its caller, it provides an entry that we may name "get identifier", which returns to its caller an object identifier, for a given character-string name, suitable for use directly with the lower level addressing architecture. Figure 14 shows a direct access file system.

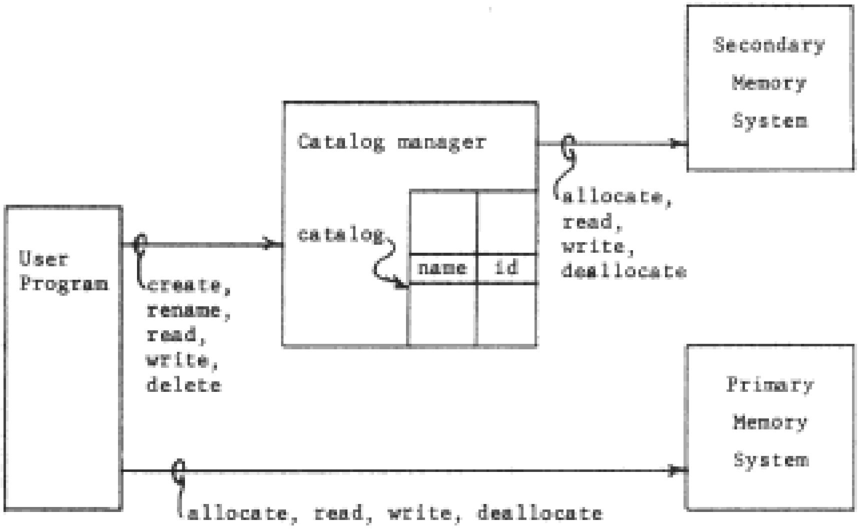


Figure 13 -- Organization of a simple copy file system, The user calls on the catalog manager to create objects and record character-string names for them. The identifiers (labelled id) stored by the catalog manager are names in the context of a secondary memory system, not directly accessible to the user program. The user program, to manipulate an object, must first copy part or all of it into the primary memory system by giving a read request to the catalog manager, and specifying the character-string name of an object and the name of a suitable area in the primary memory system.

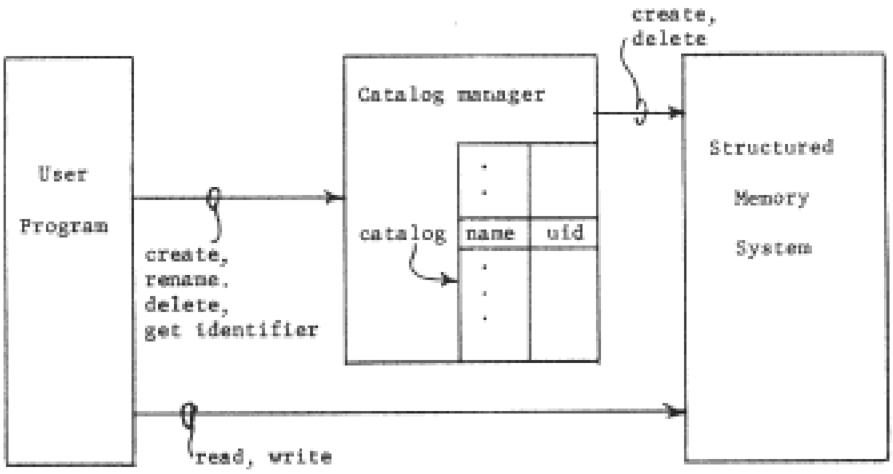


Figure 14 -- Organization of a simple direct-access file system. The user calls on the catalog manager, as before, to create objects and to record character-string names. However, the entry "get identifier" returns the object number, or unique identifier (labelled uid) by which an object is known in the structured memory system. The

objects themselves are created and stored by the structured memory system, in response to direct requests from the catalog manager. The catalog manager may itself use the structured memory system to store the catalog, but that use is both unimportant and invisible to the user of the catalog so long as there is only one catalog, since the user does not need to know the name by which the catalog manager refers to the catalog. As shown, there is nothing to prevent a user program from additionally making direct create and delete calls on the Structured Memory Subsystem, thus creating uncatalogued objects, or destroying catalogued objects without the knowledge of the catalog manager.

Which of these two kinds of designs is preferred depends on the arrangement of the available addressing architecture. If a structured memory system that provides multilevel management of all primary and secondary memory is available, then names of the structured memory system may be embedded in program contexts, and a direct-access file system seems preferable. If, on the other hand, permanent storage of large volumes of data on secondary memory is managed separately from the primary memory system used by programs, then the object names stored in a catalog would refer to a context not available to a program, and the copy form of file system design is appropriate*.

The distinction between these two kinds of file systems is an important one, since programs must be written differently in the two kinds of design. The direct-access file system is sometimes called a <u>one</u>-<u>level</u> <u>store</u> [Kilburn 1961], because a program can consider all other programs and data to be nameable in a single context, rather than in two contexts with the necessity for explicitly copying objects from one context (the permanent storage system) to another (the program execution environment) in order to manipulate them†. Note that a direct-access file system can be

* Most copy file systems obscure our precise distinction by providing a temporary context consisting of currently active files. Upon some program's declaring an interest in a certain file, the file system allocates a name for the file in the temporary context, and hands that name back to the program for use in future read/write calls. Despite the similarity in structure to the "get identifier" call, such systems are not direct-access because it is still necessary to explicitly copy things from the context of the file system to the context of the addressing architecture in order to manipulate them.

† Another name sometimes used for the direct access file system is the "Virtual Access Method".

used to simulate a copy file system, by copying objects from one part of it to another. The reverse is quite a bit harder to do, since a one-level store requires that the base level system implement a single, universal naming context.

In either a direct-access or a copy file system that implements a single catalog, the operation of a context initializer program (the program, described in part B, that connects the higher level file system context to lower level program execution contexts) is relatively straightforward. Consider the direct-access case first. The context initializer starts with a character-string name found in a prototype context. The context initializer calls the "get identifier" entry of the direct-access file system, takes the returned identifier as an object number, and inserts it in the program context being initialized. In the case of a copy file system, the context initializer goes through an extra step, known as loading the object. That is, it allocates a space for the object in primary memory, and then it asks the copy file system to read a copy of the referenced object into primary memory. Finally, it places the primary memory address of the newly-copied object in the context being initialized. To avoid copying a shared object (that is, one named by two or more other objects) into primary memory twice, the context initializer must also maintain a table of names of objects already loaded, a reference name table. Before calling on the file system, the context initializer must first look in the reference name table to see if the name refers to an object already loaded. Because the higher-level file system requires copying of objects in order to use them, the context initializer for the addressing architecture is forced to develop in the reference name table an image of those parts of the higher-level file system that are currently in use*.

---

* In practice, dealing with shared objects also involves several other complicated issues, such as measuring activity for multilevel memory management or accounting, and maintaining multiple copies for reliability; such issues lead a good distance away from the study of name binding and will not be pursued here.

In examining the operation of the context initializer in the environments of copy and direct-access file systems, we have identified the most important operational distinctions between the two designs. For simplicity in the succeeding discussion, we shall assume that a direct-access file system is under discussion, and that the adaptation of the remarks to the copy environment is self-evident.

2. Multiple catalogs and naming networks

The single-catalog system of figure 13 and 14 is useful primarily for exposing the first layer of issues involved in developing a file system, although such systems have been implemented for use in batch-processing, one-user-at-a-time operating systems*. As soon as the goal of multiple use is introduced, a more elaborate file system is needed. Since names for objects are chosen by their human creators, to avoid conflict it is necessary, at a minimum, to provide several catalogs, perhaps one per user†.

If there are several catalogs available, any of which could provide the context for resolving names presented to the file system, some scheme is needed for the file system name interpreter to choose the correct catalog. Technically, some mechanism is needed to provide a closure. A scheme used in many systems that provide one catalog per user is as follows: the state of a user's virtual processor usually includes a register (unchangeable by the user) that contains the user's name, for purposes of resource usage accounting and access control‡. The file system name interpreter simply adds another use to this name: as a closure

---

* The FORTRAN Monitor System (FMS) for the IBM 709 computer is a typical example of a batch processing system that had a single catalog, for a library of public subroutines.

† Most of the first generation of time-sharing systems, such as CTSS, APEX, the SDS-940, TYMSHARE, DTSS, VM/370-CMS, and GCOS III TSS provided one catalog per user. OS/360 provided a single system-wide catalog with multicomponent names that could be used to provide the same effect.

‡ In discussions of information protection, this name is usually called a principal identifier.

identifier. The name interpreter resolves all object names presented to it by first obtaining the current user name from the virtual processor name register, and looking that up in some catalog of catalogs, called a <u>master user catalog</u>. The user's name is therein bound to that user's personal catalog, which the interpreter then uses as the context for resolving the originally presented object name. This scheme is simple, and easy to understand, but it has an important defect: it does not permit the possibility of sharing contexts between users. Even if one user knows another user's name, and has permission to use the second user's file, the first user cannot get his program to contain the correct file system name for the file in the other user's catalog: the user's own catalog is automatically provided as the implicit context for all names found in his programs. Reusing the account or principal identifier as a closure identifier, while simple, is inflexible.

To understand the reason why shared contexts are of interest, we must recall that the file system is a higher-level naming context provided for the convenience of the human user rather than a facility of direct interest to the user's programs. The commands typed by the user at the terminal to guide the computation specify the names of programs and data that he wishes the computation to deal with, and he expects these names to be resolved in the context of the file system. The user would like to be able to express conveniently a name for any object that is of interest to him. If he makes frequent use of objects belonging to other users, then to minimize confusion he should be able to use the same names for objects that their owners use. These considerations suggest a need for a scheme that allows contexts to be shared.

A simple scheme that supplies the minimum of function is to add a second, user-settable register to the user's virtual processor, dedicated exclusively to the function of closure identification. We shall name this new register the <u>working catalog</u> register; and have the file system resolve names starting from that register rather than some register intended for a purpose only accidentally connected with naming. The working catalog register would normally contain a name that is bound, for example in a master user catalog, to the user's personal catalog*. When the user wishes to use a name found in some other catalog, he first arranges that the working catalog register be reloaded with the name of the other catalog. This scheme has been widely used, and is of considerable interest because it exposes several issues brought about by the desire to share information:

1) In some systems, protection of information from unauthorized use is achieved primarily by preventing the user from naming things not belonging to him. For example, before the working catalog register was added, the file system name interpreter resolved all names relative to the protected principal identifier register, thereby preventing a user from naming objects belonging to others. With the addition of the working catalog register, the user can suddenly name every object in the file system. Protection must be re-supplied either by restricting the range of names of catalogs that the user can place in the working catalog register (for example, permitting the user to name either his catalog or a public library catalog, but nothing else) or else by developing a protection system that is more independent of the naming system--an access control list for each object, for example†.
2) The change of context involved when the working catalog is changed is complete--<u>all</u> names encountered in the program being executed, or the context initializer program, will be resolved relative to the name of the current working catalog. If program A contains a reference to object B, and the context initializer is expected to dynamically resolve the name "B" at the time it is first used, that resolution will depend on the working catalog in force at the instant of the first reference to B. Here we have a potential conflict between the

---

* For implementation speed, the working catalog might actually be represented by its object number rather than by a character-string name requiring resolution every time a name is used.

† The lack of ability to name other users' objects was the primary file system protection scheme of M.I.T.'s Compatible Time-Sharing System. The equivalent of the working catalog register in that system was restricted to

intention of the programmer in embedding the name "B" in the program and the intention of the current user of the program, who may want to adjust the working catalog to assure correct resolution of some other name to be typed at the terminal as input to program A. Since there are two effectively independent sources of names, perhaps there should be two working catalogs, and an automatic way of choosing the correct working catalog depending on the source of the name. Unfortunately, inside the computer both kinds of names are presented to the file system by similar-appearing sources--some program calls, giving the name as an argument. (We are here encountering a problem described earlier, that the wrong implicit context may be supplied by the name interpreter.)

3) Having once changed the context in which names are resolved, the human user (or the program writer) must constantly remember that a new context is in force, or risk making mistakes. As an example of a complication that can arise, many systems provide an <u>attention</u> feature that allows a user, upon pressing some special key at his terminal, to interrupt the current program and force control to some standard starting place. If the working catalog register was changed by the current program (perhaps in response to a user request to that program) then the "standard" starting place may start with a "non-standard" naming context in force.

These last two issues suggest that an alternative, less drastic approach to shared contexts is needed: some scheme that switches contexts for the duration of only one name resolution.

One such scheme is to provide that each name that is not to be resolved in the working catalog carry with it the name of the context in which it should be resolved. This approach forces back onto the user the responsibility to state explicitly, as part of each name, the name of the appropriate context. We assume, as before, that catalogs are to have human-

---

the users' catalog, the library, or a catalog held in common among a designated group of users [Crisman, 1965].

readable character-string names, and therefore there must be some context in which catalog names can be resolved. Figure 15 shows one such arrangement, called a naming network, an arrangement characterized by catalogs appearing as named objects in other catalogs. We have chosen the convention that to express the name of an object that is not in the working catalog, one concatenates the name of the containing catalog with the name of the object, inserting a period between the two names. The absence of a period in a name can then be taken to mean that the name is to be resolved in the working catalog. One would expect names containing periods to come to programs as input arguments, originating perhaps from the keyboard; they represent a way for the user to express intent precisely in terms of the current naming structure, which can change from day to day. On the other hand, one would permanently embed in a program only single-component names, to avoid the need to revise programs every time objects are rearranged in the catalog structure, We shall return to the topic of binding names of the program to names of the catalog structure after first exploring naming networks in some depth.

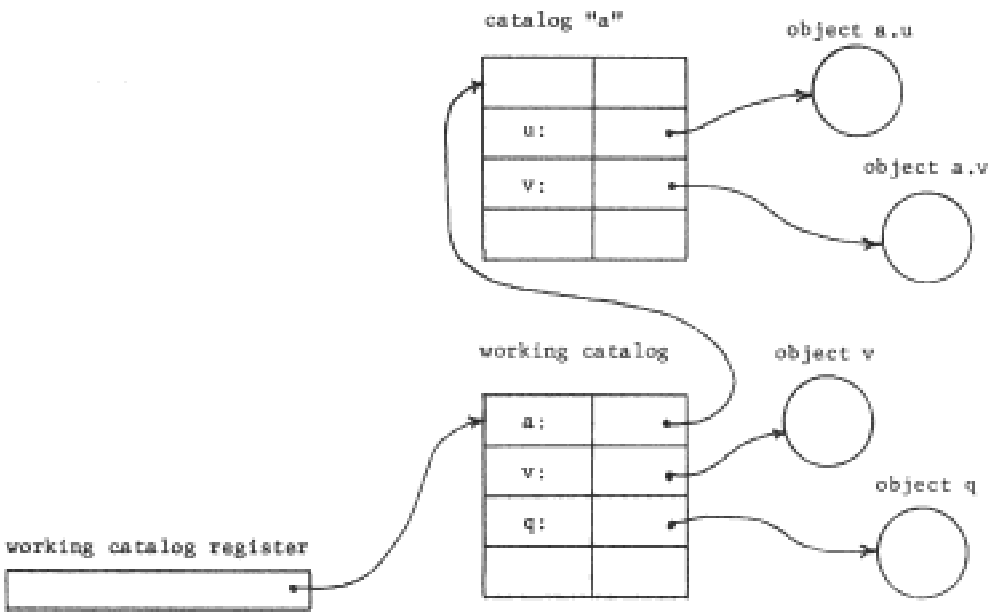


Figure 15 -- A simple naming network. All single-component names given to the file system are resolved in the context named by the working catalog register. All two-component

names are resolved by resolving the first component in the working catalog, which leads to another catalog in which the second component may be resolved. As in earlier figures, the arrows represent object addressing; the (italicized) object numbers have been omitted to simplify the figures. (The working catalog register refers to the working catalog by object number in this example, although it could also be implemented as a multicomponent path name relative to some standard starting catalog known to the file system).

A naming network generalizes in the obvious way if we admit names consisting of any number of components--these names are called path names. Thus, in figure 15, it might be that the object named "a.v" is yet another catalog, and that it contains an object named "cosine"; the user could refer to that object by providing the path name "a.v.cosine"*. Note also that the path names "v" and "a.v" refer to distinct objects--either may be referred to despite the apparent name conflict.

A naming network admits any arbitrary arrangement of catalogs, including what is sometimes called a recursive structure: in figure 16, catalog "a.v" contains a name "c", bound to the identifier of the original working catalog. The utility of a recursive catalog structure is not evident from our simple example--it merely seems to provide curious features such as allowing the object named "q" to be referred to also as "a.v.c.q" or "a.v.c.a.v.c.q". But suppose that some other processor has its working catalog register set to the catalog we have named "a.v". Then from the point of view of that processor, objects in catalog "a.v" can be referred to with single component names, while the object that the first processor knew as "q" could be obtained by the name "c.q". By admitting a recursive catalog structure, every user can have a working catalog that

---

* If it should turn out that object "a.v" is not a catalog, the user has made a mistake; in a well-designed system the file-name interpreter should have some provision for detecting this mistake. For example, in an object-oriented system, each object contains as part of its representation the identification of its type, and the underlying system would report an error to the file system if it attempted to perform a catalog lookup on an object not of the catalog type. If an object-oriented system is not used, perhaps the higher level catalog would contain for each entry a flag that indicates whether or not that entry describes another catalog.

contains named bindings to any other user's catalog. Thus the original goal, of providing shared contexts, has been met*.

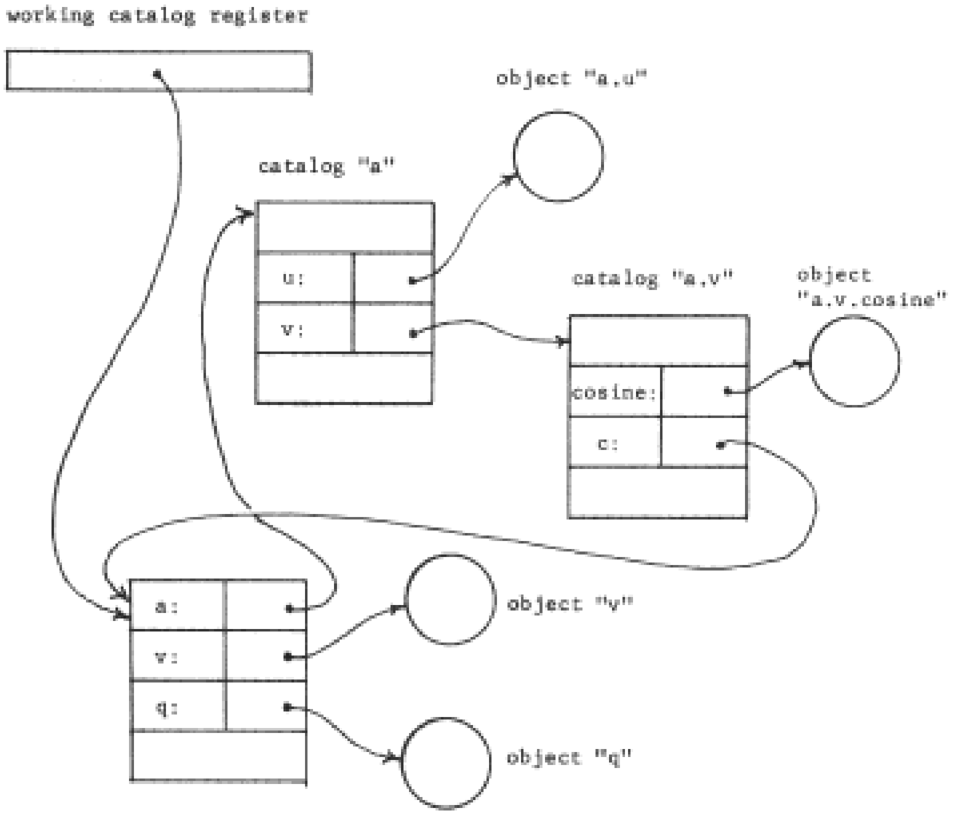


Figure 16 -- A naming network with recursive structure. Object "v" is also known by the name "a.v.c.v".

3. <u>The dynamics of naming networks</u>

If we were to implement the file systems of figures 13 or 14, with the intent of having a naming network, we would probably discover an important defect: although we could easily implement entries to "read", "write", or "get identifier" that understood path names in an existing naming network, and we could also implement an entry to create a new object or catalog, we could not write a program that called those entries to

* Naming networks are not often encountered in operating systems. The CAL time-sharing system [Lampson and Sturgis, 1976] and the CAP system [Needham and Birrell, 1977] are two examples, In data base management systems, the

create a naming network with recursive structure--we would be limited to creating a simple tree structure. Unless our applications were sufficiently static that we could insert in advance all recursive cross-references among catalogs that might ever be needed, we should make provision for programs dynamically to add recursive cross-references by calling the file system. These provisions must include:

1) some way to add to a catalog an entry that represents a binding to a previously existing object
2) some way of naming previously existing objects, so that provision one can be accomplished.

The first provision seems straightforward enough, but the second one implies a fundamental limitation of some kind in the conception of naming networks, at least so far as dynamically constructing them is concerned: one can dynamically extend a naming network only by

1) creating new objects, or
2) adding "short-cut" bindings to objects that were already nameable by some other name.

Thus, in figure 17, one could imagine a request to the file system like "Add to my working catalog a cross-reference, named 'm', to the catalog currently nameable as 'b.x.m'," or "Add to catalog 'b.x.m' a cross-reference, named 'r', to catalog 'b'." These two requests would add the bindings indicated in the figure with dashed lines, but neither request increases the range of objects that can be named. Meanwhile, there is no way available to express the concept "add a cross-reference named 'z' to catalog 'b', that allows access to the catalog labelled 'unnameable' in figure 17 ." Note that this catalog is unnameable only from the point of view of the working catalog register; some other user might have a name for it. If no one had a binding for it, it would be a "lost object", about which more will be said later.

CODASYL standard data base system defined by their Data Base Task Group (DBTG) called for a recursive naming network [CODASYL, 1971].

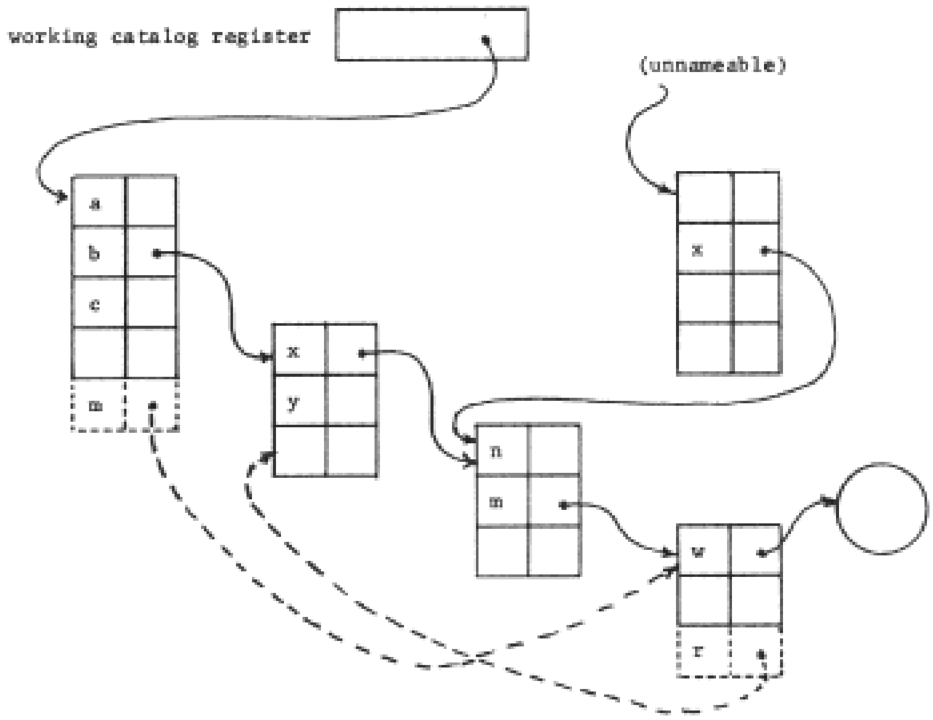


Figure 17 -- A naming network containing an unnameable catalog.

In practice, the problem of inaccessibility is not so serious as it might initially seem: a modest discipline on creation of catalogs can control the situation. A typical strategy for a time-sharing system might be as follows*:

1) When the time-sharing system is first brought into existence, create a "root" catalog, and place in it two more newly created catalogs, one for the library (named "library") and another for individual users' home catalogs (named "users".)
2) For each user of the time-sharing system, create a home catalog, arrange that whenever that user logs in, the working catalog register be loaded by the login procedure to contain the path name

* This is a version of the strategy used in the CAL time-sharing system.

(relative to the "root" catalog) of that user's home catalog, and place in the user's home catalog a (recursive) entry binding the name "root" to the unique identifier of the root catalog.

Now, each user will find that he can refer to his own files by simply giving their names, he can refer to library programs by preceding their names with "root.library.", and he can refer to a file in the catalog of his friend "Lenox" by preceding its name with "root.users.Lenox.". If he finds that he makes frequent use of files belonging to Lenox, he can place in his own catalog a new entry directly binding some appropriate name (say "Lenox") to the identifier of Lenox's catalog. The only obvious effect of this extra binding is that shorter names can now be used to refer to objects in Lenox's catalog.

Notice that if a user accidentally destroys the cross-reference to the catalog named "root", that user would find that he could name nothing but things in his own catalog. The root catalog therefore plays an important part in making a naming network useful in practice.

4. Binding reference names to path names

It remains for us to pick up several loose ends and glue them together to complete the picture of name binding operations that appear in a computer system. However, before inventing any further mechanisms, let us first stop and review the collection of machinery we have developed already, so as to understand just what functions have been provided and what is missing.

We began by assuming as an underlying base a universal naming context in which all objects have unique, system-wide identifiers. We then developed on this base a systematic way of using hardware-oriented reference names and contexts in which those names resolved to underlying universal names--an addressing architecture. The purpose of these reference names was to allow programs to be constructed of distinct data and procedure objects that refer to one another using hardware-interpretable names that are unambiguously resolved in closely associated contexts. We also observed that these contexts must be initialized, either as part of the construction of the program or e l se dynamically as the program executes (in order to avoid modifying the program when it is used in a different application). We briefly outlined the place of the context initializer program as a bridge between the machine-oriented naming world of the addressing architecture and a higher-level, human-oriented file system naming world. The purpose of the context initializer is to take symbolic names found in the prototype context segment of a program, interpret those names in the higher-level context of the naming network, and place in a context object accessible to the addressing architecture an appropriate binding to a specific object.

Next, we developed the outline of a human-engineered file system--a naming network--to be used as the context for people guiding computations. During this development we observed that the dynamic initialization of the lower-level reference name context, say of a procedure, sometimes can involve resolution of symbolic names in the higher-level file system. A problem we noticed, but never quite solved, was that these symbolic names are of two origins: some are supplied by the writer of the procedure, and are intended to be resolved according to that writer's goals, and some are supplied by the user of the procedure, in the course of supplying instructions to the program at execution time. These latter names are presumably intended by the user to be resolved in the file system relative to the user's current working catalog.

Thus, for example, a user may have a working catalog containing a memorandum needing revision, which the user has named "draft". The user invokes an editing procedure, and asks that editor to modify the object he knows by the name "draft" in the working catalog. But it is possible (even likely) that the author of the editor program organized the editor to make a copy of the object being edited (so as not to harm the original if the user changes his mind); perhaps that author chose the name "draft" for the object meant to contain the copy. We have, in effect, two categories of outward symbolic references from the program, the first category to be resolved relative to the working catalog, and the second relative to some as yet unidentified, place in the naming network. The name interpreter used for the translation from file system names to unique identifiers is being called upon to supply one of two different implicit contexts; we have so far provided only one, the working catalog.

To inform the name interpreter which context to use is straightforward: the semantics of use are different and apparent to the translator or interpreter of the program. A name supplied by the author of the program appears as a character string in the source program in some position where reference to an object is appropriate, while a name supplied by the user appears as a data character string in a position where the programmer has indicated that it should be converted into a reference to an object*. Thus if we simply arrange that explicitly programmed conversions from character string to reference be done with the working directory as a context, while all other names found in the source program be interpreted in some other (as yet unspecified) context, we can distinguish the intents of the author and the user of the program.

This approach leaves one final question: what is the appropriate other context in which to resolve outward symbolic references provided by the author of an object? We are here dealing with a situation similar to that posed by some programming languages, in which a procedure is defined, and then passed (or returned) as an argument to be invoked at a time when some context is in effect that is different from the one in which the procedure was defined†. The standard way to deal with the problem of resolving free variables found in functional arguments is to create and

---

* It should be noted that few languages provide direct semantics for conversion of character-string data into external object references. To fill this gap in language semantics, many operating systems provide subroutines to perform the conversion. Such a subroutine typically takes a character string argument representing the name of some object, and returns a reference (sometimes called a pointer or an address) to that object.

† Accomplishing correct name resolution when a function containing references to free variables is passed or returned as an argument is known, in the programming language community, as the "FUNARG problem". In our case, a compiler or assembler has returned as its output value a procedure that contains free variables--unresolved outward symbolic references. Thus, we should expect that solutions to the FUNARG program might provide a clue how to proceed.

pass not a procedure, but a closure, consisting of the procedure and the context in which its names are to be interpreted. In the case at hand, a name-containing object is being interpreted, and we are trying to discover the closure that defines its symbolic naming context.

A simple (though not quite adequate, as we shall see approach to providing a closure is to require that each catalog that contains the object also contain entries for every symbolic name used in that object. Then, we design the context initializer so that whenever any object is discovered to require a symbolic name to be resolved, the context initializer should resolve that name by looking in the file system catalog in which it originally found that object. The author (or user) of an object that uses symbolic names is instructed that in order for the object to operate correctly, someone must prepare its containing catalog by installing entries for every name used by the object. These entries are the externally-provided declarations that replace the internal-to-the-object declarations whose absence caused context initialization to be needed in the first place.

Using the containing catalog as a context for resolving symbolic names handles part of the problem, but it fails to provide modular sharing. Consider the catalog of figure 5-18 named "root.users.Smith". Smith has in mind declaring that the name "b" should be bound to a program written by Lenox, which Smith can refer to as "root.users.Lenox.b". Unbeknownst to Smith, Lenox organized "b" in several pieces, one of which is named "a". If Smith binds the name "b" directly to Lenox's procedure, and the containing catalog is used as a context, Lenox's procedure will get the wrong "a" whenever it is invoked by Smith. What is really needed is an explicit closure, rather than an implicit one, so we conclude that we should arrange things as in figure 19, with each named procedure replaced by a closure object containing a pointer to the procedure, and a pointer to the appropriate context. Now, there is no problem about how to bind the name "b" in Smith's catalog: it can be bound to the closure for procedure b, as shown. With this arrangement, when Smith's procedure "a" calls on procedure "b" the name "b" will be resolved in Smith's catalog, and it will cause initialization of a new (addressing architecture) context for b that is based on the file system context of Lenox's catalog. When procedure "b" calls for "a" it will get the "a" bound in Lenox's catalog, as intended. Although we have described the problem in terms of procedures, the same problem arises for any name using object, and the same solution, binding to the object through a closure rather than directly, should be applied. The goal of modular sharing, namely that correct use of an object should not require knowing its internal naming structure, is then achieved.

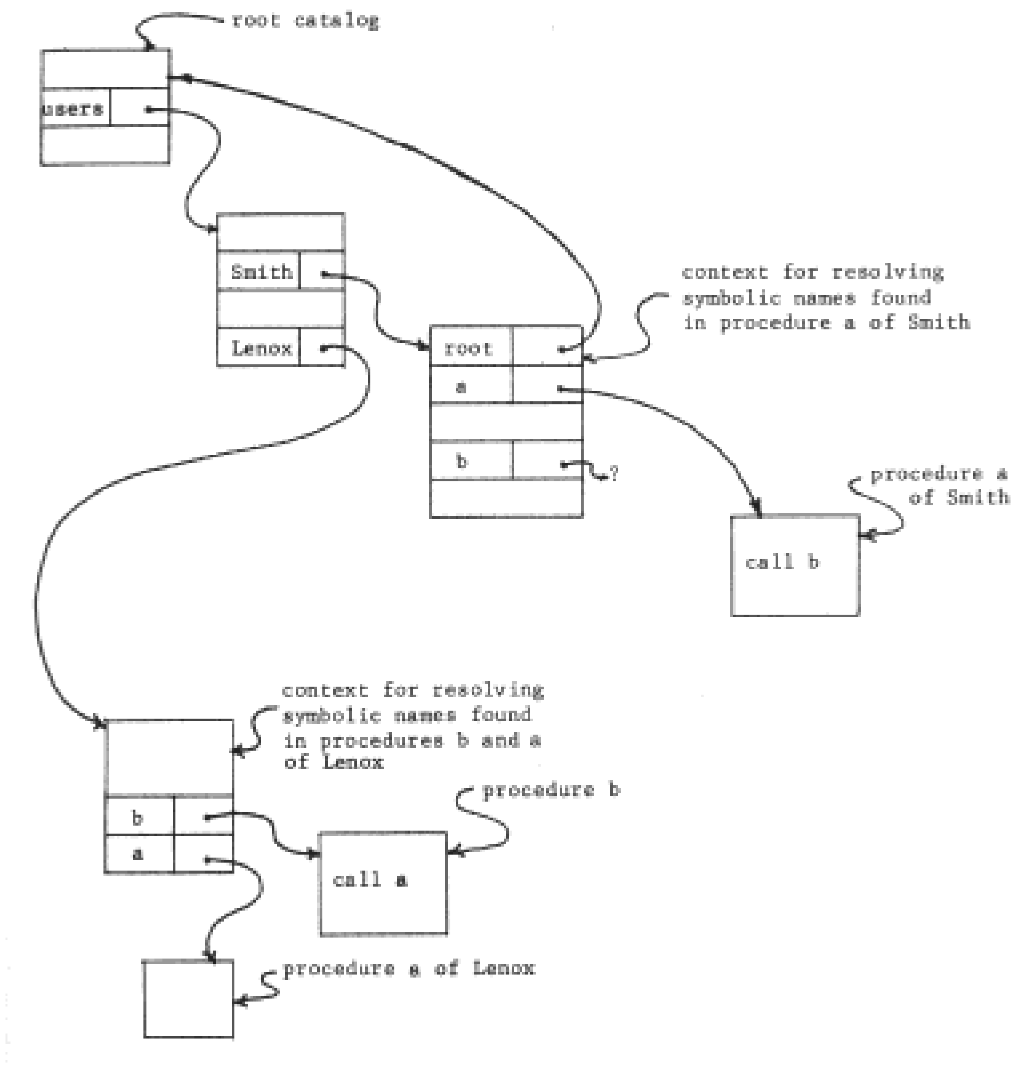


Figure 18 -- Sharing of a procedure in the naming network. If the name "b" in Smith's catalog is bound to procedure "b" in Lenox's catalog, the context initializer will make a mistake when resolving procedure b's reference to "a".

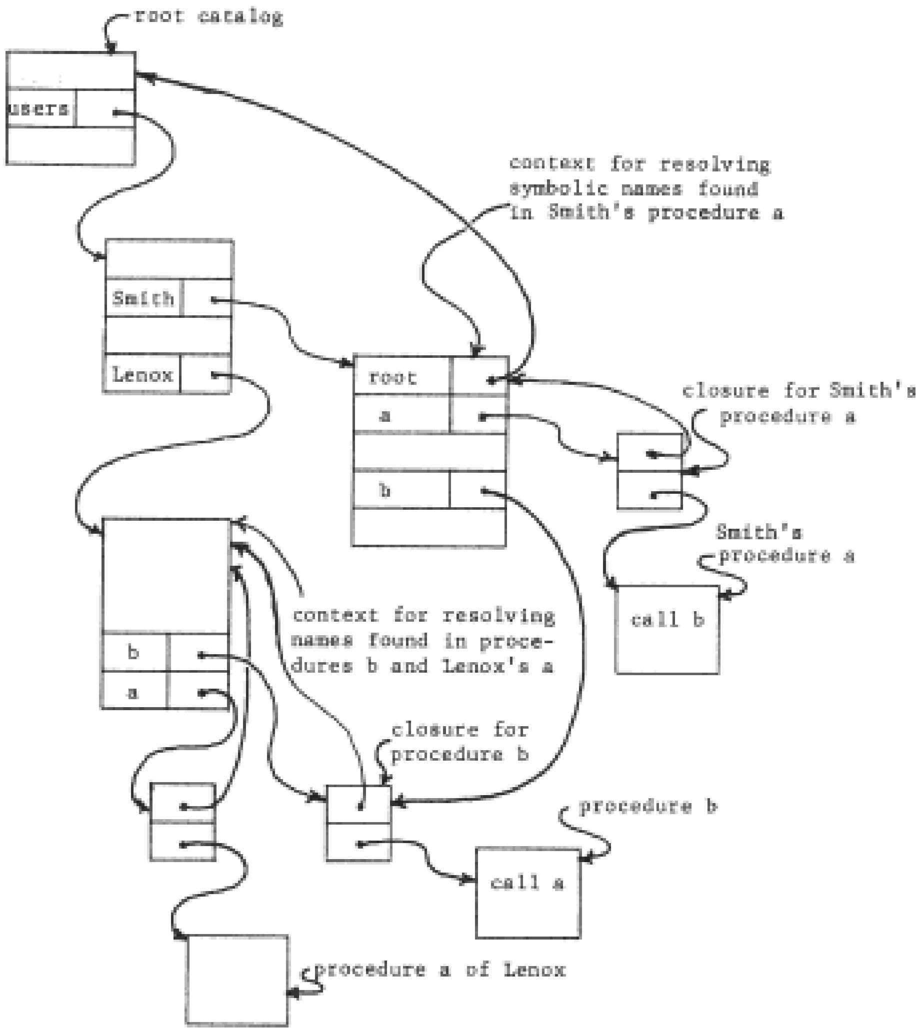


Figure 19 -- Addition of closures to allow sharing of procedures to work correctly.

5. Context initialization

The careful reader will note that we have not quite tied everything together yet: we have not shown how the context initializer of the addressing architecture makes systematic use of the file system closure mechanism. As a final step we should look into how this bridge between the addressing architecture and the file system might be organized. We approach this integration by considering one possible implementation of the context initialization programs involved.

First, let us suppose there is a catalog management function named "resolve", of two arguments, that looks up the symbolic name provided as its first argument in the catalog that has the object number supplied as the second argument. Thus to look up the name "xyz" in the catalog with object number 415, one would write

```
xyznum = resolve ("xyz", 415);
```

and upon return from "resolve", xyznum would contain the object number of "xyz". Actually, we must be a little careful here: xyznum would actually contain the object number of the file system closure of object "xyz".

The next step is to recognize explicitly the operation of making an object addressable by a running processor. This operation involves three steps:

1) creating and initializing an addressing architecture context for this processor to use when interpreting names found in the object. Let us suppose that the prototype context information is tucked away somewhere inside the object itself, in a standard, easy-to-find place.

2) inserting in this processor's closure table the addressing closure for this object; that is, the association between the object's object number and the object's just-created addressing context.

3) obtaining from the file system closure for the object the actual object number, for use by the processor in addressing the object.

Thus, we might imagine a context initialization program named "install" that would take as an argument the object number of a file system closure, and would go into the closure, find the object itself and perform those three steps. Step one, creating and initializing an addressing architecture context, may require resolving symbolic names (assuming some names are to be bound in advance of execution rather than on demand) so "install" will need to call on "resolve", giving as the second argument the catalog found in the file system closure for the object being installed. For each name so resolved, "install" will also have to call itself recursively, so as to make that object addressable, too. If binding on demand is involved, install should leave in the addressing context a copy of the object number of the file system closure, so that later demand binding faults can be correctly resolved. Figure 20 illustrates a simplified sketch of the function "install".

```
install (closure-num):  begin;
                        proc-num := closure-num.second part;
                        file-context-num := closure-num.first-part;
                        addr-context-num := create-new-object;
                        "examine the procedure proc-num to find
                        the prototype of its addressing context";
                        "initialize addr-context from that prototype";
                        "place a copy of file-context-num in addr-context";
                        "identify the symbolic names in the prototype";
                        for each such name do;
                            if name is to be bound now then do;
                                cℓ-num := resolve (name, file-context-num);
                                p-num := install (cℓ-num);
                                "insert p-num in addr-context";
                                od;
                            else
                                "insert bind-on-demand flag in addr-context";
                            od;
                        "insert proc-num and addr-context-num in
                        closure table";
                        return proc-num;
                        end;
```

Figure 20 -- Outline of the "install" function of the context initializer.

We might expect that the initial call to "install" comes, say, from an interactive program that has just read a line from the user's terminal, and discovered that the line contains the name of some object on which computation should be performed. The program might call "resolve" to convert the symbolic name to a closure object number, specifying as a context for this symbolic name resolution the current working catalog, Then it would call "install", possibly triggering a wave of recursive calls to "install", and it could then manipulate the object as required. It is instructive to follow through this sequence in detail for the file system catalog arrangement of figure 19, assuming that the working catalog is "root.users.Smith", and that Smith has typed a command that calls for execution of a program named "a". In following the sequence, note that we have provided, in a single mechanism, independent contexts for resolutions of symbolic names typed at the terminal and for symbolic names provided by the programmer; and that even when different programmers happen to use the same symbolic name for different objects, use of both their programs as components of the same subsystem is still possible.

This completes our conceptual analysis of higher-level naming systems. Table III summarizes the objectives that we have identified and also the file system facilities that implement those objectives. It remains for us to look at a variety of implementation strategies actually used in practice, most of which consist of shortcuts that abridge one or more of the objectives of the naming systems of figures 12 and 19.

Table III -- The objectives of a file system, and the facilities used to accomplish them.

Objectives	file system facilities					
	single catalog system	multiple catalog system	working catalog register	naming network	catalogs used as closures	distinct closure objects
human-oriented names	yes	yes	yes	yes	yes	yes
multiple users	no	yes	yes	yes	yes	yes
shared contexts	no	no	yes	yes	yes	yes
selectively shared contexts	no	no	no	yes	yes	yes
distinguish intent of programmer and user	no	no	no	no	yes	yes
modular sharing	no	no	no	no	no	yes

D. Implementation considerations

Up to this point we have developed two models of name binding that relentlessly pursue every implication and admit no compromise. The results are naming structures more sophisticated than any encountered in practice, although every portion of the structures described has appeared in some form in some system. In this section we explore some pressures that lead to compromise, and also investigate the effects of compromise to see which simpler structures might be acceptable for certain situations.

1. Lost objects

One potential trouble with naming networks is called the "lost object" problem. When a user deletes a binding in a catalog, the question arises of whether or not the file system should destroy the formerly referenced object and release the resources being used for its representation. If there are other catalogs containing bindings to the same object, then it should not be destroyed, but there is no easy way to discover whether or not other bindings exist. One approach is not to destroy the object, on the chance that there are other bindings, and occasionally leave an orphan that has no catalog bindings. If the system has a modest amount of storage it is then feasible to scan periodically all catalogs to mark the still accessible objects, and then sweep through storage looking for unmarked orphans, a technique known as "garbage collection". A substantial literature exists on techniques for garbage collection [Knuth, 1968], but these techniques tend not to be applicable to the larger volume of storage usually encountered in a file system.

An alternative approach is the following: when an object is created, the first binding of that object in some catalog receives a special mark indicating that this catalog entry is the "distinguished" entry for that object. If the user ever asks to delete the distinguished entry, the file system will also destroy the object*. This alternative approach burdens the naming system with the responsibility of remembering, in addition to a name-to-object binding, whether or not the entry is distinguished. This burden is significant: the mechanisms of naming and those of storage allocation are being tangled; the catalog has become the repository for an attribute of the object that has nothing to do with its name. Even when "garbage collection" is used there is a subtle entanglement of naming with storage allocation: that strategy calls for destruction of objects whenever they become nameless. Entanglement of mechanisms with different goals is not necessarily bad, but it should always be recognized. As we shall see in the next section, it can easily get out of hand.

Finally, a more drastic approach to avoiding lost objects is to eliminate the multiple bindings completely: require that each object appear in one and only one catalog. Then, when the binding is deleted, the object can be destroyed without question. But this constraint has far-reaching consequences for the naming goals. The naming network is restricted to a rooted tree, called a naming hierarchy. Although any object in such a hierarchy can refer symbolically to any other object, it can do so only by expressing a path name starting either from its own tree position or from the root, and thereby embedding the structure of the naming hierarchy in its cross references. User-dependent bindings become impossible.

Also, the constraint is more drastic than necessary. One can allow non-catalog objects to appear in as many catalogs as desired, and maintain with the representation of each object a counter (the reference count) of the number of catalogs that contain bindings to the object. As bindings are created or deleted, the counter can be updated, and if its value ever reaches zero, it is time to destroy the object†. (This scheme would fail if

* This approach was used, for example, in the CAL time-sharing system [Lampson and Sturgis, 1976].

† This strategy of reference counts to allow a file to appear in many catalogs was used in the file system for the UNIX time-sharing system [Ritchie and Thompson, 1974].

applied to the more general naming network. Consider what would happen in figure 16 if the only binding to the structure shown were the pointer in the working catalog register, and that binding were destroyed. Since all of the objects in the figure have at least one binding from other objects in the figure, the reference count would not go to zero, and the entire recursive structure would become a lost object*.)

2. Catalogs as repositories

In many real systems, to minimize the number of parallel mechanisms and to allow symbolic names to be used for control and in error messages, the catalog is used as a repository for all kinds of other attributes of an object besides control of its destruction†. These other attributes are typically related to physical storage management, reliability, or security. Some examples of attributes for which some repository is needed are:

a) the amount of storage currently utilized by the object,

b) the nature of the object's current physical representation,

c) the date and time the object was last used or changed,

d) the location of redundant copies of the object, for reliability,

e) a list of users allowed to use the object, and what modes of use they are permitted.

f) the responsible owner's name, to notify in case of trouble.

* The CAP system actually used this approach, on the basis that abandoned recursive structures will occur infrequently, and that occasional reload of the entire contents of the file system from backup copies will have the effect of reclaiming the last storage. A mixed scheme has been suggested by R. Needham, but never implemented. The idea would be to use both distinguished entries and reference counts, and to refuse to delete a distinguished entry if non-distinguished entries still existed, as indicated by a reference count greater than one [Needham and Birrell, 1977].

† In fact, the UNIX time-sharing system appears to be the only widely-used system that completely avoided the repository functions [Ritchie and Thompson, 1974].

The most significant effect of this merging of considerations of naming with considerations of physical storage management, reliability, and security is this: to provide control there should be only one repository for the attributes of any one object. Therefore, systems with this approach usually begin with the rule that there can be no more than one catalog entry for any one object. Thus the form of the naming network is restricted to that of a rooted tree, or naming hierarchy, with the ills for naming mentioned in the previous section. The use of the catalog as a general repository indeed distorts the structure of the naming system. However, there are two refinements that have been devised to restore some of the lost properties, indirect catalog entries and search rules.

3. Indirect catalog entries

Some systems provide an ingenious approximation to a naming network within the constraint that there be a single repository for each object. They begin with a naming hierarchy, as described above, but they permit two kinds of catalog entries. A direct entry provides a binding of a name to an object and its attributes, as usual. Exactly one direct entry appears for each object. An indirect entry provides a binding of a name to a path or tree name of some object elsewhere in the catalog hierarchy. The meaning of such an indirect entry is that if the user attempts to refer to an object with that name, the references should be redirected to the object whose path name or tree name appears in the indirect entry. There can be any number of indirect entries that ultimately lead to the same object*. Indirect entries provide most of the effect of a naming network: a single object may appear in any of several contexts. Yet the systematic use of indirect entries can confine the embedding of path names or tree names to

---

* Most such systems permit the possibility that the target of the redirected reference can redirect the reference to yet another catalog entry. In such cases, protection must be provided to prevent the name interpreter from going into a loop when a careless user leaves two indirect entries referring to each other.

catalogs. Further, a convention can be made that all symbolic references made by an object are to be resolved in the context consisting of the catalog in which that object's direct entry appears. In effect, this convention provides an automatic rule for associating a procedure with its symbolic naming context, and eliminates the need for an explicit closure to make the association*. Figure 21 illustrates the situation of figure 19, except that a naming network allowing indirect entries is used†.

---

* It also means that any one procedure can be associated with one and only one context, whereas with explicit closures, several closures could be provided for a single procedure, each naming different contexts.

† The CTSS system was probably the first to provide indirect catalog entries, doing so under the name "links". IBM's TSS/360 and Honeywell s Multics also were organized this way. The CAL time-sharing system allowed both indirect entries ("soft links") and any number of direct entries ("hard links").

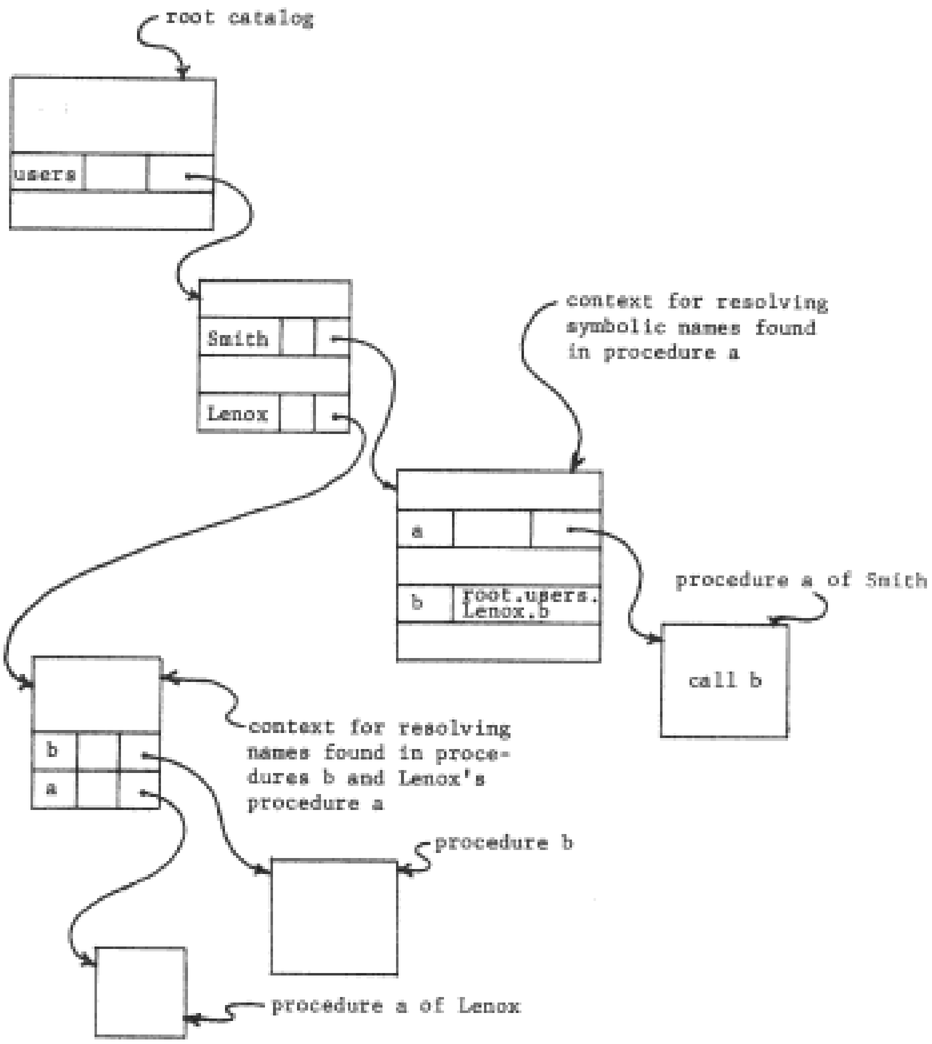


Figure 21 -- Sharing of procedures in a naming hierarchy with indirect catalog entries. The name «b» in Smith's catalog is bound to the tree name of Lenox's procedure, rather than its unique identifier. Since in a naming hierarchy the root catalog is distinguished, Smith no longer needs a binding for it; the name interpreter is assumed to know its unique identifier.

4. Search rules

Yet another approach to operation within the constraint of one catalog entry per object and a hierarchical naming tree is to condition the name interpreter to look in several different catalogs when resolving a name. Thus, for example, many systems arrange that names are looked up first in the user's working catalog, and failing there, in some standard library catalog. Such a simple system correctly handles a large percentage of the outward name references of traditional programs, since many programs call on other programs written by the same programmer or else universally available library programs.

The search rule scheme becomes more elaborate if other patterns of sharing are desired. One approach allows the user to specify that the search should proceed through any sequence of catalogs, including the working catalog, the catalog containing the referencing object, and arbitrary catalogs specified by path name. Further, a program may dynamically change the set of search rules that are in effect. This set of functions is intended to provide complete control over the bindings of outbound programs and data references, and allow sharing of subsystems in arbitrary ways, but the control tends to be clumsy. Unintended bindings are common, since a catalog belonging to another subsystem may contain unexpected names in addition to the ones for which the catalog was originally placed in the search path, yet every name of the catalog is subjected to this search. Substitution of one object for another can also be clumsy, since it typically requires that the search rules be somehow adjusted immediately before the reference occurs, and returned to their "normal" state before any other name reference occurs. To make this substitution more reliable, an artificial reference is sometimes used, with no purpose other than to force the search at e convenient time, locate the substituted object, and get its bindings installed for later actual use*.

Table IV summarizes the effect of using catalogs as repositories, indirect entries, and search rules on the various objectives that one might require of a file system and the reference name resolution ability of the context initializer for the addressing architecture.

* The search rule strategy described here is essentially that used in Multics [Organick, 1972].

Table IV -- Effect on objectives of various implementation strategies.

Objective	model file system	distinguished catalog entries	general repositories, naming hierarchy	indirect entries	search rules
no lost objects	no	yes	yes	yes	no effect
independence of naming and storage allocation	yes	no	no	no	no
control of object attributes	no	no	yes	no effect	no effect
cross references without knowledge of tree structure	yes	yes	no	yes	yes
multiple contexts for an object	yes	yes	no	no	yes, but clumsy
<u>Resolving</u> <u>reference</u> <u>names</u>: easy to predict result	yes	yes	yes	yes	no
easy to specify	no	no	no	no	yes
correct operation under local rearrangement	yes	sometimes	no	no	no
old object still available to old users if reference name is rebound	yes	sometimes	no	no	no
precise substitution	yes	yes	no	yes	no

E. Research directions

Name binding in computer systems, as should by now be apparent, is relatively ad hoc and disorganized. Conceptual models capture only some parts of what real system designers face. The simple model of names, contexts, and closures can be used to describe and better understand many observed properties and misbehaviors of practical systems, but one's intuition suggests that there should be more of an organized approach to the subject. Since the little bit of systemization so far accomplished grows out of studying equivalent, but smaller scale, problems of the semantics of naming within programming languages, one might hope that as that study progresses, further insight on system naming problems will result.

Apart from developing high-level conceptual models of name binding in file systems and addressing architectures, there are several relatively interesting naming topics about which almost nothing systematic is known, and the few case studies in existence are more intriguing for their irregularity, inconsistency, and misbehavior than for guidance on naming structure. These topics arise whenever distributed systems are encountered.

A system is distributed from the point of view of naming whenever two or more parallel and independently operating naming systems are asked to cooperate coherently with each other. For example, two or more separate computers, each with its own addressing architecture and file system, are linked by a communication network that allows messages to flow from any system to any other. The unsolved questions that arise surround preparing the addressing architectures and file systems so that:

1) Within each system the goals of sharing named objects are met essentially as in the models of this chapter.
2) Object sharing can occur between systems, so that an object in one system can have as constituents objects physically stored in other systems. Sharing between systems seems to involve maintaining contexts that work with multiple, independent name generators.

3) Objects can, if desired, be permanently moved from one system to another without the need to modify cross references to and from that object, especially cross references arising on systems not participating in the move.
4) Operations can proceed smoothly and gracefully even if some systems are temporarily disabled or are operating in isolation. This goal leads to consideration of keeping multiple copies of objects on different systems, and produces some real questions about how to name these multiple copies. It also means that name generation within any one system must be carried out independently of name generators on other systems, and it leads to problems of keeping name generators coordinated.

These descriptions of goals barely scratch the surface of the issues that must be explored, and until there are more examples of distributed systems that attempt coherent approaches to naming, it will not be clear what the next layer of questions is.

There are several activities underway that could shed some light on these questions. At the University of California at Irvine, a system named D.C.S. (for Distributed Computing System) has been designed and is the subject of current experimentation [Farber, et al., 1973]. At Bolt, Beranek, and Newman, a program named RSEXEC was developed that attempts to make all the file systems of a network of TENEX computers look to the user as a single, coherent file system [Thomas, 1973]. The Advanced Research Projects Agency of the U. S. Department of Defense has developed a "virtual file system" that operates on a variety of networked computers as part of a research program known as the National Software Works [Carlson and Crocker, 1974].

The current direction of hardware technology, leading to much lower costs for dedicated computers and networks to interconnect them, is making feasible a decentralization that has always been desired for administrative convenience, and one should expect that the problems of inventing ways of providing coherence across distinct computers that run independent naming systems will rapidly increase in importance.

Acknowledgements

The initial view of organization of this chapter grew out of conversations with M.D. Schroeder. Extensive and helpful technical comments on early drafts were made by D. Clark, L. Svobodova, G. Popek, and A. Jones, and observations about pedagogical problems were provided by J.C.R. Licklider and M. Hammer. Suggestions on style were made by K. Sollins. For several years, undergraduate students in the M.I.T. subject "Information Systems" suffered through early, somewhat muddled attempts to explain these ideas.

Sections A.2, A.3, and A.4 of the introduction follow closely the development by A. Henderson in his Ph.D. thesis [Henderson, 1975, pp. 17-26], with his permission.

Suggestions for further reading

1) The mechanics of naming [Henderson, 1975; Fabry, 1974; Dennis, 1965; Redell, 1975; Clingen, 1969].

2) Case studies of addressing architectures [Organick, 1972; Bell and Newell, 1971; Organick, 1973; Radin and Schneider, 1976; Needham, 1972; Watson, 1974, Chapter 2].

3) Case studies of file system naming schemes [Murphy, 1972; Bensoussan et al., 1972; Lampson and Sturgis, 1976; Needham and Birrell, 1977; Ritchie and Thompson, 1974; Organick, 1972; Watson, 1974, Chapter 6].

4) Object-oriented systems [Wulf et al., 1974; Janson, 1976; Redell, 1974].

5) Historical sources [Holt, 1961; Iliffe and Jodeit, 1962; Dennis, 1965; Daley and Dennis, 1968].

6) Semantics of naming in systems [Fraser, 1971].

References

Bell, C.G., and Newell, A., Computer Structures, McGraw-Hill, New York (1971). (SR)

Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics virtual memory: concepts and design," CACM IS, 4 (May, 1972), pp. 308-318. (A.4, B, SR, App)

Bratt, R.G., "Minimizing the Naming Facilities Requiring Protection in a Computing Utility," S.M. Thesis, M.I.T. Dep't of Elec. Eng. and Comp. `Sci. (Sept., 1976). (Also available as M.I.T. Lab` for Comp. Sci. Technical Report TR-156.) (App)

Carlson, W.E., and Crocker, S.D., "The impact of networks on the software marketplace," Proc. IEEE Electronics and Aerospace Convention (EASCON 1974), pp. 304-308. (E)

Clingen, C.T., "Program naming problems in a shared tree-structured hierarchy," NATO Science Committee Conference on Techniques in Software Engineering 1, Rome (October 27, 1969) (SR)

CODASYL (Anonymous), Data Base Task Group Report, Association for Computing Machinery, New York (1971). (C.2)

Crisman, P., Ed., The Compatible Time-Sharing System: A Programmer's Guide, 2nd ed., M.I.T. Press, Cambridge (1965). (C.2)

Daley, R.C., and Dennis, J.B., "Virtual memory, processes, and sharing in Multics," CACM 11, 5 (May, 1968), pp. 306-312. (B.2,B.3, SR, App)

Dennis, J.B., "Segmentation and the design of multiprogrammed computer systems," JACM 12, 4 (October, 1965), pp. 589-6O2. (A.1, SR, SR)

Fabry, R.S., "Capability-based addressing," CACM 17, 7 (July, 1974), pp. 403-412. (SR)

Falkoff, A.D., and Iverson, K.E., APL\360: User's Manual, IBM Corporation, White Plains, New York (1968). (A.4)

Farber, D.J., et al., "The distributed computing system," Proc. 7th IEEE Computer Society Conf. (COMPCON 73), pp. 31-34. (E)

Fraser, A.G., "On the meaning of names in programming systems," CACM 14, 6 (June, 1971), pp. 409-416. (A.1, SR)

Henderson, D.A., Jr., «The binding model: A semantic base for modular programming systems,» Ph.D. thesis, M.I.T. Dep't of Elec. Eng. and Comp. Sci. (February, 1975). (Also available as M.I.T. Project MAC Technical Report TR-145.) (SR)

Henderson, P., and Morris, J.H., "A Lazy Evaluator," Proc. 3rd ACM Symposium on Principles of Programming Languages, (January, 1976), pp. 95-103. (B.3)

Holt, A., "Program organization and record keeping for dynamic storage allocation," CACM 4, 10 (Oct., 1961), pp. 422-431. (A.1,SR)

IBM (Anonymous) Reference Manual: 709/709O FORTRAN Operations,

IBM Corporation, White Plains, New York (1961). C28-6O66 (A.4, A.4)

Iliffe, J., and Jodeit, "A dynamic storage allocation scheme," Computer Journal 5, (Oct., 1962), pp. 2OO-209. (A.1.SR)

Janson, P.A., "Removing the Dynamic Linker from the Security Kernel of a Computing Utility," S.M. Thesis, M.I.T. Dep't of Elec. Eng. and Comp. Sci. (June, 1974). (Also available as M.I.T. Project MAC Technical Report TR-132.) (App)

Janson, P.A., "Using type extension to organize virtual memory mechanisms," Ph.D. Thesis, M.I.T. Dep't of Elec. Eng. and Comp. Sci. (Sept., 1976). (Also available as M.I.T. Lab. for Comp. Sci. Technical Report TR-167.) (SR)

Kilburn, et al.. "One-Level Storage System," IRE Trans. on Elec. Computers EC-11, 2 (April, 1962), pp. 223-235. (C.1)

Knuth, D., The Art of Computer Programming, Volume l/Fundamental Algorithms, Addison Wesley, Reading, Mass. (1968), Chapter 2. (D.1)

Lampson. B.W., and Sturgis, H.E., "Reflections on an operating system design," CACM 19, 5 (May, 1976), pp. 251-265. (C.2, D.1, SR)

Moses, J., "The function of FUNCTION in LISP," SIGSAM Bulletin (July, 197O), pp. 13-27. (A.4)

Murphy, D.L., "Storage organization and management in TENEX," AFIPS Conf. Proc. 41 I (FJCC, 1972), pp. 23-32. (B.2,SR)

Needham, R., "Protection systems and protection implementation,' AFIPS Conf. Proc. 41 I (FJCC, 1972), pp. 571-578. (B.2,SR)

Needham, R.M., and Birrell, A.D., "The CAP filing system," Proc. 6th ACM Symposium on Operating Systems Principles (November, 1977), pp. 11-16. (C.2,D.1,SR)

Organick, E.I., The Multics System: an Examination of its Structure, M.I.T. Press, Cambridge (1972). (D.4,SR,SR,App)

Organick, E.I., Computer System Organization, The B5700/B6700 Series, Academic Press, New York (1973). (SR)

Radin, G., and Schneider, P.R., "An architecture for an extended machine with protected addressing," IBM Poughkeepsie Lab Technical Report TR 00.2757 (May, 1976). (B,SR)

Redell, D.D., "Naming and protection in extendible operating systems," Ph.D. Thesis, Univ. of Cal. at Berkeley, (Sept., 1974). (Also available as M.I.T. Project MAC Technical Report TR-140.) (B,SR,SR)

Ritchie, D.M., and Thompson, K., "The UNIX time-sharing system," CACM 17, 7 (July, 1974), pp. 365-375. (D.1,D.2,SR)

Schroeder, M.D., and Saltzer, J.H., "A hardware architecture for implementing protection rings," CACM 15, 3 (Mar., 1972), pp. 157-170. (B.2,B.2)

Thomas, R.H., "A resource sharing executive for the ARPANET," AFIPS Conf. Proc. 42 (1973 NCC), pp. 159-163. (E)

Watson, R.W., Timesharing System Design Concepts, McGraw-Hill New York (1974). (SR,SR,App)

Wulf, W., et al., "HYDRA: The kernel of a multiprocessor operating system," CACM 17, 6 (June, 1974), pp. 337-345. (SR)

Appendix A: Case Study of Naming in Multics

The Multics system implements a shared-object addressing architecture and a variation of a direct-access file system. In doing so, this system illustrates an interesting set of design choices and compromises. To a close approximation, the designers of Multics had in mind achieving all of the naming objectives discussed in this chapter, but they had to work within the framework of modest extensions to an existing, fairly simple addressing architecture, that of the General Electric 635 computer. In addition, Multics was one of the first designs to be ventured in this area, and some of its design decisions would undoubtedly be handled differently today. For clarity in this case study, we shall examine only the user-visible naming facilities of Multics. We shall ignore the closely-related, but user-invisible multilevel memory management (paging) machinery, and also the closely-related information protection facilities. The reader should, however, realize that these three functions are actually implemented with integrated, overlapping mechanisms.

1. The addressing architecture of Multics

Multics implements a single, simple kind of object known as a *segment*. A segment is an array of 36-bit words, containing any desired number of words between zero and 262,144. An individual word within a segment is named by specifying its *displacement*, or distance from word zero of the segment, as in figure 22.) Although segments have unique identities, the addressing architecture does not use unique identifiers to name segments*. Thus there is no universal, underlying context for naming segments. Instead, Multics implements a large number of small segment-naming contexts, known as *address spaces*. Typically, one address space is

* The Multics file system, described below, maintains a unique identifier for each file, but that unique identifier cannot be used by the addressing architecture.

provided for each distinct active user of Multics, and an address space has the lifetime of a user terminal session.

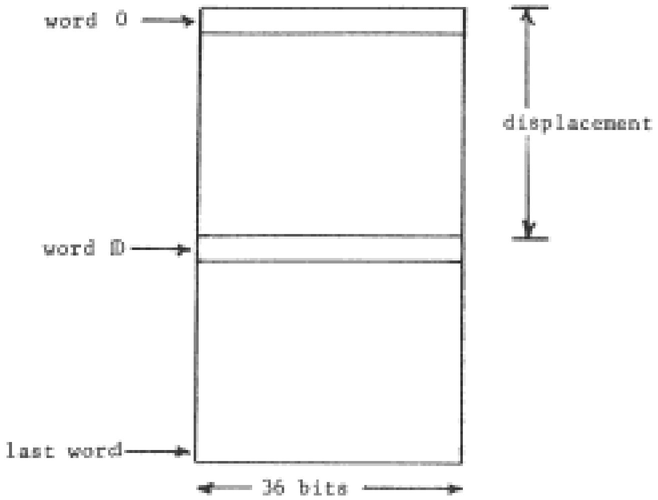


Figure 22 -- A Multics segment.

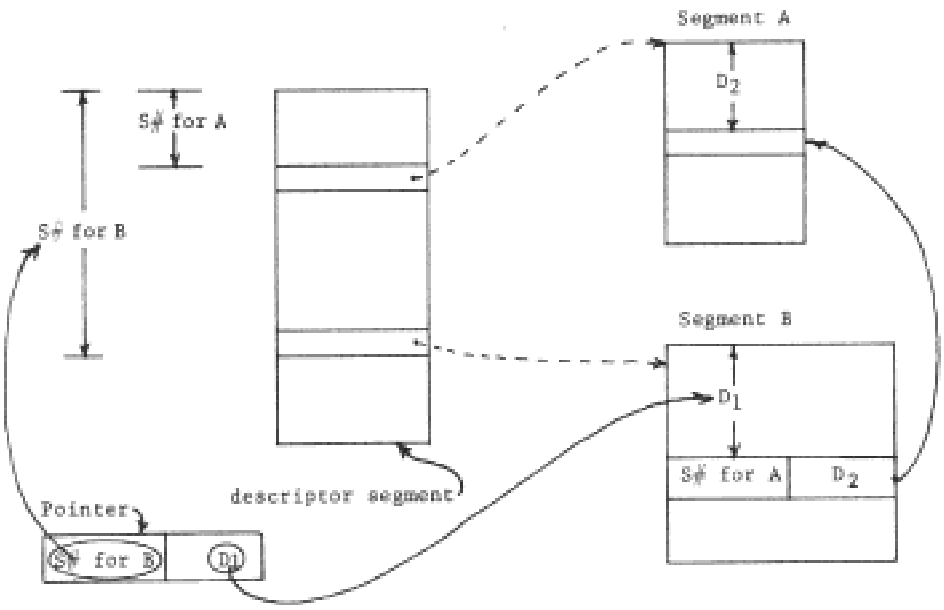


Figure 23 -- Pointer addressing in Multics. A pointer contains a segment number (S#), which is a displacement in the descriptor segment that identifies the lower-level (physical) address of the segment. The pointer also contains a displacement that identifies a word within the ultimately addressed segment. The pointer on the left refers to a word in the segment labelled B, and

that word happens to contain a pointer to a word in the segment labelled A.

The reason for such a choice is that each individual address space need be no larger than the number of segments actually used in the lifetime of a terminal session, and the names resolved by the address space can therefore be a small, compact set of integers, known as segment numbers. Only a few address bits are then needed to hold a segment number, and the context itself can be implemented as a small array of entries using the segment number as an array index: the context is called a descriptor segment, and a segment number is simply a displacement within some descriptor segment. (See figure 23.) Although the hardware architecture allows segment numbers up to 18 bits in length, typical address spaces have only a few hundred segments, and the software tables required to keep track of what those segment numbers mean have size limits of only a few thousand entries.

The compromise involved is that segment numbers are meaningful only in the context in which they were defined: if one user passes a segment number to another user, there is no reason to expect that number to have the same meaning in the other user's address space. Thus, segment numbers cannot be used to implement complex, linked structures that are shared between different users. On the other hand, sharing is still quite feasible since, as we saw in section B.2, when user-dependent bindings are a goal, one takes care not to embed object numbers in shared objects anyway. A second aspect of the compromise is that there is no a priori relationship between segment identities and segment numbers; when a job begins a new address space must be created and the segments of interest must be mapped (Multics literature uses the word initiated) into the new address space. Initiation represents a run-time cost, and requires supporting tables to keep track of which segments have been initiated.

One segment may contain a reference to another in two ways in Multics. The simplest way, but usable only in segments not shared with other address spaces and having a lifetime no greater than the current address space, is by means of a pointer which is just a segment number and displacement address stored in a standard format. (See figure 23.) Because segment numbers have different bindings in different address spaces, a more complex form of inter-segment reference is required if the containing segment is to be shared. For example, procedure segments need to refer to data segments and other procedure segments, but procedures are commonly shared. This second form of reference is by way of a per-procedure context, known in Multics as a linkage section. A pointer register in the processors known as the linkage pointer, points to the beginning of the linkage section, as shown in figure 24. An instruction that refers to an object outside the procedure uses for its operand address an indirect address, specifying some displacement relative to the linkage pointer. The linkage section contains at that displacement an ordinary pointer to the desired object.

If a second user shares the procedure segment, a second linkage section is involved, as in figure 25. Note that there are two address spaces involved in this figure, but that no pointers in either address space lead to the other. The only segments shared between the two address spaces contain no embedded pointers. In addition, all references to the shared segments are by way of pointers in either the processor or linkage section private to an address space, so a shared segment can have different segment numbers in the two address spaces.

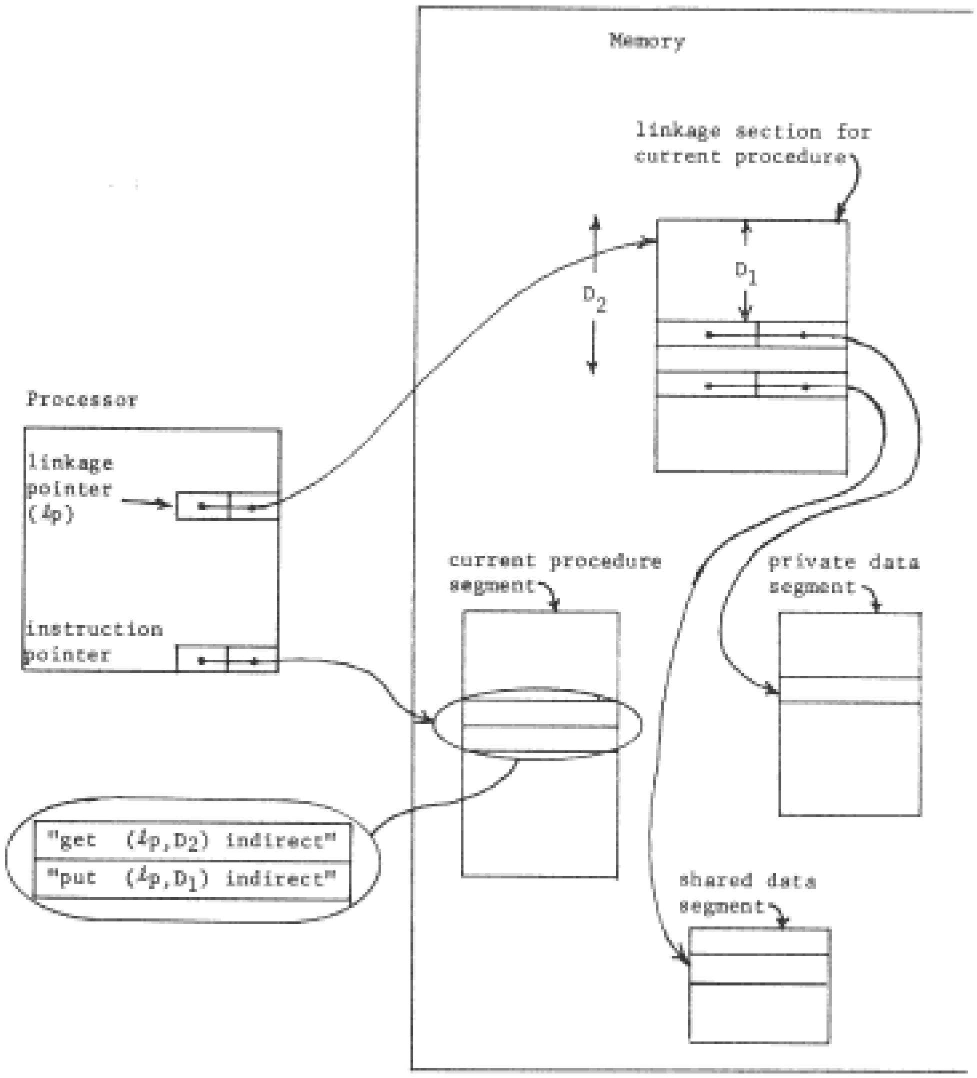


Figure 24 -- Inter-segment references in Multics by procedure. The operand address specifies indirect addressing via the Dth word of the segment that is the current target of the linkage pointer register. The linkage section is thus a context for the current procedure. For simplicity, the descriptor segment is not shown, but all four of the pointer references of the figure, it should be remembered, are interpreted using the descriptor segment as the context.

Finally, for speed, each processor contains eight pointer registers (actually, the linkage pointer is one of them) that a program may load with any desired pointer. Once such a register is loaded, that register can then be used as an operand address, thus avoiding the use of the slower indirect reference through memory.

When calling from one procedure to another, the value of the linkage pointer must be changed to point to the linkage section of the new procedure. This change is not accomplished by automatic hardware in Multics, but rather by a conventional sequence of instructions in the called program. A per address space table of closures, known as the linkage offset table, contains pointers to every linkage section, one for each procedure in the address space. One of the processor pointer registers conventionally contains a pointer that leads (after indirection through a stack segment, not of concern to us here) to the base of the linkage offset table. The conventional sequence upon entry to a procedure is then the following: sometime before the first intersegment reference of the procedure is encountered, use this procedure's segment number as a displacement in the linkage offset table to load the linkage pointer register with a pointer to the linkage section of this procedure. This operation is so similar to the corresponding operation of the addressing architecture developed earlier that figure 12 can be used directly to illustrate it, with a slightly different caption.

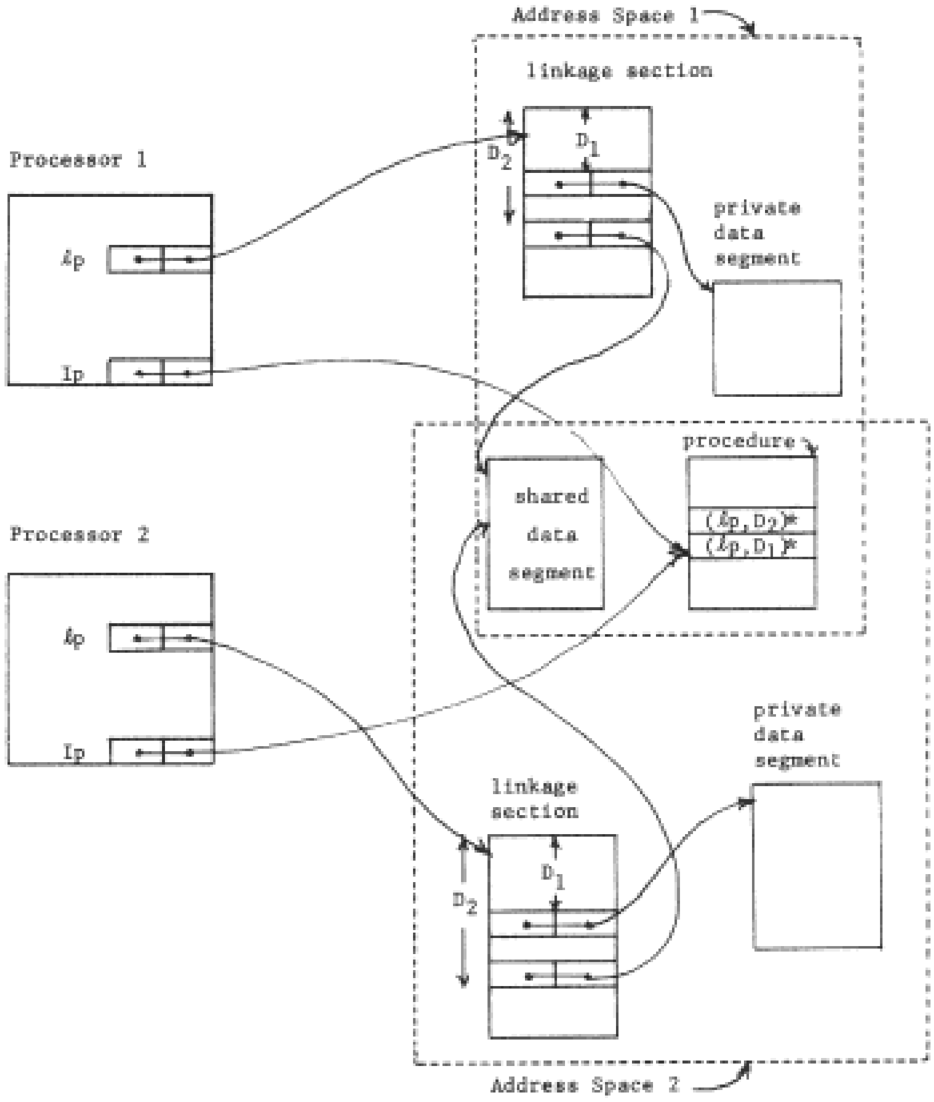


Figure 25 -- Shared address spaces in Multics. Hidden in each processor (and not shown in this figure) is a register (the <u>descriptor</u> <u>segment</u> <u>base</u> <u>register</u>) that contains the physical address of a descriptor segment (also not shown). The two processors above are using different descriptor segments, and thus different address spaces. However, these address spaces overlap in that a procedure segment and a shared data segment appear in both. Note that the shared segments can have different segment numbers in the two address spaces, yet all intersegment references work correctly. The notation (lp,D2)* means that the pointer addressed by the pair (lp,D2) should be used as an indirect address. This figure should be compared with figure 10, which illustrates a similar configuration.

2. The Multics file system

For its file system, Multics implements a direct-access hierarchy of named catalogs and files. The catalogs are known as directories, and for each catalogued object contain not only any number of synonymous names, but also are the repository for a physical storage map, a unique identifier, an access control list, reliability control, and resource accounting information. A root directory is "known to the system" and when a user program presents a tree name the file system resolves it by starting in that root directory. Every directory and file except the root is represented by exactly one direct entry, known as a branch, in some directory, so a strict hierarchy results. This strict hierarchy was chosen on the basis of simplicity--the lost object problem need not be considered--and expediency--using the directory as a repository seemed the quickest design to implement.

For flexibility, then, indirect directory entries are also provided, called links. Any number of indirect entries can lead to the same directory or file. A link is a directory entry that binds a directory entry name to an associated tree name that leads from the root directory to the desired object*. If the entry named by the link is another link, then that second link is followed; a link depth counter limits such successive links to ten, so as to catch links accidentally arranged in an infinite loop.

The file system maintains for each user a cell containing the absolute path name of some directory, called the user's working directory. One can then more briefly express the name of some objects by a relative path name that starts from the working directory. Names typed at the terminal are usually interpreted relative to the working directory,

* A link may actually contain an absolute path name; that is, a name that begins at the root and proceeds, possibly via other links, to lead to some named object in the directory hierarchy. (The difference between an absolute path name and a tree name is that a tree name may not involve links.) Allowing links to contain absolute pathnames makes links to directories useful; however the procedure that resolves tree names may become recursive.

although the user can also type in an absolute path name by using a distinctive syntax.

The pattern of use of the file system is as follows: suppose the user is in interactive communication with an application program, such as an editor, and he types to the editor the instruction to look at something in a file named "a". The editor takes the relative path name "a", and first passes it to a utility program that combines it with the name of the current working directory and returns an absolute path name, suitable for presentation to the file system. The editor then calls the initiate entry of the file system, presenting the absolute path name. The file system resolves this name through its directory hierarchy, following any links encountered along the way. Assuming it finds a file by that name, this user has permission to use it, and it has not previously been initiated, the file system then selects an unused segment number in the current address space, fabricates a descriptor segment entry, and returns the segment number to the editor. The file system also makes an entry in a table for this address space, the Known Segment Table, relating the unique identifier of the file, the chosen segment number and the directory in which the file was found. If the file in question had previously been initiated in this address space, its unique identifier would already be in the known segment table, and the segment number found there would be returned, rather than a new one. In either case, the editor program uses the returned segment number and an appropriate displacement to fabricate a pointer, and it then reads the contents of the file by direct reference to memory. The file of the file system has been mapped into a segment of the addressing architecture, and neither the file system nor the addressing architecture make any interpretation of the contents--they are simply an array of up to 262,144 36-bit words*. A file may be larger than 262,144 words, in which case the addressing architecture provides a window into some portion of the

---

* Interpretations such as "indexed sequential" files are provided by application level library programs that use the file system and addressing architecture as tools.

file, up to 262,144 words in size. The window can be moved by a call to the address space manager*.

3. Context initialization in Multics

The previous example of mapping of a file of the file system into a segment of the addressing architecture started with an interactively supplied file name. Initialization of procedure naming contexts also may involve a step of mapping a file system file into a segment of the addressing architecture.

Consider a procedure named "a", which contains within it a call to another procedure named "b"†. The compiler of "a" produces an object program consisting of the instructions to carry out procedure "a" and a prototype of the linkage section that will serve as that procedure's context during execution. Contained in that prototype linkage section is a dummy pointer, corresponding to the outbound reference to "b", and consisting of a flag (which will cause the hardware addressing architecture to signal a fault) and the character string "b"‡. Suppose that the segment containing procedure "a" and its linkage section have been previously mapped into the addressing architecture, and procedure "a" attempts to call "b" for the first time. Upon trying to interpret the dummy pointer, the addressing architecture will signal the fault, passing control to the context initializer program, known in Multics as the dynamic linker§. The

---

* Files larger than 262,144 words and movable windows have not yet been implemented as of Spring, 1978.

† Multics includes an entry-point naming scheme that allows one program to call on a named entry point, say "c", of a named procedure, say "b". The following discussion ignores that feature; strictly it describes what happens if program "a" calls entry point "b" of procedure "b", which is the most common case, and the default if only one name is supplied by the calling program.

‡ Note that, because segment numbers for a given procedure are different in different address spaces, the compiler cannot place an operational pointer in the prototype linkage segment.

§ As may be apparent by now, the word <u>link</u> is used in Multics for at least two unrelated but similar concepts: indirect catalog entries and the pointers of procedure contexts. Sentences containing the word "link" do not always come supplied with closures, so confusion is common.

dynamic linking program retrieves from the dummy pointer the character string name "b", resolves it into a segment number, replaces the dummy pointer with a real one containing the segment number and displacement of the entry point, and restarts the interrupted procedure "a". Since the dummy pointer has been replaced with a real one, the procedure call will now work correctly. Further, if procedure "a" ever calls procedure "b" again, that later call will be resolved at the speed of the addressing architecture, rather than the speed of the dynamic linker. The outbound reference from "a" to "b" has been bound, at the addressing architecture level, for the lifetime of the address space.

The interesting aspect of the dynamic linking sequence is, of course, how the dynamic linker resolves the name "b" into a segment number. As we noted in the general discussion of context initialization, the goal is to resolve the name "b" according to the intent of the subsystem writer who chose to use "a", rather than the current interactive user, so the name resolution takes this goal into account, though imperfectly. The first step is to look in a table for this address space, called the reference name table, which the dynamic linker maintains for this purpose. Each time it maps a procedure or data object into the address space, the dynamic linker places the name of that procedure and its segment number in the reference name table. Thus, the name "a" is certain to be in the reference table, and if it has been previously called from within the address space, so will the name "b". The primary purpose of the reference name table is to avoid the expense of following a path name through the file system directory hierarchy more than once per procedure; often, procedures of a subsystem are used by more than one other procedure of that subsystem. A related effect of the reference name table is to capture a single, address-space-wide binding for a name, so as to guarantee that all callers for that name consistently get the same result, independent of what we shall see is a potentially variable strategy used by the dynamic linker for name binding. The difficulty with the reference name table is that when two independently conceived subsystems are invoked in the same address space, they share the single reference name table, and name conflict can occur.

So much for the reference name table. What if the name "b" is not found there, indicating that procedure "b" has never been mapped into this address space? Actually, the check of the reference name table is properly viewed as the first search rule of a multistep search for the procedure named "b". If that rule fails, the search is carried on to the file system, using the assumption that the procedure "b" is to be found in some file named "b" in the directory hierarchy. As an approximation to the idea that the directory containing a procedure can be used as a closure for that procedure's outbound references, the next step of the search is to look for the name "b" in the directory that the calling procedure, "a", came from. (Remember that the known segment table contains for each segment, among other things, the directory that the segment was found in. One reason was to allow this search rule to be implemented easily.) The dynamic linker thus attempts to initiate a segment named "b" from that same directory. If that attempt succeeds, it creates an empty linkage section for "b" to use as its context, places a pointer to that linkage section in the linkage offset table for this address space, installs "b" in the reference name table, and then proceeds as if "b" had always been in the reference name table. If it fails, it continues with the search. The next step is to try the current working directory, and then to begin going down a list of tree names of other directories, attempting to initiate the file "b" in each of those directories in turn. These directories are usually system and project libraries, though a user may place the tree name of any directory in his search list. The user may also rearrange the search rules in any order, for example by placing the search of the reference name table last, or even deleting it from the list. (Note that even if the reference name table is not part of the search, it is used to prevent duplicate context initialization.)

Probably the chief virtue of the search rule strategy of Multics is that its imprecision in binding can be exploited to accomplish things for which specific semantics would otherwise need to be provided. For example, the proprietors of the system library can move a library program from, say, the normal library to a library containing obsolete procedures. Users of that program will discover the move through a failure report from the dynamic linker, but they can recover simply by adding the obsolete library directory to their search rules, without changing any programs or links. For a different example, a procedure can be moved from a project library directory to the system library directory; users of that procedure will continue to find it in the normal course of a search without any change to their programs or operation at all.

On the other hand, as was suggested earlier, in the general discussion of adjustable search rules, they are a fairly clumsy and error-prone way to accomplish user-dependent bindings. Probably the primary reason that search rules have persisted in the Multics design, rather than being superceded or augmented by addition of catalogued closure objects, is that most other operating systems, both previous to and contemporary with Multics, have used some form of search at least for system libraries, so this pattern of operation is familiar to both system designers and users.

An interesting complication arises from the combination of design compromises in the addressing architecture and file system of Multics. The addressing architecture provides a very large address space, in which it is feasible to map simultaneously several closely communicating but independently conceived subsystems. On the other hand, the address space survives only for the time of a user's terminal session (or batch job) and therefore inter-procedure linkages cannot be bound in advance; they must be initialized dynamically. But dynamic initialization involves a search, which can be relatively expensive. As a result, two strategies have evolved for minimizing the number of times the dynamic linker is actually invoked.

First, a utility routine, known as the binder, will take as input several procedures, and combine them and their linkage sections into a single procedure and linkage section, with all cross-references among the set resolved into internal displacements rather than pointer references. This new single procedure is written out into a single file, and when used is mapped into a single segment of the addressing architecture. Using the binder eliminates a large percentage of dynamic linking operations, but if one library procedure is used in two different subsystems, to avoid dynamic linking the shared procedure must be copied into both of them.

Second, the conventional way of using the Multics system is to leave all procedure contexts, once initialized, in place for the duration of the terminal session or batch job, so that if the procedure should be used again as part of another command interaction or job step, it will not be necessary to repeat initialization of its context. Thus the interprocedure links are persistent.

Although at first glance this persistence seems to be exactly right, actually it is desirable to be able to reverse or adjust these bindings occasionally. For example, when a programmer is testing out a newly-written set of procedures, and discovers an error, the usual approach is to correct the symbolic version of the procedure, and then recompile it. Conventionally, the compiler overwrites the old procedure with the new one, so the procedure's segment number does not change, but occasionally the entry point of the procedure moves to a new displacement, as a result of fixing the error.

When that happens, any old pointers to this procedure found in linkage sections of other procedures become obsolete, and should be readjusted. Multics does not attempt to maintain the tables and backpointers that would be required to automatically readjust interprocedure linkage pointers, so when the displacement of an entry point changes the usual result is an attempt to enter the procedure at the wrong displacement, followed by some rather baffling failure of the application. The user can request a fresh address space at any time, and that fixes the problem by initializing a fresh set of contexts, all consistent with one another. Unfortunately, completely initializing a fresh address space is a relatively expensive operation. In addition, a beginning programmer typically encounters the first example of an incorrect, persistent interprocedure link long before he is prepared to understand the complex underpinnings that cause it; many programmers consider the need to occasionally request a fresh address space to be a mystery as well as a nuisance.

4. Bibliography on naming in Multics

Primary sources (original research papers) [Daley and Dennis, 1968; Bensoussan, Clingen, and Daley, 1972; Bratt, 1975; Janson, 1974].

Secondary sources (tutorials and texts) [Organick, 1972; Watson, 1974].