M.I.T. LABORATORY FOR COMPUTER SCIENCE December 9, 1976

Computer Systems Research Division Request for Comments No. 131

NAME BINDING IN INFORMATION SYSTEMS

by J. H. Saltzer

Attached is the first complete draft of some material I have been developing for subject 6.033 (Information Systems). There are two reasons for distributing it at this time: 1) I am interested in any comments you may have, at any level--concepts, approach, pedagogy, examples, historical accuracy, grammar, or whatever. 2) It may be relevent to our interest in distributed systems. One aspect of that research is to figure out how to extend the material of these notes to include the case of coherent naming across decentralized systems with independent name generators.

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS 02139

6.033 -- Information Systems

November, 1976

5. NAME BINDING IN INFORMATION SYSTEMS

CONTENTS

Overview 5-3

Glossary 5-3

A. Introduction

1. Names in computer systems 5-5

2. Some examples of existing naming systems 5-14

3. The need for names with different properties 5-21

4. Plan of study 5-24

B. An architecture for addressing shared objects 5-25

1. Multiple naming contexts 5-29

2. Larger contexts and context switching 5-36

3. Binding on demand and binding from higher level contexts 5-44

C. Higher level naming contexts, or file systems

1. Direct-access and read-write organizations 5-49

2. Multiple catalogs and naming networks 5-56

3. The dynamics of naming networks 5-64

4. Binding reference names to path names 5-67

D. Implementation considerations

1. Lost objects 5-74

2. Catalogs as repositories 5-78

3. Indirect catalog entries 5-79

4. Search rules 5-81

E. Research directions 5-82

Acknowledgements 5-85

Suggestions for further reading 5-86

References 5-89

Problems 5-89

Last page 5-94

Overview

A property of an information system that determines its ease of use and its range of applicability is the way it creates and manages the objects of computation. One aspect of object management is allocation of physical resources to implement objects. Another, equally 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 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 names and objects. A catalog is an example of a context (q.v.)
closure	- an object consisting of an object that refers to other objects by name, and a context in which those names are bound.
component	- an object that is named by another object.
context	- the abstract concept of a particular mapping 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 that several users refer to. 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 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 a unit worthy of a distinct name.

path name - a multi-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.

substitution - when a shared object containing components of its own refers to different components depending on the identity of the user of the shared object.

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

tree name - a multi-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 with 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.

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

A. Introduction

1. Names in computer systems

The topic of naming objects spans a wide range. At one end of the range is the naming of the individual variables of a program, and the rules of scope and lifetime that apply to names used within a collection of programs that are constructed as a single unit. At the other end of the range are database management systems, which provide retrieval of answers to sophisticated queries for information permanently filed by name and by other attributes. These two extremes lie on either side of our intended domain of discussion. The first extreme is generally studied under the label "semantics of programming languages" and the second extreme is studied under the label "database management".

In between, however, is a middle ground important in practice, consisting of 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 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 middle region, 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 format 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 to easily "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.

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 which computation occurs. Objects may be simply arrays of uninterpreted 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, or a

---

* Examples include the Compatible Time-Sharing System (CTSS) constructed at M.I.T. for the IBM 7090 computer, the Honeywell Information Systems Inc. Multics, IBM's TSS/360, the TENEX system developed at Bolt, Beranek and Newman for the Digital Equipment PDP-10 computer, the M.I.T. Lincoln Laboratories APEX system for the TX-2 computer, the University of California (at Berkeley) CAL system for the Control Data 6400, and the Carnegie-Mellon HYDRA system for a multiprocessor DEC 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 relations among modularity, sharing, and naming in his arguments for segmented memory systems [Dennis, 1965].

name for the component object may be included in the containing object.

In the first of these schemes, 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. 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. Under the copying scheme 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 other procedure can never see the changes.

The 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. The pattern for the use of names 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, a context is created mapping that name into that component object, 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 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 map a name into an object is called binding that name to that object in

---

* In the study of programming language semantics, the term environment is used for a closely related concept. Usually, an environment provides a mapping with the possibility of duplicate names, a stack or tree structure, and 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.

that context. Using a context to locate an object from a name is called resolving that name in that context. Figure 5-1 illustrates this pattern.

In examining figure 5-1, two further issues are apparent: 1) the context must either include or name the contained object; 2) the containing object must be associated with a context. Figure 5-2 illustrates the familiar example of a location-addressed memory system, in which electrical wiring in effect includes the contained object in its context and includes the context in the containing object.

The alternative approach for handling the first-mentioned 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 5-3 provides an example in which an interpreter's symbol table is the first, higher-level context, and the location-addressed memory of figure 5-2 provides the lower-level context. A more elaborate example could be constructed, with several levels of names and contexts, but at the end there must always be some context that directly includes its objects (as did the location-addressed hardware memory) rather than naming them in still another context. Further, since the goal of introducing names was sharing, and thus to eliminate multiple copies of objects, each object ultimately must be included in one and only one context.

Returning to figure 5-1, it is also necessary for a containing object to be associated with its context. This association is normally 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. An object that exists solely for the purpose of associating some other name-containing object with its context is known as a closure. In many cases, the closure is implicitly supplied by the name interpreter. For example, in

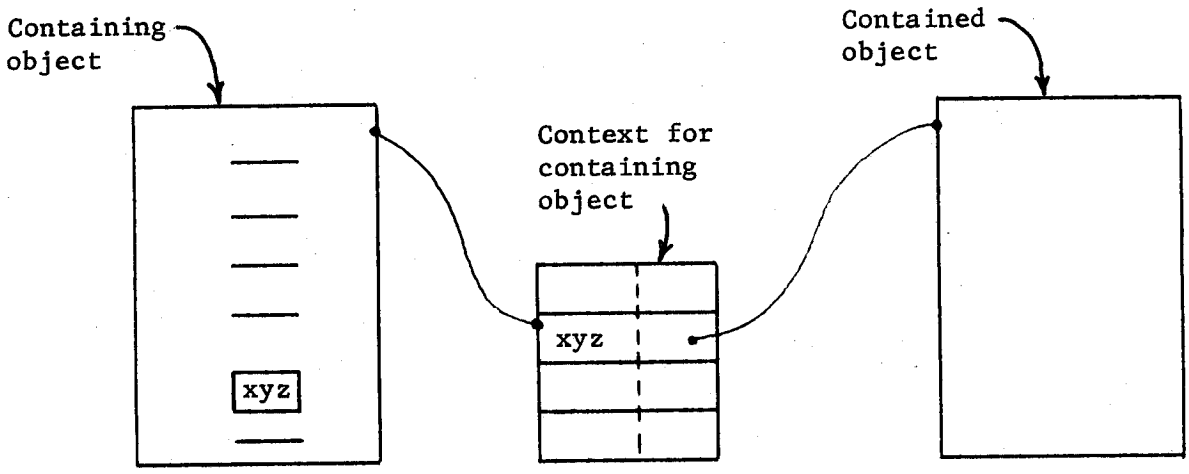


Figure 5-1 -- Pattern for use of names. The containing object includes a use of the name "xyz". The containing object is somehow associated with a context. The context contains a mapping between the name "xyz" and enough information to get to the contained object. Because the contained object has not been copied into the containing object, it is possible for some third object to also contain this object; thus sharing can occur.

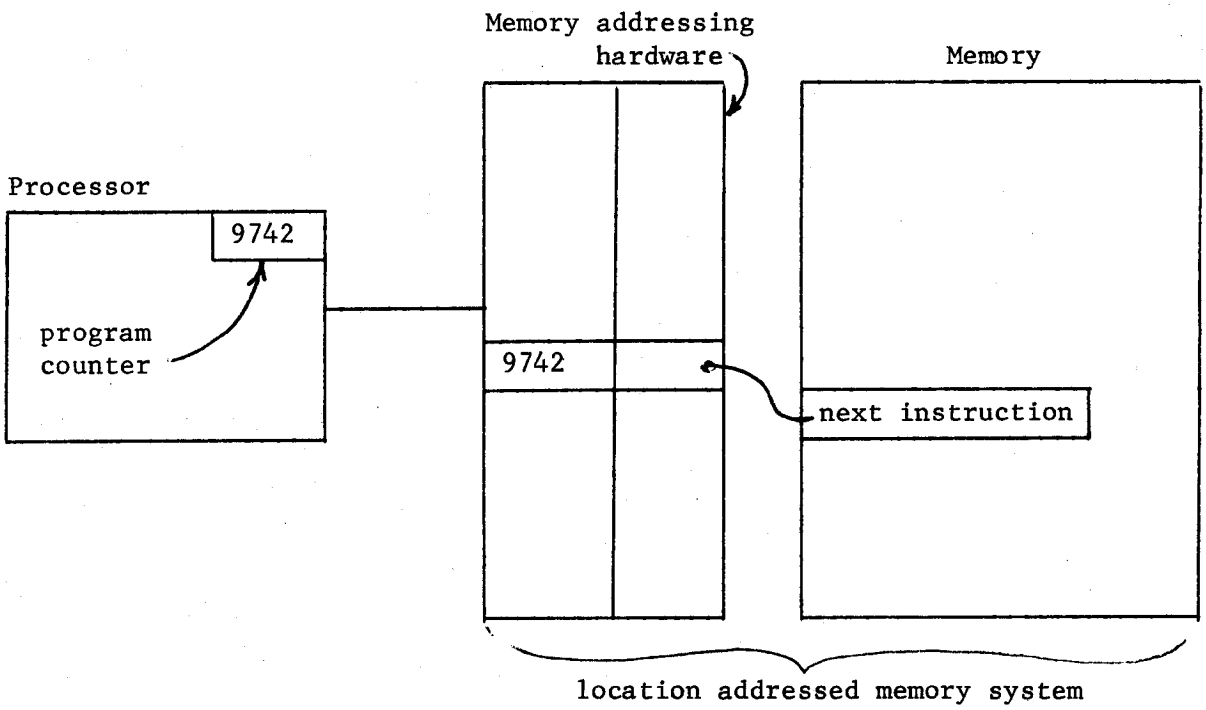


Figure 5-2 -- Instruction retrieval as an example of naming. The processor program counter names the next instruction to be interpreted. The processor is associated with a context, the memory addressing hardware, by means of an electrical cable. The context maps the name "9742" into the physical location in memory of some particular word of information, again using electrical cable to form the association.

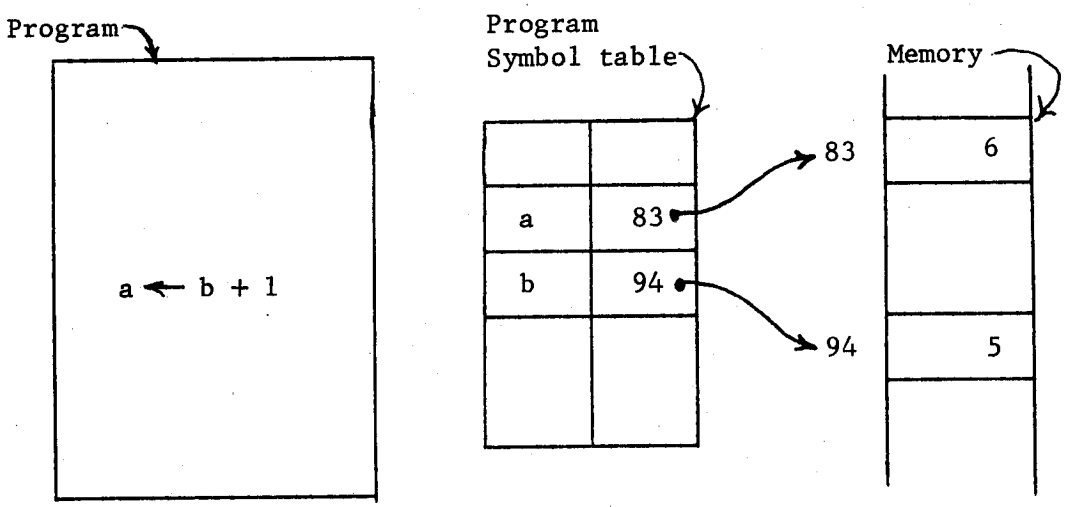


Figure 5-3 -- A two-level naming example. An interpreter executes a program containing the names "a" and "b". The interpreter resolves these names using the context represented by a symbol table that maps the names "a" and "b" into lower level names, which are addresses in the memory. These lower level names are resolved as in figure 5-2.

figure 5-3, the interpreter uses the program's symbol table 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.

A problem arises if the wrong context is automatically supplied by the 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 containing object. 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 take cognizence of the names of the objects that the shared object uses (for example, by avoiding use of those names) we have a failure of the goal of modularity. We will 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 in the form of name conflict, in which for some reason it seems necessary to bind one name to two or more objects in the same context. This situation often occurs when putting two independently conceived sets of programs together 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 and change the uses of the conflicting names.

Sharing should also be controllable, so that if the internal structure of a shared object is understood, selective substitutions can be made for some of its components. However, these substitutions should not necessarily affect other users of the shared object. For example, when a single subprogram is used in different applications, it may be appropriate for that subprogram to have a different context for each application. In this case, the different contexts would resolve the same set of names, but some of them might resolve to different objects. There are two common situations in which an object might need different contexts for different applications:

1. When the object is a procedure, and its specifications include 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.
2. When a programmer makes a change to one small part of a large subsystem, and wants to test it together with the unchanged parts of the subsystem. Copying the entire subsystem is one way to proceed, but that approach does not take advantage of sharing, and in cases where shared writable data is involved may produce the wrong result. The alternative is to locate the contexts that refer to the modified part, and create special versions that refer to the new part instead of the original.

In both of these situations some provision must be made for an object to be connected to 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 provides an association of the 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.

There are other potential problems with using names to refer to objects. For one, the bindings contained in contexts are often changeable. For example, catalogs often serve as contexts, and usually 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 insure that when the time comes to use that name and context the desired object will be accessed.

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.

2. 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]. In them, 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 subprograms is put together (an

---

* The names of individual FORTRAN variables and arrays are handled by the compiler using a mechanism distinct from the one of interest here.

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 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 not acceptable input to a further loading operation. This constrains the use of modularity, since a program cannot be contained by another program.

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.

Similar problems arise in APL as in FORTRAN: 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 writable 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 temporary variables is impossible without knowing what the other two programmers are doing. If the programmer of B names a variable "X" and declares it local to B, that may disrupt communication between

procedure A and C if the other programmers happen to use the name "X" for that purpose, since 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 conventional way of using them is similar to APL systems in many ways [Moses,1970]. Each programmer has a single context for use by all LISP functions. Functions of other programmers must be copied into the context of an employing function. Call-chain name resolution is used.

Internally, LISP has a naming mechanism that is used to eliminate naming problems within the scope of a single programmer's set of functions. The atoms, functions, and data of a single programmer are all represented as objects with unique internal names. When an object is created, it is bound to this internal name in a single (per programmer) context. (The implementation of this mechanism varies among operating systems. It usually is built on operating system main-memory addressing mechanisms and a garbage collector or compactor.) These internal names usually cannot be re-bound. Internal 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 a closure is invoked, the LISP interpreter resolves names appearing in the function by using its associated context. The objects and data having 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 system, 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

---

* 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.

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 provides catalogs of permanent objects. Names used in objects are resolved automatically using as a context one of the catalogs of the file system. The names used to indicate files are consequently called "file names". In some of these systems, it is not necessary to copy an object in order to use it; in these systems writable objects can be successfully contained by more than one object.

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 can be done 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. For the convenience of a programmer creating a collection of related objects, file names appearing in objects in the collection and indicating objects within the collection can be abbreviated by omitting the programmer's name. This convenience requires an additional sophistication of the name resolution mechanisms of the file system.

Such abbreviation schemes, although convenient, must be used with care. For example, if an abbreviated name is passed as a parameter to an object 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. (For example, Multics provided a tree-structured file naming system.) 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 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 sorts 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 will 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.

3. 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 machine use:

   a) Names intended for use by human beings should be (within limits) arbitrary-length character strings.

   b) Names intended for interpretation by hardware should be fixed length, preferably fairly short, strings of bits.

2) Local versus universal names:

   a) In a system with multiple users, every object must have a distinct, unique identity. Thus there is usually 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 requirements at once, as in figure 5-4, we can identify a possible need for both human- and machine-interpretable local and universal names--a total of four combinations. Further, since an object may be referred to by many other objects, it may actually have many local names, both human- and machine-readable. In figure 5-4, some typical names an object might have in a general-purpose computer system are illustrated. A program may include a call to a library cosine-calculating subprogram that it names "COS", a local, human-readable name. The universal, system-wide name (in this case a hierarchical tree name) might be "library.math_package.cosine". A 36-bit, system-wide unique identifier is also illustrated. Finally, a local, fixed-length name is typically a small binary integer, as illustrated in the lower right hand corner; perhaps in this context there are 25 program and data objects and the cosine subprogram was the 14th to be assigned a local name. Another context that refers to the same object would use the same universal names, but might have its own local names, different from those illustrated.

A further complication, especially on 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 to be used. Synonyms are often useful when two previously distinct contexts are combined for some reason.

---

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

	universal name	local name
human readable, arbitrary length character string	library.math_package.cosine	COS
machine readable, fixed length bit string	11100111010001000100111011111010011	01110

Figure 5-4: **Kinds of names found in computer systems**

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. Tree names are examples of names from unlimited naming contexts, as are unique identifiers, by definition. In a limited context the names themselves are a kind of 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 creating or destroying an object that is named in an unlimited context, while speaking of allocating or deallocating 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 used 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 an unlimited virtual memory, that some intermediate layer can implement an unlimited context for use of outer layers and user applications.

4. Plan of study

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 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

---

* 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.

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 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 machine-interpretable names. Although we shall see points of contact between these machine-oriented names and 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 execution 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 machine executes instructions in order not to become a severe bottleneck.

Figure 5-2 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 5-5, 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. Therefore, we redraw figure 5-5 as in figure 5-6, 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.

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

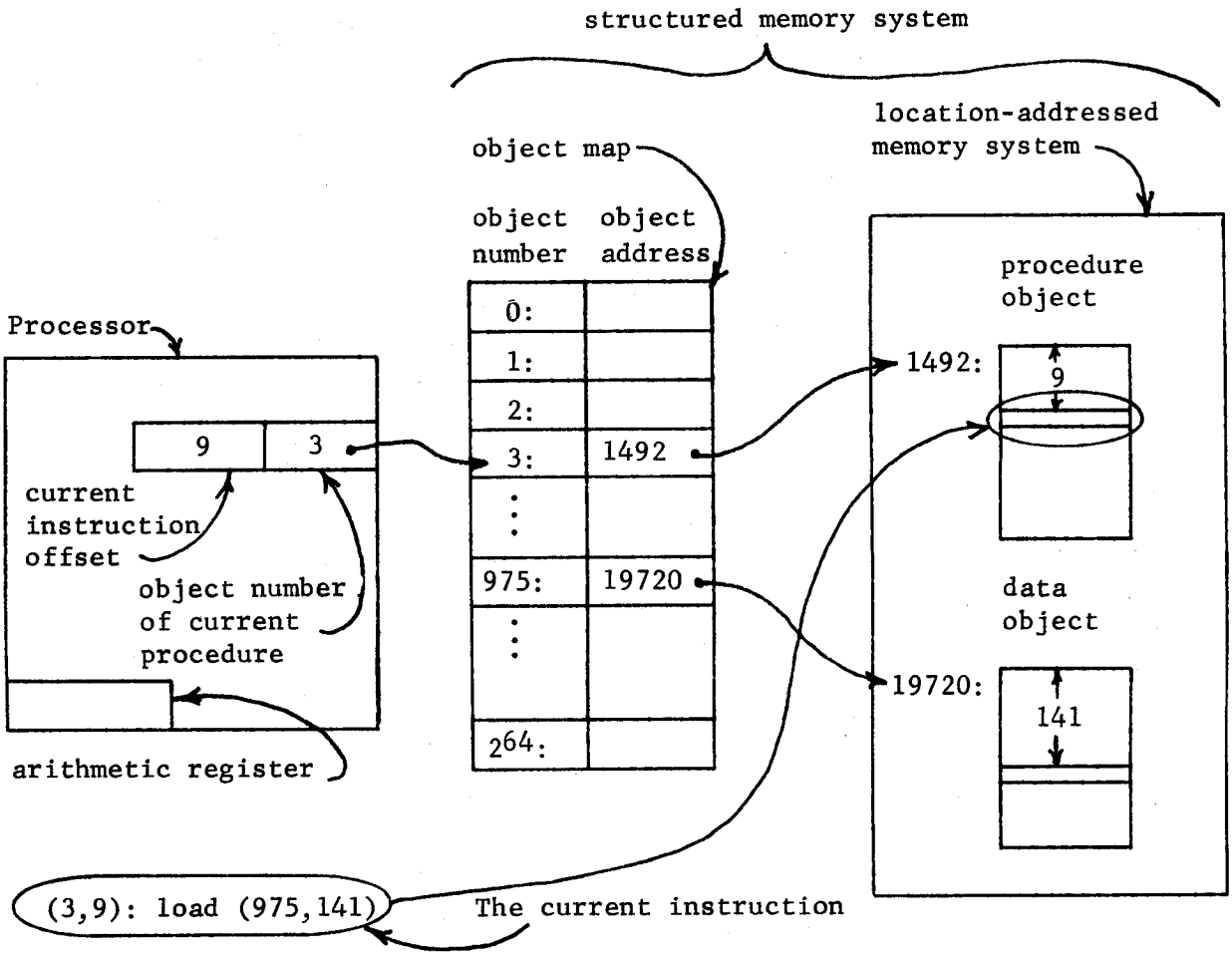


Figure 5-5 -- The structured memory system. The processor is executing instruction 9 of procedure object 3, located at address 1501 in the memory. That instruction refers to location 141 of data object 975, located at address 19861 in the memory. The columns of the object map relate the object number to the physical address. In a practical implementation, one would add a third column to the object map to hold further information about the object. For example, for a segment object, one might store the length of each segment, and include checking hardware to insure that all data offsets are of values within the size of the segment.

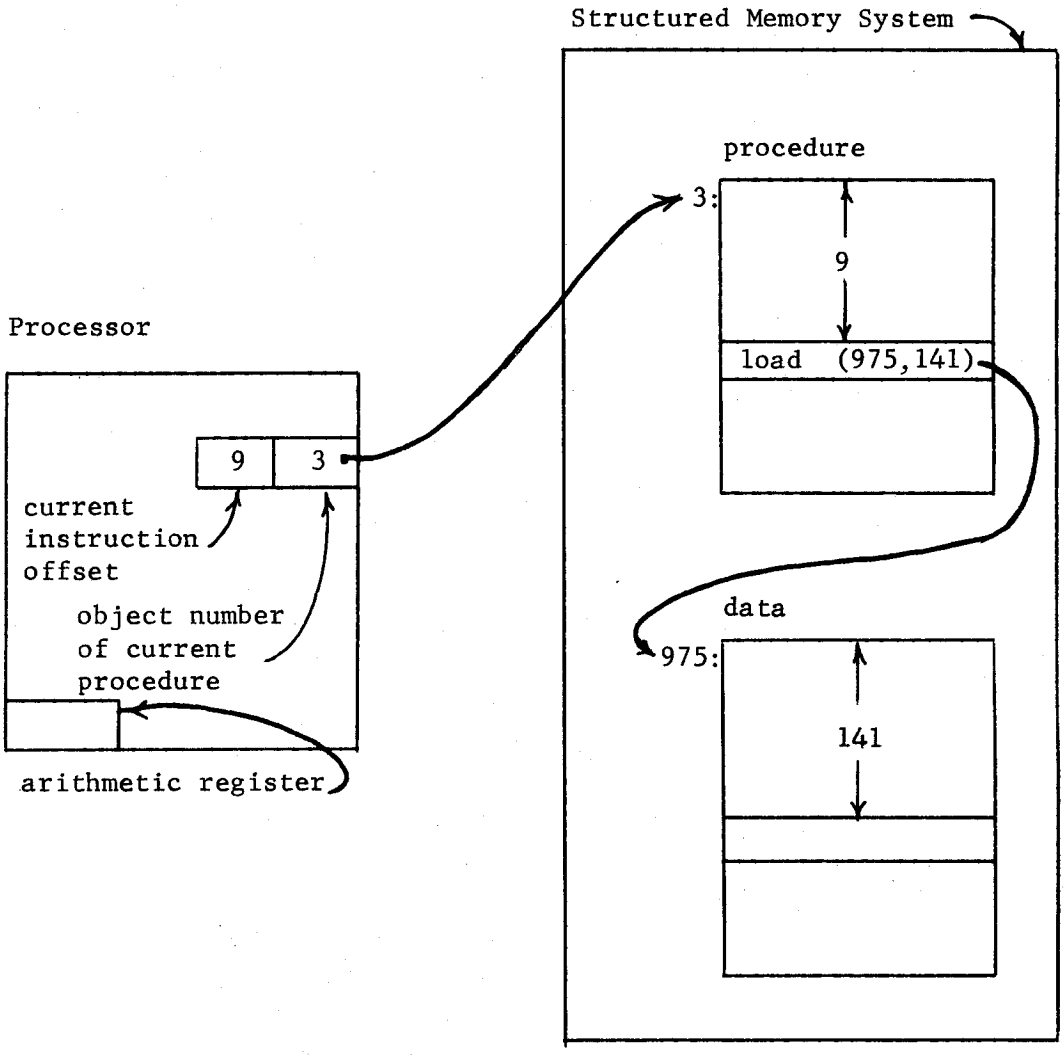


Figure 5-6 -- The structured memory system of figure 5-5 with the object map assumed and therefore not shown. Note that the procedure object contains the name of the data object, 975.

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 5-I, on page 5-.., 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.*

1. Multiple naming contexts

As out 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 5-6 should be private to the user of the program, and there are two users of the same program. One approach (the copying scheme) would be to make a copy of the procedure, which 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.

The alternative scheme, using naming rather than copying, requires that we somehow provide a naming context for the procedure that can be different for different users. The obvious approach is to make the context depend on which virtual processor is in use, and arrange separately

---

* Although unique-identifier object maps have been proposed [Radin and Schneider, 1976; Redell, 1974] there are formidable implementation problems, 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.

that each user or application operate using a separate virtual processor. This approach leads to figure 5-7, in which two processors are shown, and each processor has been outfitted with an array of pointer registers, each of which can hold one object number. The pointer registers are numbered, and the part of the address of each instruction that used to contain an object number is now interpreted as a register number instead.

Thus, in figure 5-7, 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 "2" is resolved in the context of processor number one, object number 975 is found, which is taken to be the name of the desired object in the next lower (in this case, the universal) context. Thus when processor one evaluates the address of the instruction (3, 9) it will obtain the 141^{st} item of object 975. Similarly, when processor two evaluates the same address, it will obtain the 141^{st} item of object 991.

We have thus arranged that a procedure can be shared without the onerous requirement that everything the procedure refers to must also be shared--we are permitting selective substitution of subcomponents of procedure objects. (See table 5-I).

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,

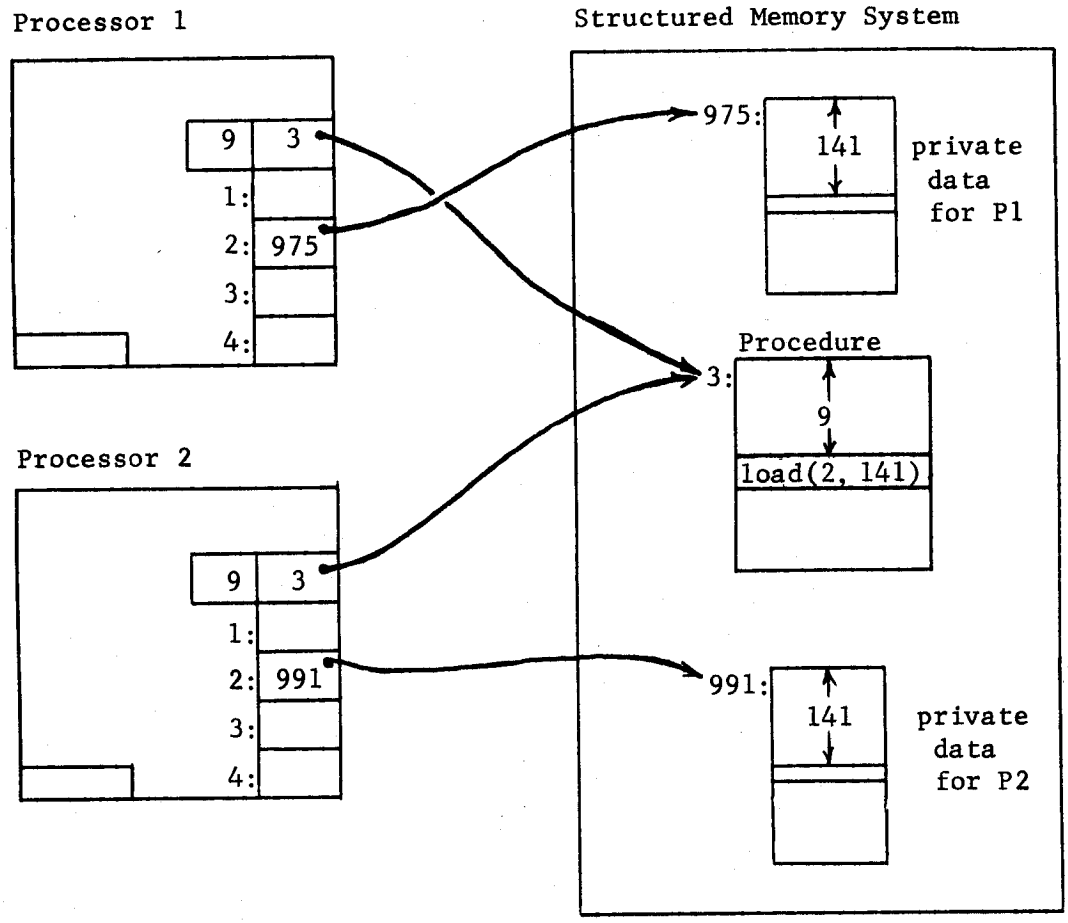


Figure 5-7 -- Addition of pointer registers to the processor, to permit a single procedure to have a processor-dependent naming context.

so in accordance with the standard pattern for using names, the compiler should also provide for binding of that name to the correct object. In this case, it might do so as in figure 5-8, 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 a 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 its object number with the context binding information as in figure 5-9. The result would be the pattern of reference shown in figure 5-10.

Translation from the high level context description of figure 5-9 to the register context of figure 5-10 is accomplished by a program known here as a context initializer* and most such programs permit a wider variety of object interlinking possibilities than illustrated in figure 5-9. Before getting into that subject, we should first consider three elaborations on the naming conventions already described.

---

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

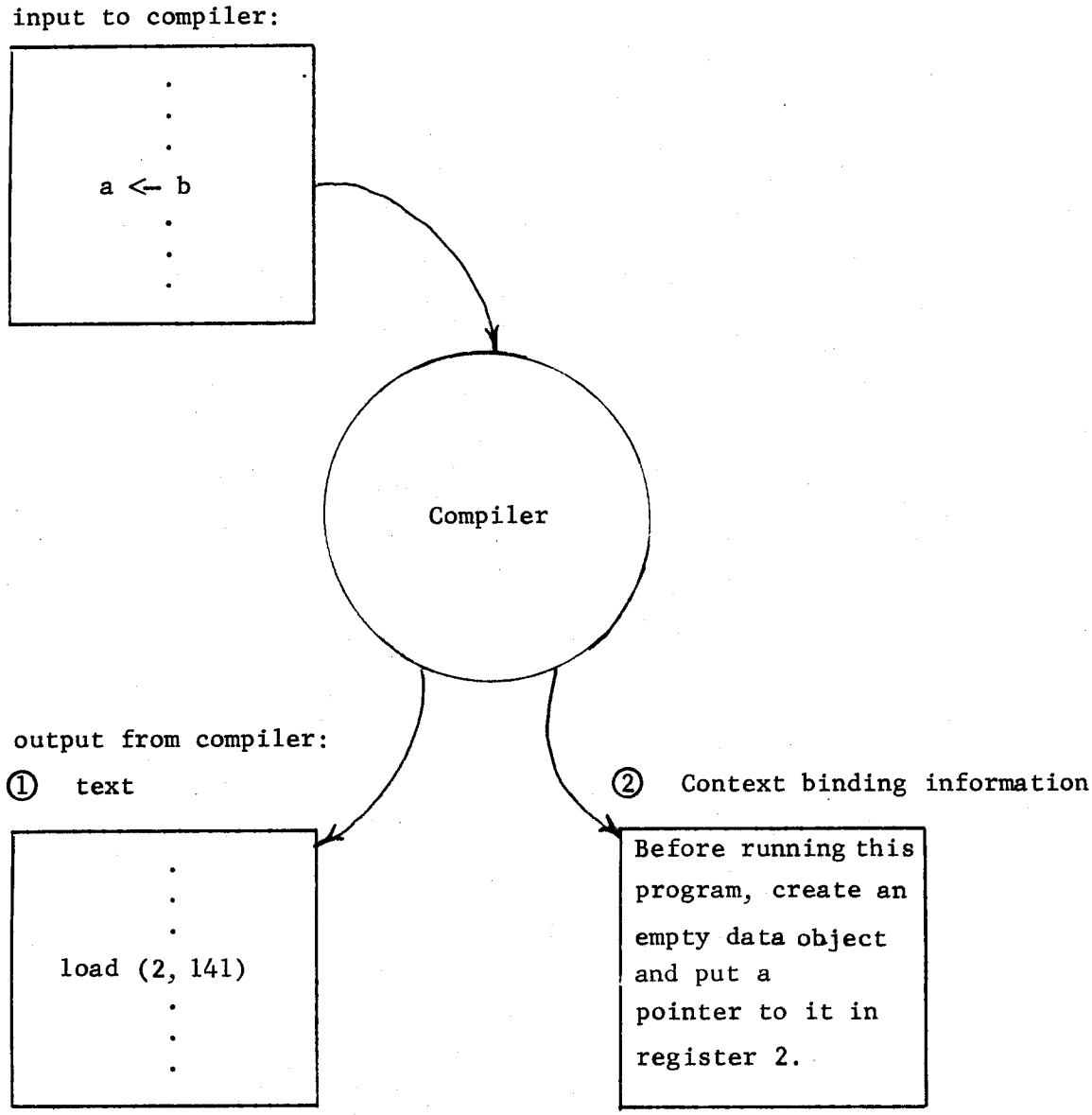


Figure 5-8 -- When a per-processor addressing context is used, one of the outputs of the compiler is information about the bindings needed to create that context.

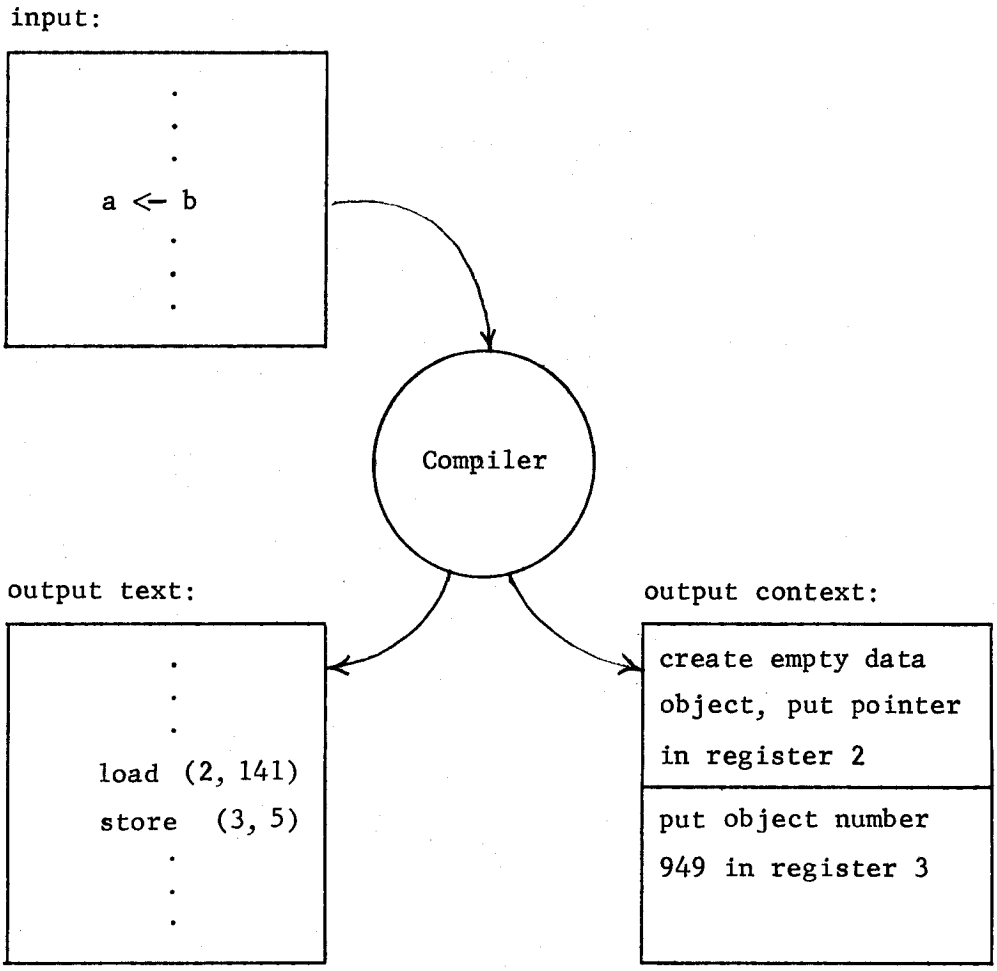


Figure 5-9 -- Shared data objects can be handled by appropriate entries in the context part of the compiler's output. This output context produces the reference pattern of figure 5-10.

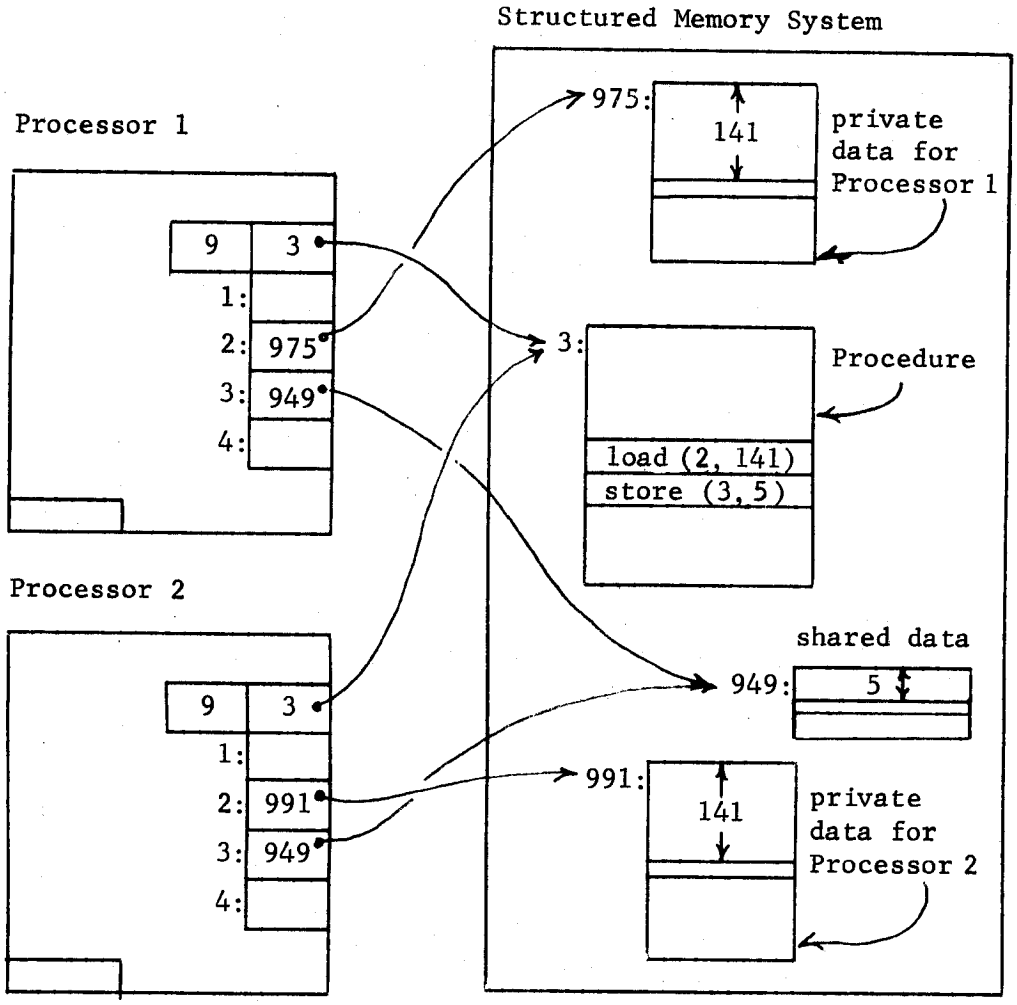


Figure 5-10 -- A shared procedure, using both per-processor private data and shared data, with a context in the processor registers and bindings supplied by the compiler of figure 5-9.

2. Larger contexts and context switching

The first of these elaborations is to prepare for the possibility that some procedure may need to refer to more objects than there are available processor registers. (Recall that a limited context is a common naming problem.) This possibility can be handled by moving the processor context into memory, in a data object, and leaving behind a single processor register, the current context pointer, that points to this context object. Figure 5-11 illustrates this architecture, figure 5-12 shows the corresponding changes needed in the context-establishing information that the compiler must supply, and Table 5-I continues to chart our progress.*

The establishment of the context for resolving names of the procedure is going on at three different times:

1) Partly at 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) Partly 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) Finally, 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 virtual processor. In that case, on second and later uses of the program, only the third step may be required.

---

* 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 output [Daley and Dennis, 1968]. In the IBM System/360, the control section (CSECT) acted as a context object that was initialized by the link-editor by reference to prototype control sections created by the compilers [Hedberg, 1963].

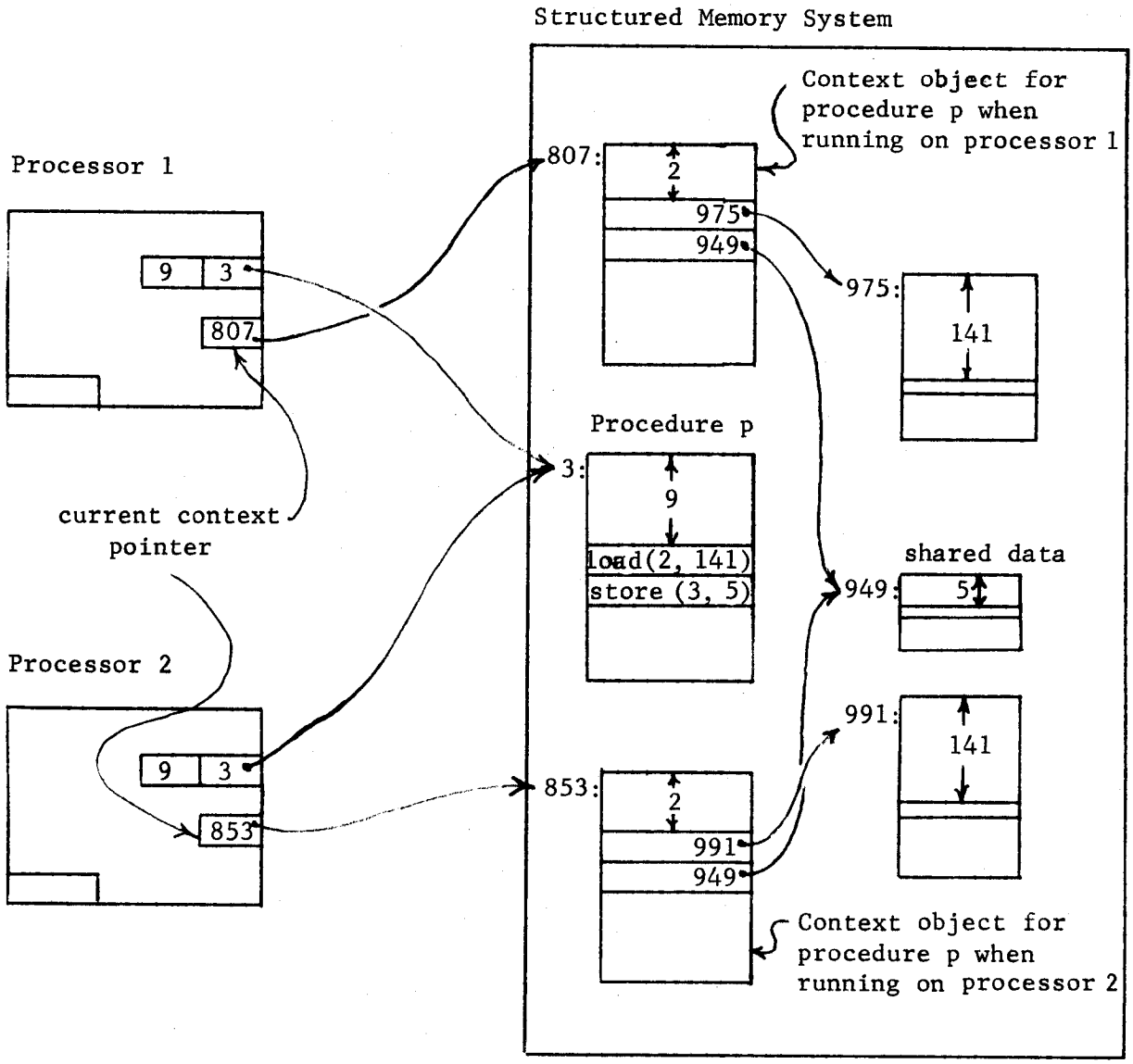


Figure 5-11 -- By placing the per-processor context in a data object in memory, and adding a current context pointer register to the processor, the context is not limited to the number of processor registers. Instead, all addresses are assumed to be interpreted indirectly relative to the segment named 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.

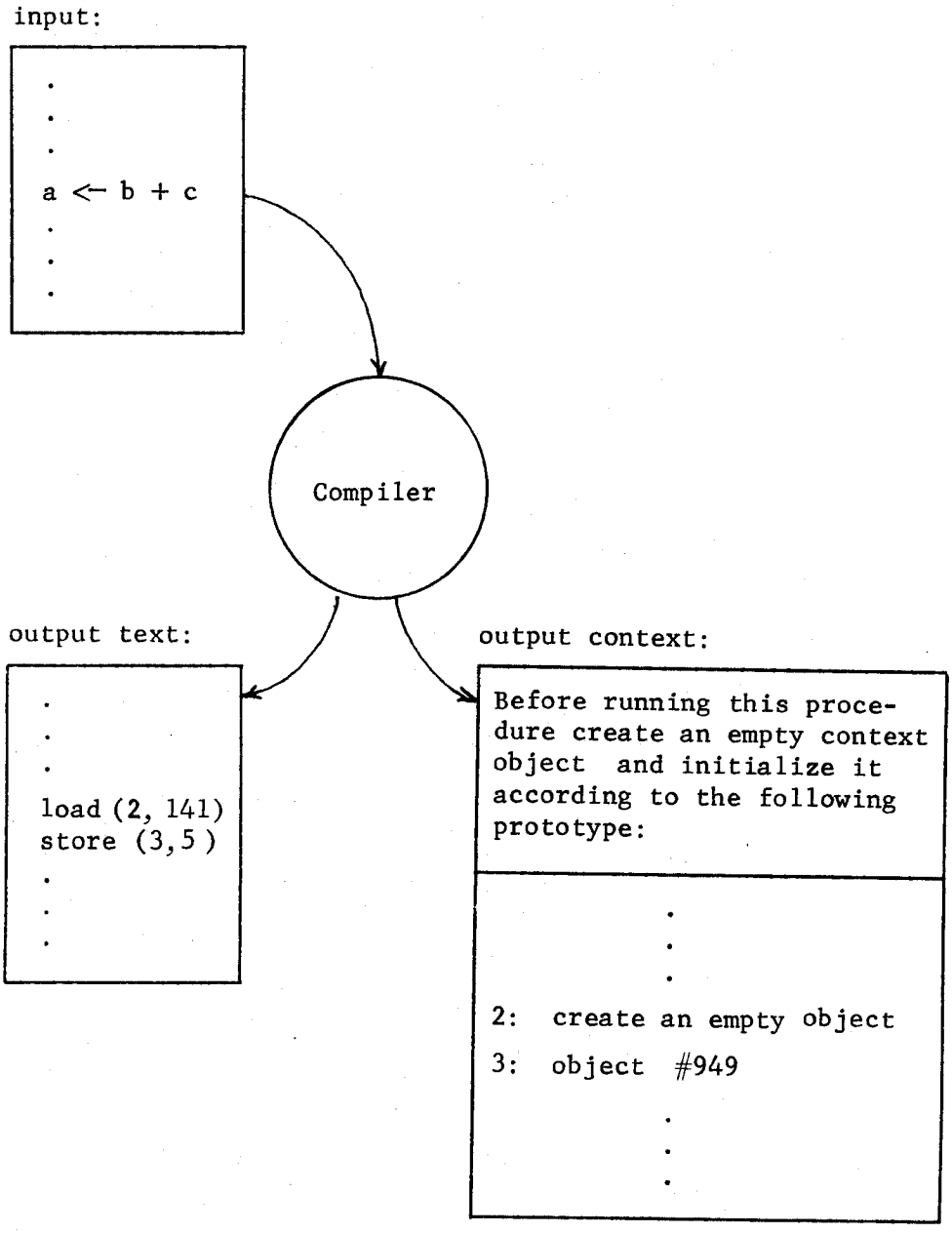


Figure 5-12 -- Compiler output needed to initialize the context objects of figure 5-11. In addition to the instructions provided by the compiler, one further step is needed: just before calling procedure "p", the object number of its context object for this processor must be loaded into the current context pointer register.

The second elaboration of our per-procedure context scheme is to provide for 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, 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 it should change back. 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. Figure 5-13 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 5-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.

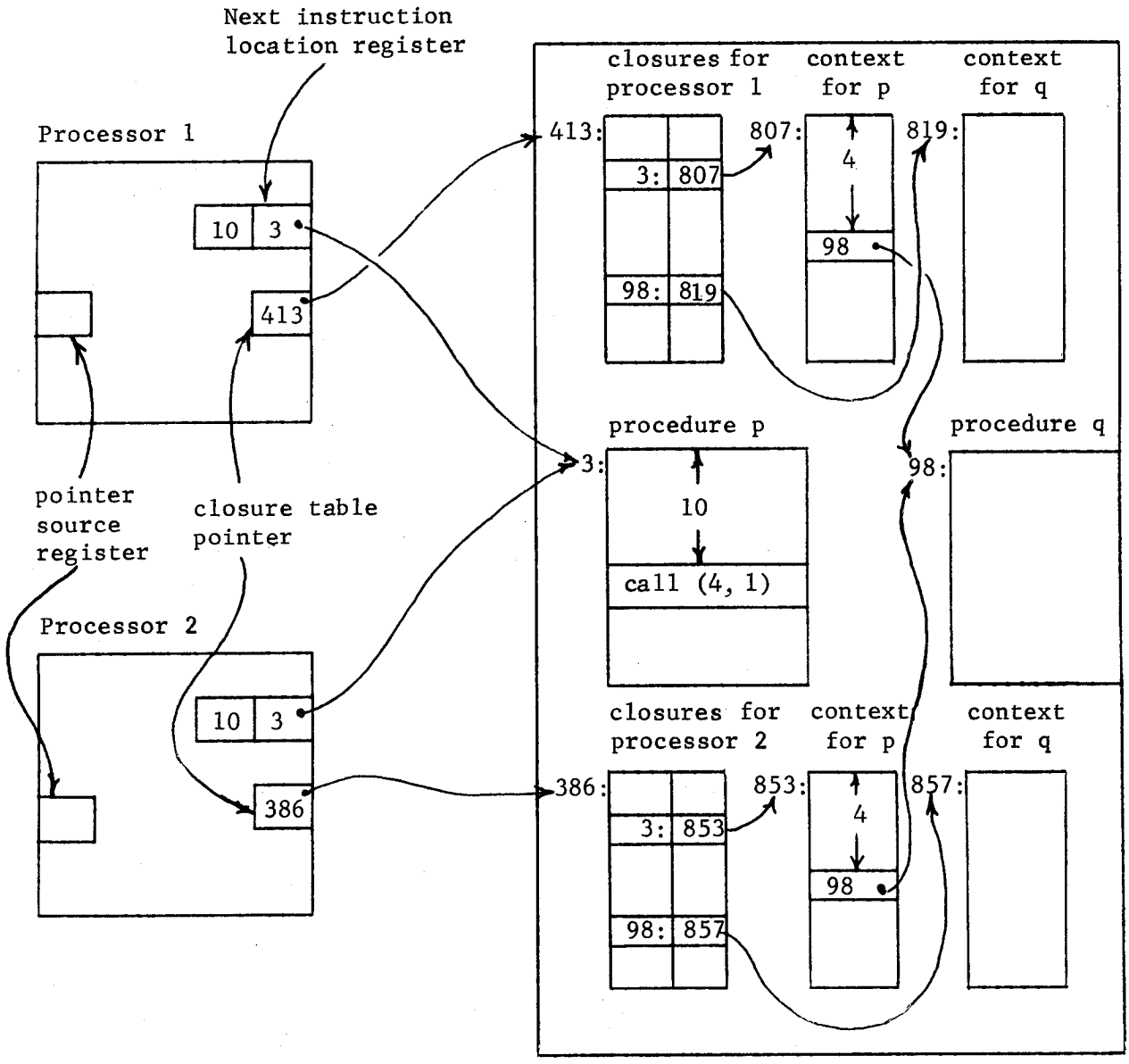


Figure 5-13 -- Context switching. Procedure "p" is just about to call procedure "q", in both processors. The current context for resolving names is located in two steps, starting with the closure table pointer and the object number of the current procedure. This pair (for processor 1, the pair is (413,3)) leads to a location containing the context for "p" (for processor 1, p's context 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 pointer 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 per-processor context. This possibility would be required if it is desired to share some data object without sharing all of its component objects. Such selective substitution 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 5-13 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, in the style of an indirect address. 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 address. Presumably, location 7 of the data object contains, instead of an arithmetic item, an outbound reference (say (4,18)) which 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 must explicitly put in the processor a pointer source register which the processor always automatically loads with the object number of the object from which

it obtained the pointer it is currently following. To obtain the correct context, the processor always uses the current value of the pointer source register as the index into the closure table. Since it obtains most pointers from a procedure, the pointer source register will usually contain the object number of the current procedure. However, whenever an indirect address chain is being followed, the pointer source register follows along, assuring that for each indirect address 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 insure 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 may now make one final observation to draw all of this machinery together: we should consider a virtual processor itself to be an object that both contains procedures as objects and also utters names that it finds while interpreting procedures. The closure table pointer is the name of the context in which names uttered by the virtual processor are resolved, albeit after several indirect steps.

---

* 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].

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 pointer source register are accessed to form a closure address.
2) The closure table is accessed 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 accessed 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 accessed.

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 to the location-addressed memory system, if both an object table and also block allocation (paging) are used. Thus it appears that we may be requiring a nine to one expansion of the rate of memory accesses over that required for a single data reference. The 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 cache memory for frequently used data items. Since the current context object is referenced once for

every instruction that has an operand address, its object number would almost certainly remain in even the smallest cache 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.

In these two sections we have developed a rather elaborate collection of machinery, and we should pause for a moment to try to place this machinery in perspective. Table 5-I exhibits compactly the relation among the naming objectives and the successive layers of machinery.

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

Figure 5-13 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 process creator, or the process creator may supply only one entry in the closure table, and one minimally completed context for the context initializer procedure, 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 is known as binding on demand*, and 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 term dynamic linking was used in Multics, one of the few systems that actually implemented this idea [Daley and Dennis, 1968].

Table 5-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 pointer 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 limited substitution of subcomponents	no	no	yes	yes	yes	yes
sharing procedure components with unlimited substitution of subcomponents	no	no	no	yes	yes	yes
automatic change of context on procedure calls	no	no	no	no	yes	yes
sharing data objects with substitution of subcomponents	no	no	no	no	no	yes

the processor temporarily suspends its normal sequence, saves its registers, 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 registers 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, 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, during the construction of a large program as a collection of subprograms, it can be very useful if one programmer can 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.

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 5-12, the compiler creates a prototype context object for use by the context initializer. If the context initializer were prepared for it, this 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 into the context being initialized. There are several situations in which it might be advantageous to 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 will arise 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 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 objects with which he works: source programs, compiled procedures, data files, messages, and so on. This higher-level context, usually called a file system, is designed primarily for the convenience of people, rather than programs. Among the 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 from the human-oriented file system as possible. 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 5-II, all of the examples of naming and name binding that occur in our addressing architecture alone. This table emphasizes two points:

1. In even a simple naming system there are many examples of naming and name binding.
2. 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 context 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 contexts).

C. Higher-Level Naming Contexts, or File Systems

1. Direct-Access and Read-Write Organizations

Higher-level naming contexts, or file systems, are provided in computer systems primarily for the convenience of the human users of the system. 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.

Table 5-II: Examples of naming and binding in the model addressing architecture.

	object that contains the name	form of the name	connection to context	context
1.	location-addressed memory system	none	none	none (objects are directly included)
2.	multilevel memory management logic	absolute physical address	cable	some location addressed memory system
3.	memory allocator	absolute virtual address	implicit	virtual memory address space provided by MLM
4.	object map	absolute virtual address of object	implicit	space of allocated object addresses
5.	virtual processor retrieving instructions	procedure object number	implicit	object map
		instruction location counter	procedure object number register	procedure object
6.	procedure context object	object number	implicit	object map
7.	procedure object	offset within context object	via closure table	procedure context object
8.	closure table	object number of context object	implicit	object map
9.	processor retrieving operand, step 1	procedure object number	closure pointer	closure table
10.	processor retrieving operand, step 2	offset within context	dynamically constructed in step 1	procedure context
11.	symbolic name in prototype context	symbolic object name	supplied by context initializer	file system
12.	symbolic name in source program	name resolvable at compile time	supplied by compiler	file system
		name not resolvable at compile time	containment in procedure text	prototype context

The foremost property of a file system is that it accepts names that are chosen and interpreted by human beings--typically 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. A catalog in a file system is used in one of two ways, leading to two generic kinds of file systems: a read-write file system, or a direct access file system.

In a read-write file system, the catalog manager conceals the existence of the lower level naming context used by the object identifiers in its catalog. A user may call upon the catalog manager to create an object with a given file system name. When the user wishes to read or write data from or to the object, he again calls the catalog manager, at a read or write entry point, giving the character-string name of the object. Normally, the name-resolution mechanism of the file system is sufficiently cumbersome that it is not economically feasible to use it for access to single words. Instead, usually one requests a large volume of information at once, and provides the address of a data buffer in some high performance memory with its own addressing architecture. The catalog manager looks up the character-string name, finds

---

* 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.

the object identifier, and then performs the read or write operation by copying the data from its permanent storage area to the high performance memory system. Thus, in a read-write file system, use of an object by name is coupled with the kind of data movement usually associated with multilevel memory management. Figure 5-14 illustrates.

In a direct access file system, the catalog manager does not attempt to conceal the existence of the lower level naming context. Instead of performing read and write operations for its caller, it provides an entry that we may name "get_identifier", which returns to its caller the object identifier for a given character-string name. The caller then is expected to do reading and writing to and from the object by direct use of the lower level addressing architecture. Figure 5-15 shows a direct access file system.

Which of these two kinds of designs is preferred depends partly on the arrangement of the available addressing architecture. If a structured memory system that provides multilevel memory management 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 read-write 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.

---

* Most read-write 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 cannot be considered direct access because the temporary identifiers are not usable in the context that contains the original files.

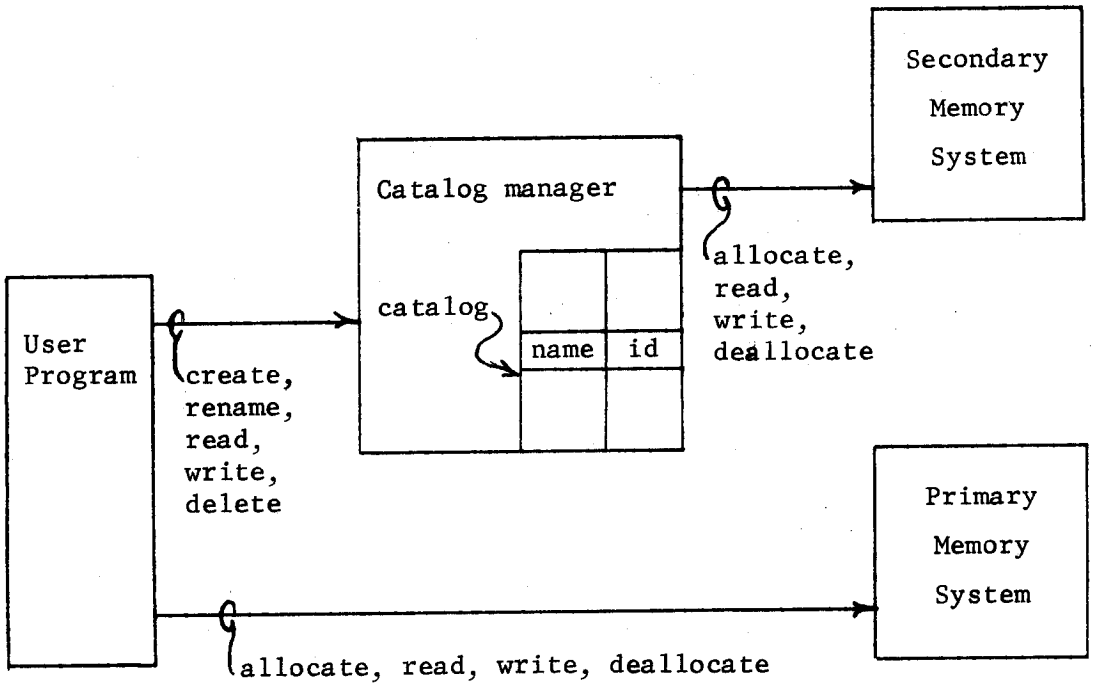


Figure 5-14 -- Organization of a simple read-write file system. The user calls on the catalog manager to create objects and record character-string names for them. The identifiers (labelled id) stored by the catalog manager are names in the context of a secondary memory system, not directly accessible 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.

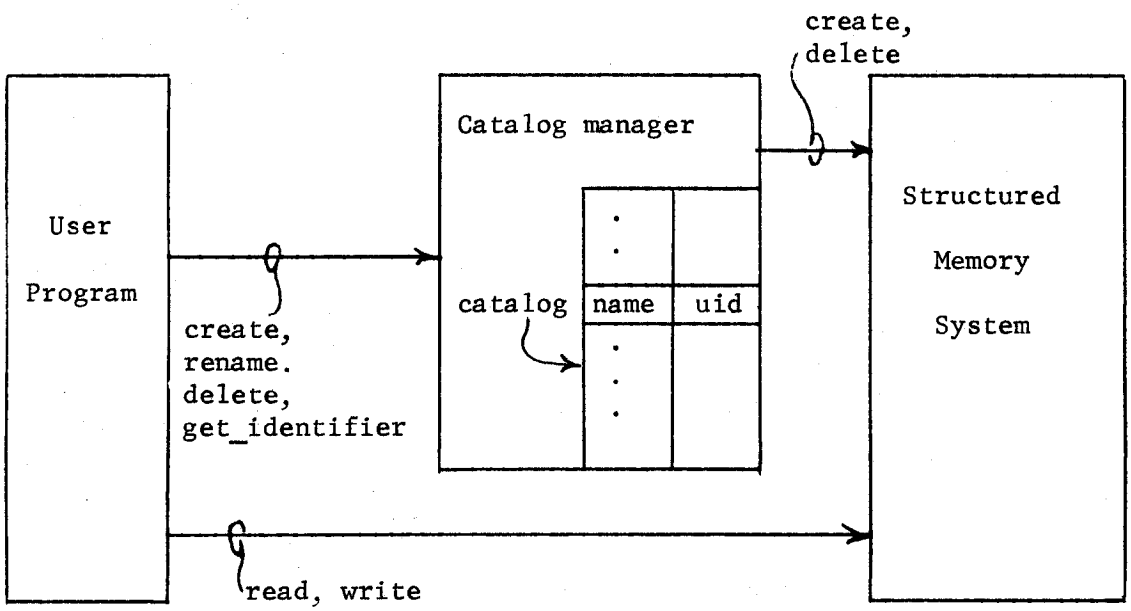


Figure 5-15 -- Organization of a simple direct-access file system. The user calls on the catalog manager, as before, to create objects and to record character-string names. However, the entry get_identifier returns the object number, or unique identifier (labelled uid) by which an object is known in 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.

The direct-access file system is sometimes called a one-level store, 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 used to simulate a read-write 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 read-write file system that consists of 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 read-write 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 read-write 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 examining the operation of the context initializer in the environments of read-write 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 read-write environment is self-evident.

2. Multiple catalogs and naming networks

The single-catalog system of figures 5-14 and 5-15 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***.

---

* 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.

** 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.

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. 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 resolves all object names presented to it by first obtaining the current user name from the virtual processor name register, and looking it up in some catalog of catalogs, called a master user catalog. The user's name is therein bound to that user's personal catalog, which the interpreter then uses as the context for resolving the 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 utter 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 uttered by his programs.

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 conveniently express a name for any object that he is permitted to

---

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

use. 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 register to the user's virtual processor: a user-settable working catalog register; and to have the file system resolve names starting from that register rather than the principal identifier register. The working catalog register would normally contain a name that is bound, for example by 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

---

* 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.

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--all names uttered by 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 call to program 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 call to B. Here we have a potential conflict between the intention of the programmer in embedding the name "B" in the program and the intention of the current user of the program, who may have adjusted the working catalog to assure correct resolution of some other name just typed at the terminal. 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 arrive at the file system from 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.)

---

* The lack of ability to name other user's 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 the user's catalog, the library, or a catalog held in common among a designated group of users [Crisman, 1965].

3) Having once changed the context in which names are resolved, the human user 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 attention 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 explicitly state, as part of each name, the name of the appropriate context. We assume, as before, that catalogs are to have human-readable character-string names, and therefore there must be some context in which catalog names can be resolved. Figure 5-16 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 say 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

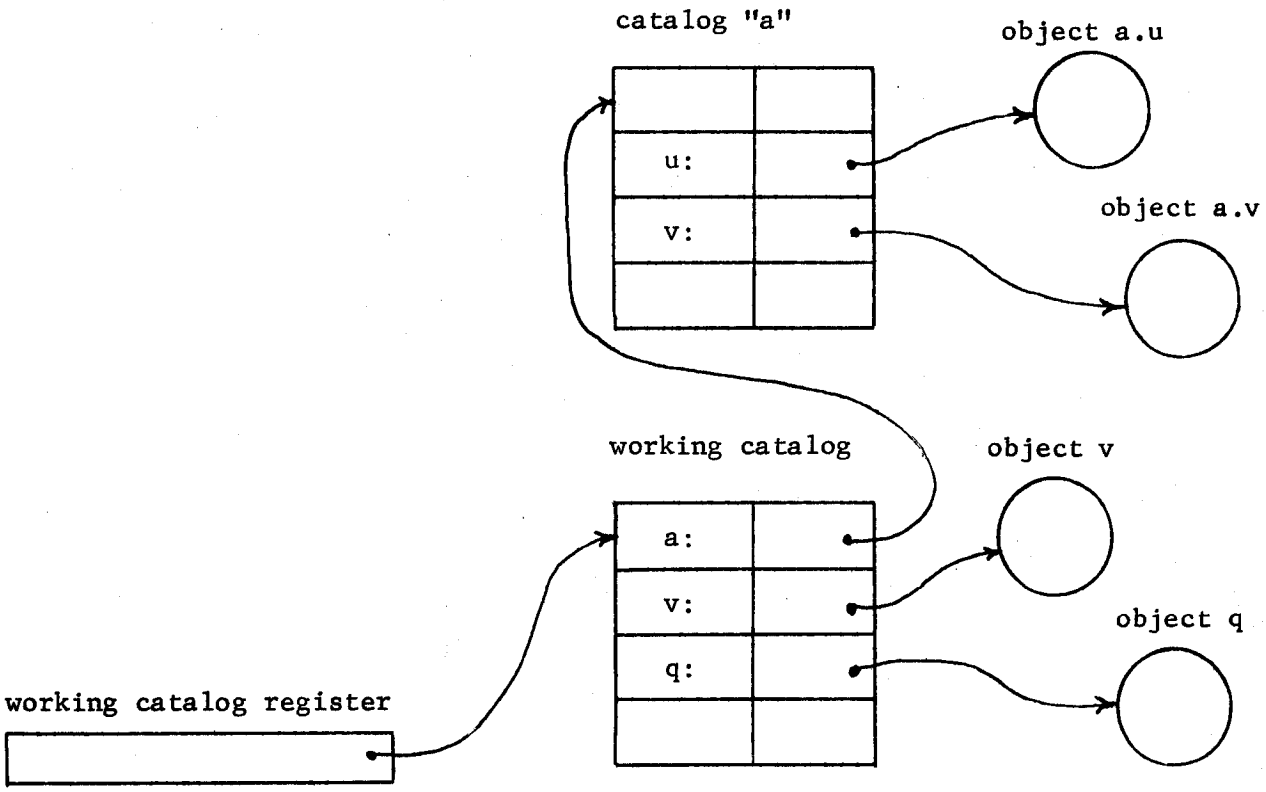


Figure 5-16 -- A simple naming network. All single-component names given to the file system are resolved in the context named by the working catalog register. All two-component names are resolved by resolving the first component in the working catalog, which leads to another catalog in which the second component may be resolved. (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.)

periods to come to programs as input arguments, originating perhaps from the keyboard; they represent a way for the user to precisely express intent 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 will return to the topic of binding names of the program to names of the catalog structure after first exploring naming networks in some depth.

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 5-16, 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 5-17, 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

---

* 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.

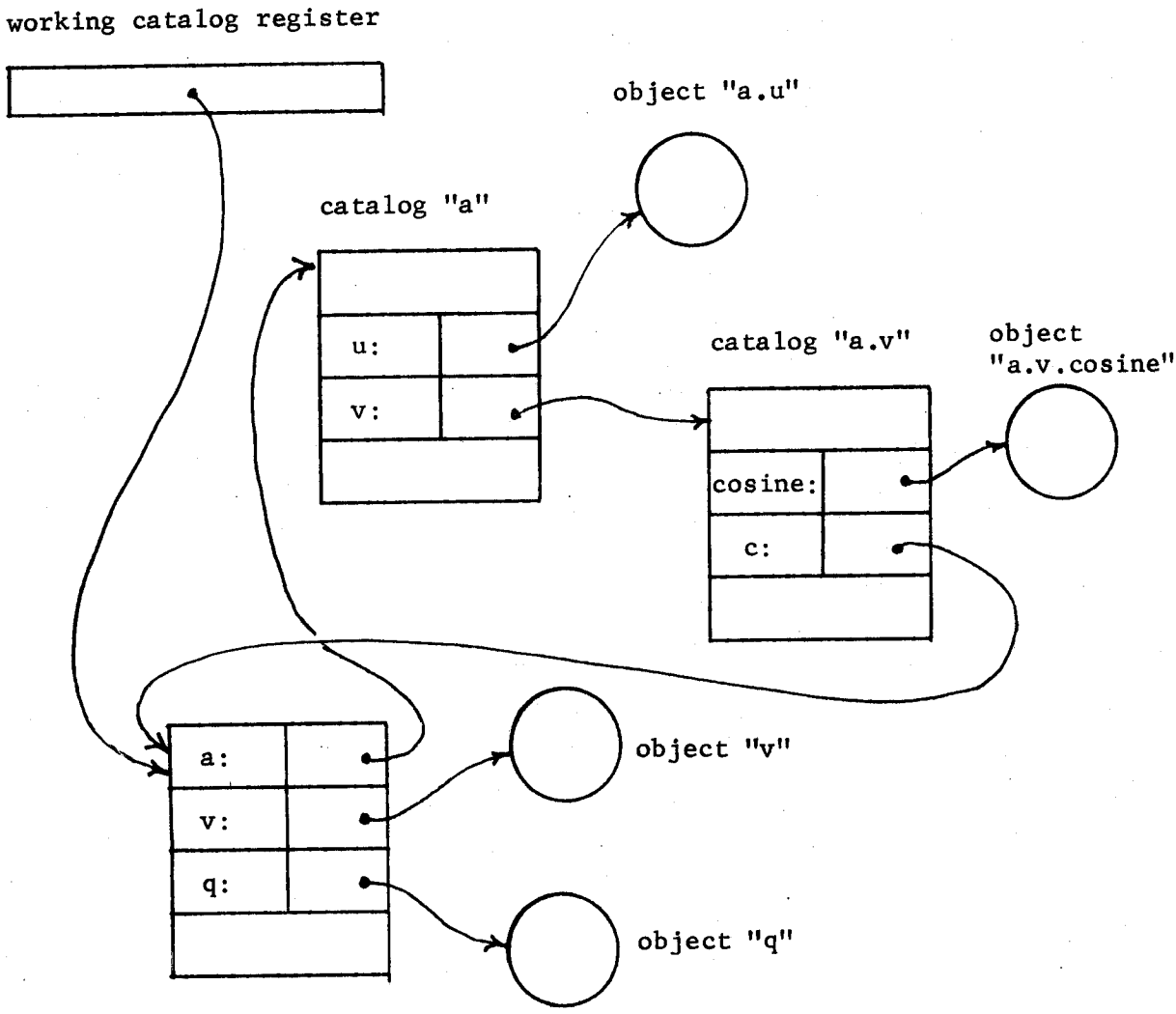


Figure 5-17 -- A naming network with recursive structure.

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

3. The dynamics of naming networks

If we were to implement the file systems of figure 5-14 or 5-15, with the intent of having a naming network, we would discover an important defect: although we might successfully implement programs to "read", "write", or "get_identifier" that understood pathnames, we could not write a program to create a naming network for those programs to work on. Unless our applications were sufficiently static that we could construct a static naming network in advance with all cross-references among catalogs that might ever be needed, we should make provisions for programs to add cross-references by calling the file system. These provisions must number three:

1) a provision to create new objects, including new catalogs.

2) a provision to add to a catalog an entry that represents a binding to a previously existing object.

3) some way of naming previously existing objects, so that provision two can be accomplished.

The first two provisions seem straightforward enough, but the third one appears to cause some trouble. If there were a way of naming a previously existing object, then why are we trying to dynamically add a cross-reference to it? The prospective cross-reference was intended to be the way by which we name the previously existing object.

---

* Naming networks are not often encountered in operating systems. The CAL time-sharing system [Lampson and Sturgis, 1976] provided one example. In data base management systems, the CODASYL standard data base system defined by their Data Base Tech Group (DBTG) called for a recursive naming network [CODASYL, 1971].

This apparently recursive dilemma suggests that there is a fundamental limitation 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 5-18, 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 5-18."

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' working catalogs (named "users".)

---

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

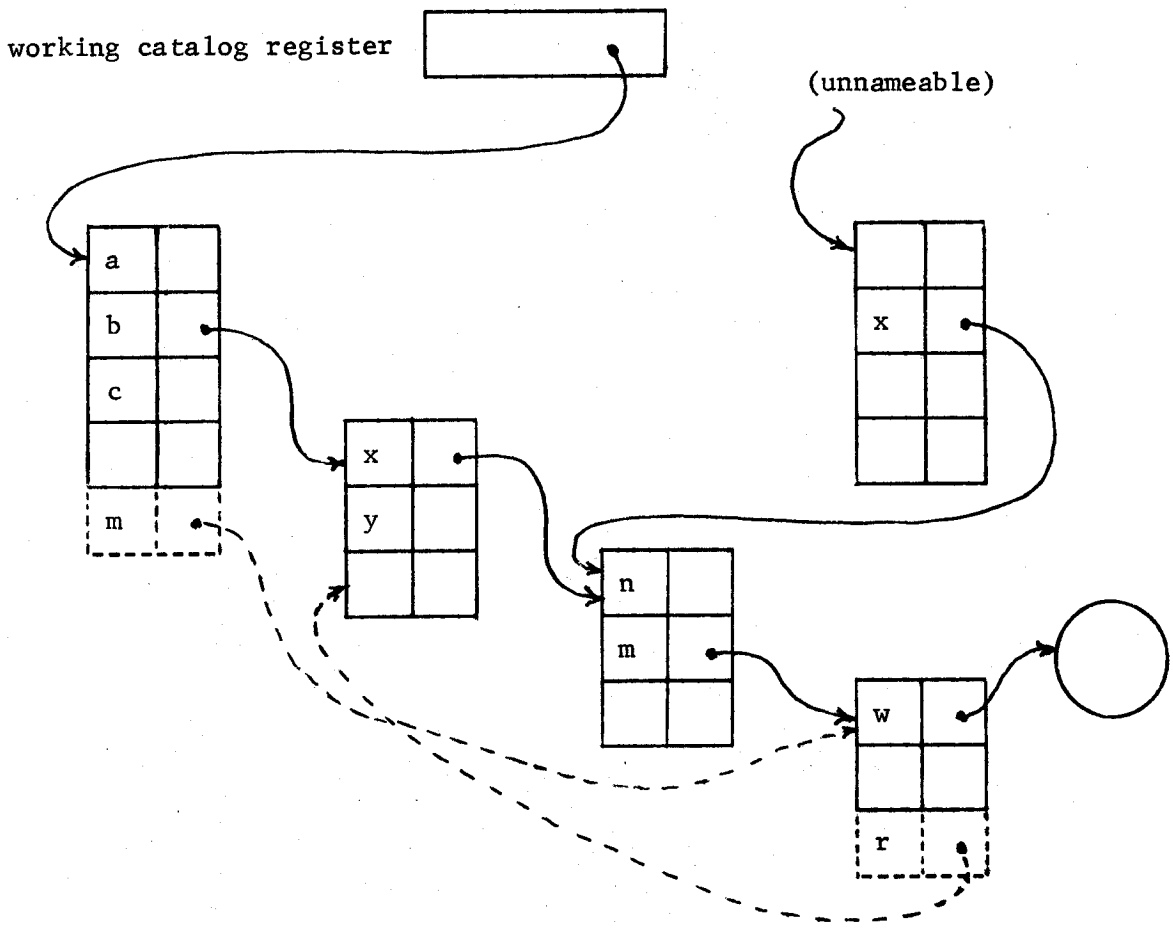


Figure 5-18 -- A naming network containing an unnameable catalog.

2) For each user of the time-sharing system, create a working catalog, arrange that whenever that user logs in, the working catalog register be loaded by the login procedure to contain the path name (relative to the "root" catalog) of that user's working catalog, and place in the user's working 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 destroyed 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 else dynamically as the program executes (in order to avoid modifying the program when it is used in a different application,) and 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 from interactive terminals. 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. We again observe that these symbolic names are of two origins: some are supplied by the <u>writer</u> 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 interactively supplying instructions to the program. 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 invoked 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 outbound 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 allow the name interpreter to know 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 embedded in the program definition 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 created, 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 is to create and 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.

The simplest approach to providing a closure is to require that the catalog that contains the object also contain entries for every symbolic name used in that object. Then, we may simply inform the context initializer that whenever any object is discovered (dynamically) 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.**

---

* 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.

** A possible way of finding that catalog is for the context initializer, when it creates a new context for a never-before-used object, to place in that context an entry containing the object number of the catalog in which the object was found.

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 dynamic context initialization (binding on demand) to be needed in the first place.

Finally, we must be careful of one more point: the achievement of modular sharing. Consider the catalog of figure 5-19 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 places in his own catalog a direct reference, named "b", to that object, the context initializer will use Smith's catalog as the context in which to resolve all sumbolic names found in procedure b, which would be an error, since Lenox's procedure "b" will get the wrong "a". (Remember that we instructed the closure initializer to use as a context the catalog in which it found the object making the reference.) The problem is that Smith should not bind the name "b" directly to the procedure in Lenox's catalog, but rather to a closure, consisting of a pointer to Lenox's procedure and a pointer to Lenox's catalog. This need arises for every object that uses names, so we conclude that we should instead arrange things as in figure 5-20, with each named procedure replaced by a closure 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 procedure "a" calls on procedure "b", the name "b" will be resolved

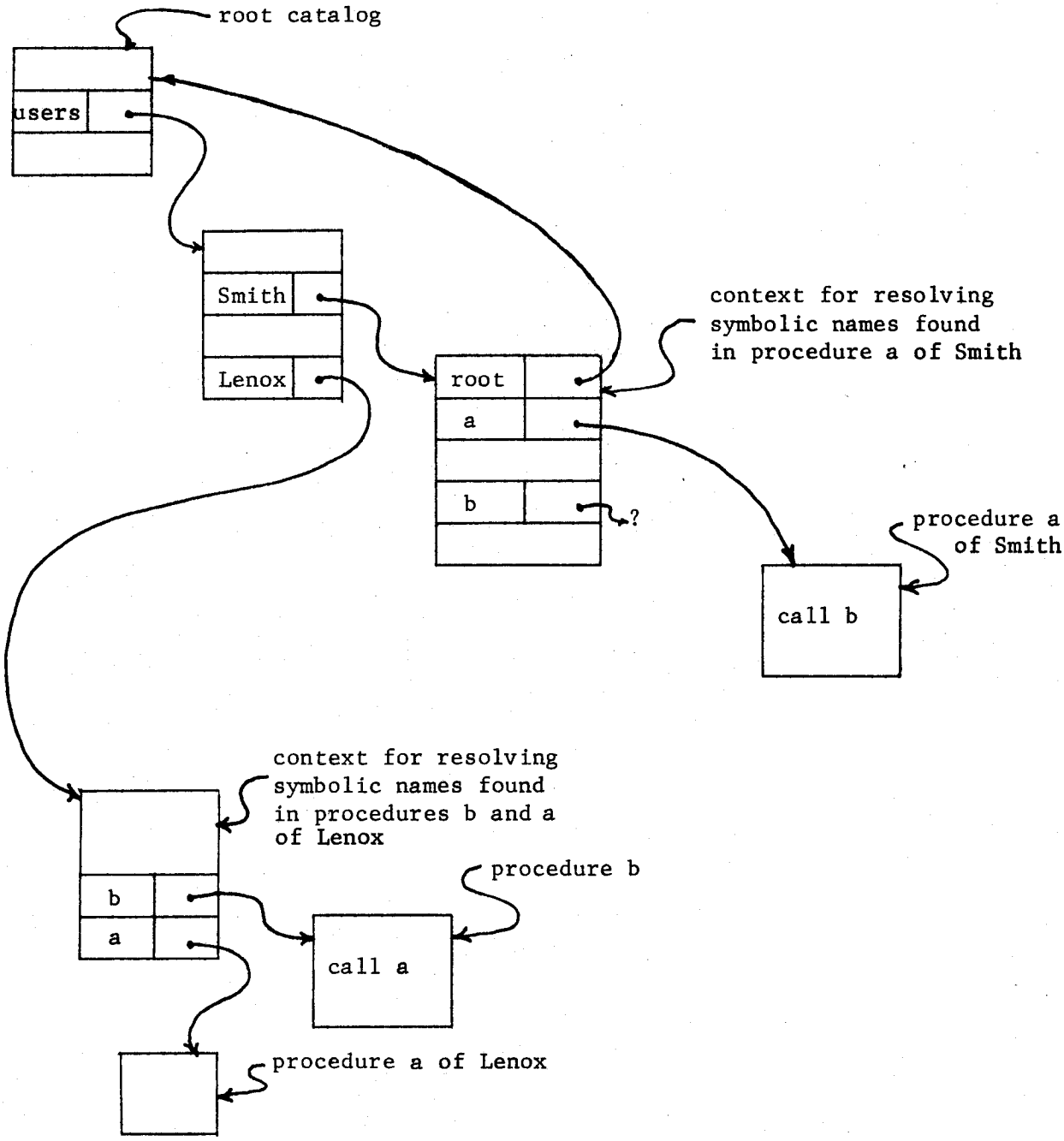


Figure 5-19 -- Sharing of a procedure in the naming network. If the name "b" in Smith's catalog is bound to procedure "b" in Lenox's catalog, the context initializer will make a mistake when resolving procedure b's reference to "a".

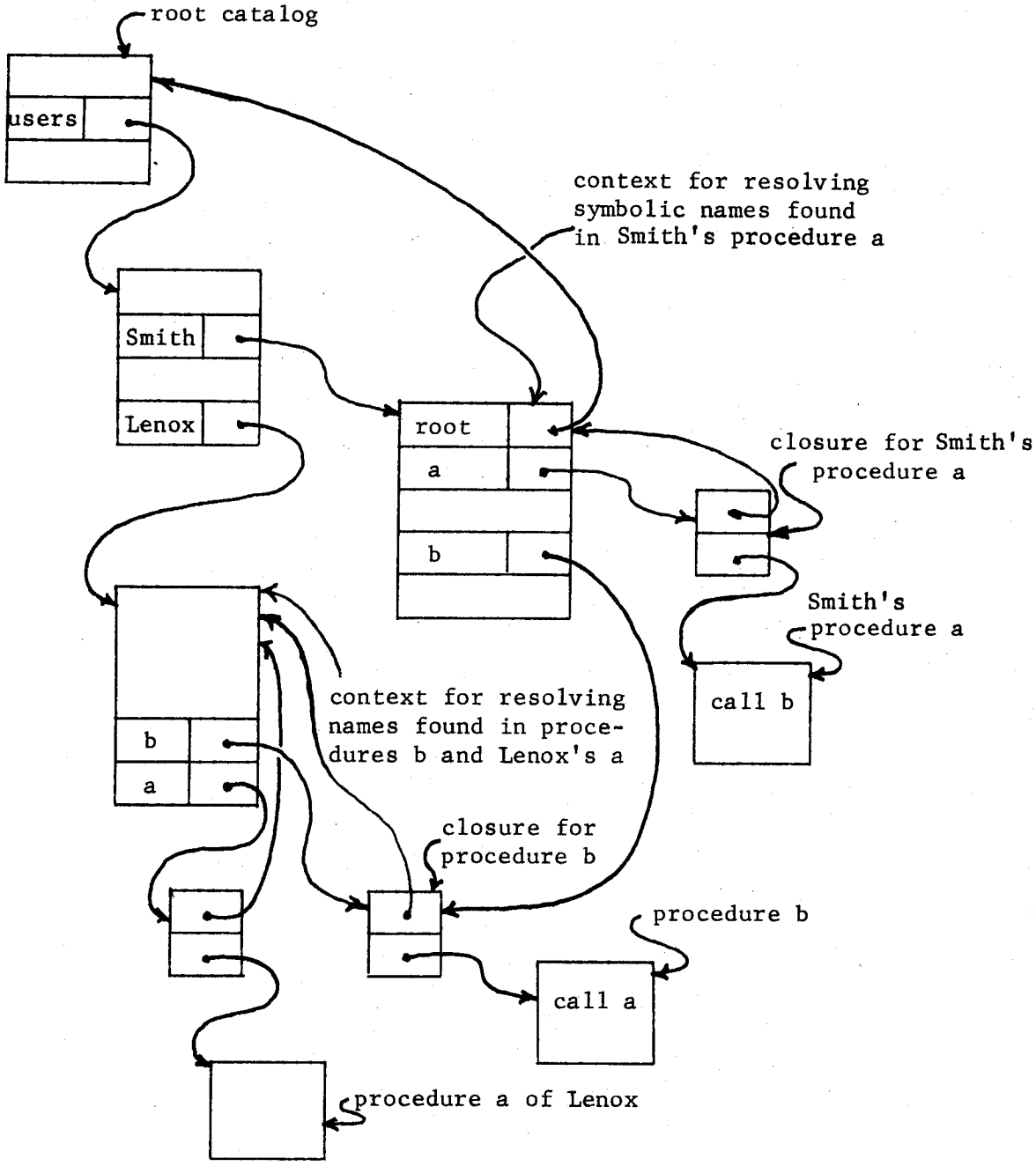


Figure 5-20 -- Addition of closures to allow sharing of procedures to work correctly.

in Smith's catalog, and it will cause initialization of a new (machine level) context for b that is based on the higher-level context of Lenox's catalog. When procedure "b" calls for "a" it will get an "a" bound in Lenox's catalog, as intended. The goal of modular sharing, namely that correct use of an object should not require knowing its internal naming structure, is achieved.

The observation that not just procedure objects, but every object that uses symbolic names should be referred to only through a closure completes out conceptual analysis of higher-level naming systems. Table 5-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 techniques actually used in practice, most of which consist of shortcuts that abridge one or more of the objectives of the naming systems of figures 5-13 and 5-20.

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 has appeared in some form in some system. In this section we explore the pressures that lead to compromises, 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

Table 5-III. The objectives of a file system, and the facilities needed 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

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 to not 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 "controlling" entry for that object. If the user ever asks to delete the controlling entry, the file system will also destroy the object.* This alternative approach appears to burden the naming system with the responsibility of remembering, in addition to the fact of a name-to-object binding, the control status for every catalog entry. What is actually happening is deeper: the mechanisms of naming and those of storage allocation are being subtly intertangled, and 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. Intertanglement 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.

---

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

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. Procedure sharing with substitutions becomes impossible. Also, it 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 places that catalogs 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 applied to the more general naming network. Consider what would happen in figure 5-17 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.)

---

* This strategy was used in the file system for the UNIX time-sharing system [Ritchie and Thompson, 1974].

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 most significant effect of this merging of considerations of naming with considerations of physical storage management, reliability, and security is that to provide control there should be only one repository for the attributes of any one object. Thus, systems with this approach usually begin with the rule that there can be no more than one catalog entry for any one object. Again, the form of the naming network is restricted to that of a routed 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.

---

* 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].

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 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 5-21 illustrates the situation of figure 5-20, except that a naming network allowing indirect entries is used.***

---

* 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.

** 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").

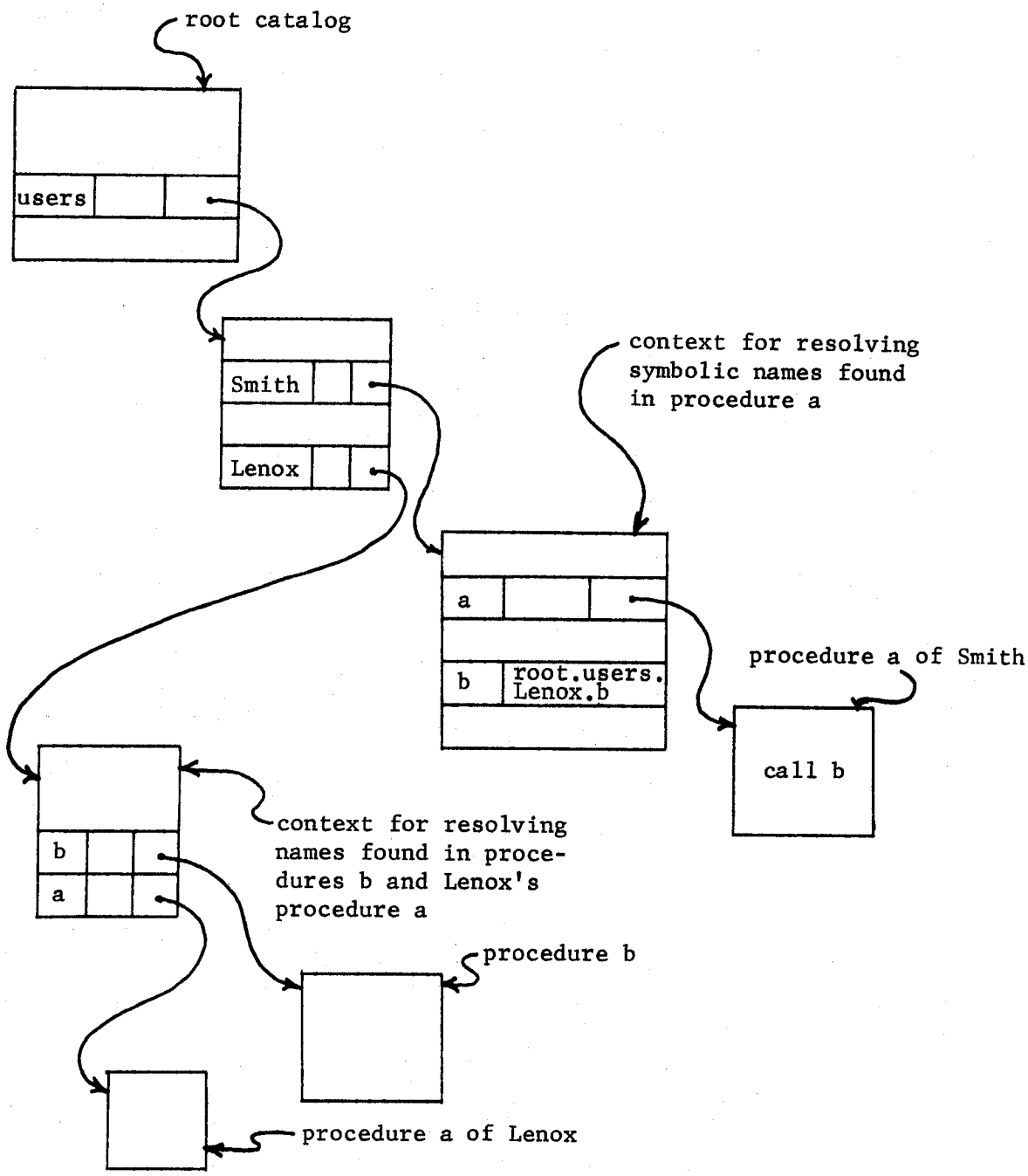


Figure 5-21 -- Sharing of procedures in a naming hierarchy with indirect catalog entries. The name "b" in Smith's catalog is bound to the tree name of Lenox's procedure, rather than its unique identifier. Since in a naming hierarchy the root catalog is distinguished, Smith no longer needs a binding for 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 outbound 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 a convenient time, locate the substituted object, and get its bindings installed for later actual use.*

Table 5-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.

E. Research directions

Name binding in computer systems, as should by now be apparent, is a relatively ad hoc, disorganized area of study, in which 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 on guidance on how to think. These problems arise whenever distributed systems are encountered.

---

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

Table 5-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	yes
independence of naming and storage allocation	yes	no	no	no	no
control of object attributes	no	no	yes	yes	yes
cross references without knowledge of tree structure	yes	yes	no	yes	yes
multiple contexts for an object	yes	yes	no	no	no
Resolving reference names: 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 used if name is reused	yes	sometimes	no	no	no
precise substitution	yes	yes	no	yes	no

A system is distributed from the point of view of naming, whenever two or more parallel and independently operating naming systems are asked to coherently cooperate 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;
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 are.

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]. And 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

Sections A.1 and A.2 of the introduction follow closely the development by A. Henderson in his Ph.D. thesis [Henderson, 1975, pp. 17-20], 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, 19xx; Watson, 1974, Chapter 2].

3) Case studies of file system naming schemes [Murphy, 1972; Bensoussan et al., 1972; Lampson and Sturgis, 1976; 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].

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 15, 4 (May, 1972), pp. 308-318. (B,SR)

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)

Dennis, J.B., "Segmentation and the design of multiprogrammed computer systems," JACM 12, 4 (October, 1965), pp. 589-602. (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.2)

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

Hedberg, R., "Design of an integrated programming and operating system, part III--The expanded function of the loader," IBM Sys. J. 2, 4 (Sept-Dec, 1963), pp. 298-310. (B.2)

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)

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

IBM (Anonymous) Reference Manual: 709/7090 FORTRAN Operations, IBM Corporation, White Plains, New York (1961). C28-6066 (A.2,A.2)

Iliffe, J., and Jodeit, "A dynamic storage allocation scheme," Computer Journal 5, (Oct., 1962), 200-209. (A.1,SR)

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)

Knuth, D., The Art of Computer Programming, Volume 1/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, 1970), pp. 13-27. (A.3)

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 implementations," AFIPS Conf. Proc. 41 I (FJCC, 1972), pp. 571-578. (B.2,SR)

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

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)

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

Problems

5.1 The structured memory system must generate a unique identifier for each newly created object. Of the two methods for generating this unique identifier that are proposed below, which is preferable and why?

Proposal 1: Maintain a 36-bit variable in the supervisor portion of primary memory which is incremented by the catalog manager each time an object is created. The current value of this counter provides the unique identifier.

Proposal 2: Add to the computer system a 36-bit, microsecond, hardware clock that has an independent, reliable power source. Use the current clock reading as the unique identifier.

5.2 Assume a computer system with unique identifier addressing and a human oriented naming network. Every object has a human readable tree name relative to a root catalog, and a machine readable unique identifier. While executing a particular user program a context initialization fault occurs that cannot be satisfied using the current context, and the system must report the name of the procedure object containing the unresolvable reference name. Unfortunately, the context initializer has available only the unique identifier of the procedure object--a name that is not very meaningful to the user.

a) Describe a scheme for converting from a unique identifier to a tree name using only the information contained in the naming hierarchy.

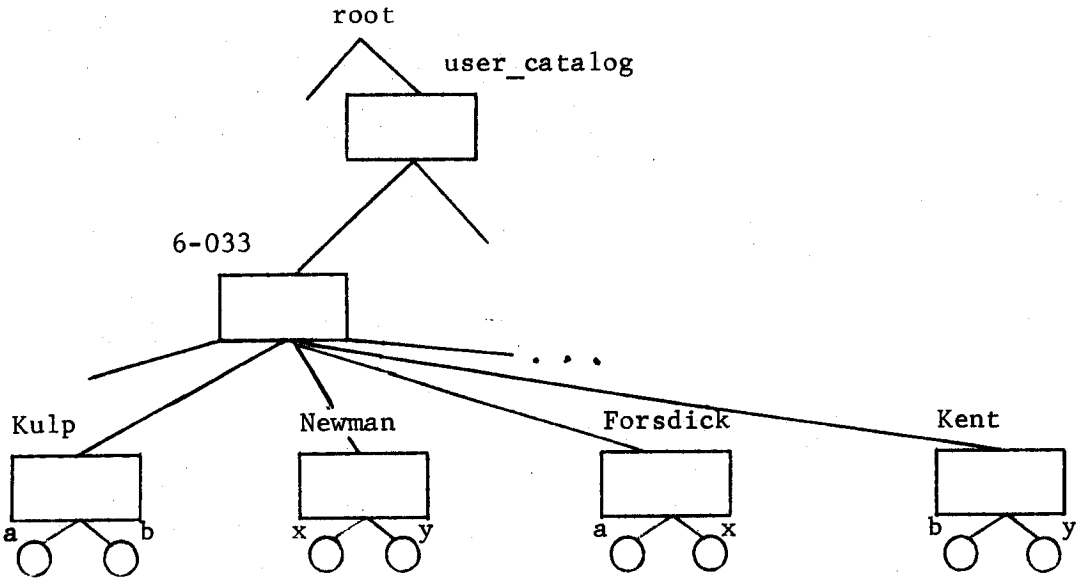
b) Propose an additional data base that would make the conversion easier to perform. Describe its maintenance and use.

c) Devise a scheme that attempts to report a path name relative to the current working catalog. Can your scheme always report such a path name? If not, characterize the conditions under which it will fail.

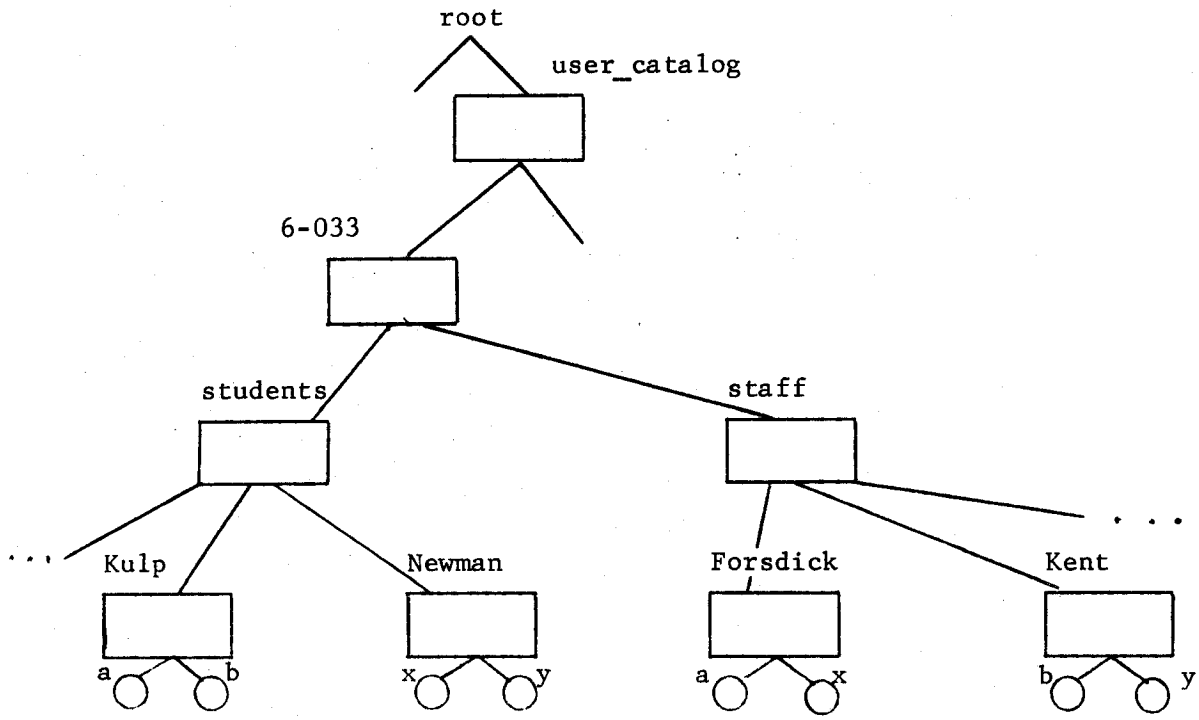
5.3 (JHS) In figure 5-20, it was assumed that Smith, not knowing that Lenox's procedure b called on something named a, did not want to disturb that pattern. Suppose instead that Smith has found out about the call to the procedure named a inside b, and he wishes to replace Lenox's a with his own procedure x, which he has since added to his own (Smith's) catalog. What additional catalog entries and objects (if any) must be created to accomplish this goal, if we also require that other users of Lenox's procedure b are not to have Smith's x substituted for Lenox's a? (A redrawn version of figure 5-20 is probably the best medium in which to answer this question.)

5.4 (JHS) Repeat problem 5.3, starting instead with the naming system of figure 5-21.

5.5 (Suggested by M.D. Schroeder) The figure below represents a portion of a naming hierarchy that might be in existence for 6,033 students and staff members.



For administrative reasons, it is decided to restructure the hierarchy so that students are separated from the staff. This is done by creating two new catalogs named "students" and "staff" and rearranging the hierarchy so that it has the structure depicted below.

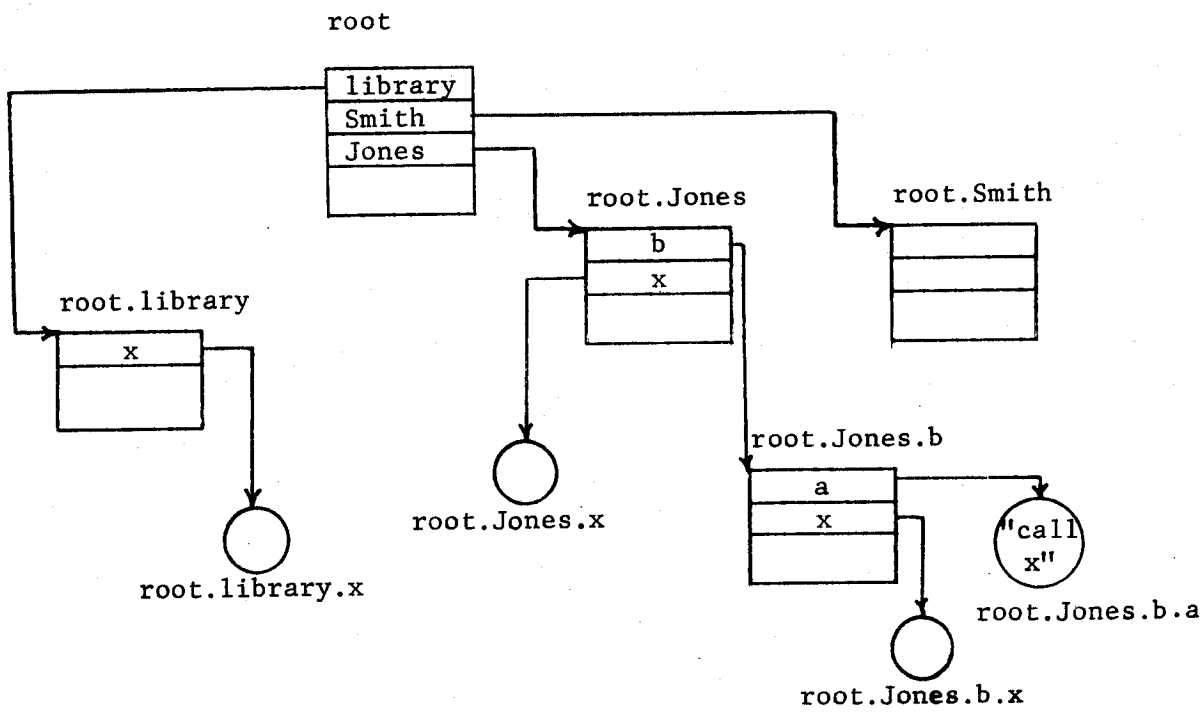


continued on next page

Problem 5.5 continued

Unfortunately, knowledge of the tree names of various catalogs and other objects belonging to the students and staff has propagated to other unknown parts of the system. For example, some other catalog in the hierarchy may contain an indirect entry specifying the tree name "root.user_catalog.6-033.Forsdick.x", and the system may have recorded in an obscure place the fact that "root.user_catalog.6-033.Kent" is Kent's normal working catalog. Thus, the restructuring may cause the catalogs and objects of the students and staff to appear lost. Proper use of indirect entries, however, can allow the restructuring without invalidating the "old" tree names for the moved catalogs and objects. Describe a set of indirect entries that will eliminate the possibility of all such problems after the restructuring is done.

5.6 (Suggested by A. Huber) Below is a portion of a hierarchically organized memory system, with catalogs denoted by rectangles and data objects as circles. Note that three different "x"'s exist, and the procedure object root.Jones.b.a contains a statement "call x".



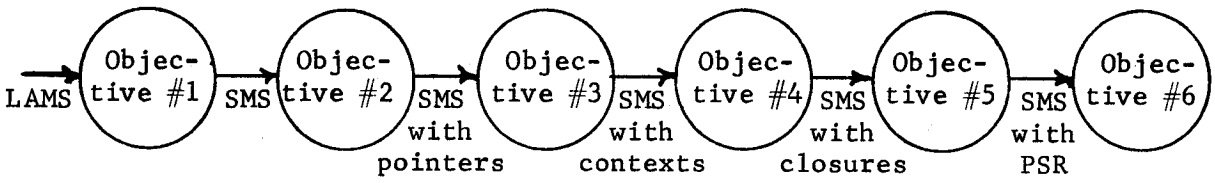
On the next page are five situations in which the order of search rules is left unspecified. In each case, the results of executing the program "root.Jones.b.a" twice are listed, with a different working catalog used in each execution. In all cases the search rules are known to be some variation of referrer's catalog, working catalog, library catalog. Determine the search rules the system must be using in each case to produce the indicated results. Note that in some cases you may be able to completely specify the order of the rules, while in others you may only be able to identify a partial ordering. (In some cases the results listed may be impossible to achieve with any ordering.)

case		working catalog	version of "x" used by "call x" statement
1.	a.	root.Jones	root.Jones.x
	b.	root.Smith	root.Jones.b.x
2.	a.	root.Jones	root.library.x
	b.	root.Smith	root.library.x
3.	a.	root.Jones.b	root.Jones.b.x
	b.	root.Smith	root.library.x
4.	a.	root.Jones	root.Jones.b.x
	b.	root.Smith	root.library.x
5.	a.	root.Jones	root.Jones.b.x
	b.	root.Jones.b	root.Jones.b.x

5.7 (Suggested by S.T. Kent) A new search rule is to be added to our list of search rules for use in resolving reference names. The rule will be called the "already referenced" rule and will function as noted below:

> Whenever a reference name is resolved, through the use of any other search rule, a new entry is made in a table, the "referenced object table", consisting of the reference name and the tree name of the object to which the reference name is now bound. The "already referenced" rule simply searches this table (from the beginning) to determine if the reference name in question has already been used. If a match is found, then the reference name will be bound to the object indicated by the tree name in the table.
>
> How does this new rule affect:
>
> a) efficiency of the search rule concept,
>
> b) the problem of having two users of a reference name be resolved to different objects,
>
> c) unexpected name resolutions.

5.8 (JHS) In table 5-I successive architectural features meet successive naming objectives, and it appears as though to accomplish any objective after the first, that meeting all the preceding objectives is a prerequisite. That is, the apparent prerequisite structure is as follows:



Redraw the above diagram to show the actual prerequisite structure required to meet the objectives.

5.9 (Suggested by J.C.R. Licklider) User "Jones" has three procedures in his catalog, named A´ B, and C. User "Smith" has two procedures in his catalog, named A and B. Jones's procedure C is executing, and is about to encounter the statements:

```
Call A
Call B
```

Jones wants the first of these statements to result in a call to Smith's A, but the second one to result in a call to his own (Jones's) B. Smith's procedure A contains a statement:

```
Call B
```

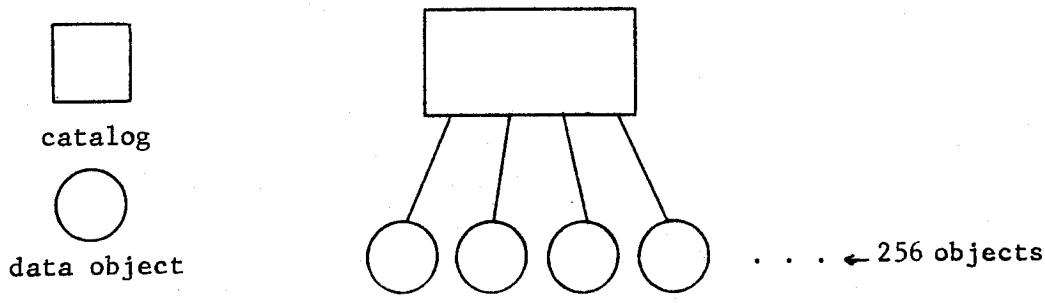
that should result in a call to Smith's own B, while Smith's B contains the statement:

```
Call A
```

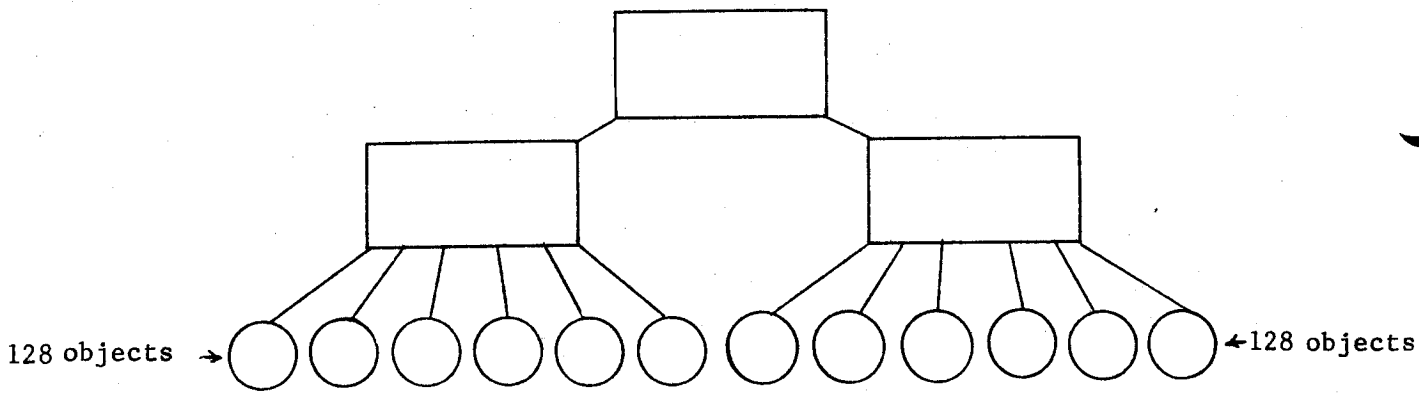
that should result in a call to Jones's procedure A. Describe and illustrate the use of a name-resolving system that allows this collection of procedures to operate as described.

5.10 Consider a naming hierarchy. Suppose that it takes exactly $(A + B \cdot n)$ instructions to search a catalog containing n entries for a particular entry whose name is known to be in the catalog.

a. For the hierarchy indicated below, in which 256 data objects are stored in the hierarchy, what is the number of instructions required to find the entry describing a segment whose tree name is known?



b. For the hierarchy below, in which 256 data objects are stored also, what is the number of instructions required to find the entry describing a data object whose tree name is known?



c. Under what condition does the second hierarchy require fewer instructions to search than does the first hierarchy?

d. Construct a general rule for deciding how many levels of catalogs and how many entries to put at each level, for a given value of A, B, and number of objects , assuming that the only concern is speed of searching. (Note: this is a hard problem.)