DATE: AUGUST 22, 1969

TO: REPOSITORY DISTRIBUTION

FROM: C. T.CLINGEN

The attached working paper entitled "Program Naming Problems in a Shared Tree-Structured Hierarchy" is to be delivered at the Conference on Techniques in Software Engineering in Rome, October 27-31, 1969.

G0090

WORKING PAPER ON

PROGRAM NAMING PROBLEMS IN A SHARED TREE-STRUCTURED HIERARCHY

C. T. Clingen

INTRODUCTION

The recent proliferation of direct-access systems with large amounts of on-line secondary storage has greatly increased the capability of programmers to share information easily. On systems such as CTSS, Multics, TSS/360, etc., the ability for a programmer to incorporate procedures written by others into his own programs has become increasingly attractive as a way to save programming time and labor, to stimulate creativity, and to encourage the solution of problems that would not otherwise be solved.

However, environments which encourage controlled sharing have given rise to a new set of technical issues centered primarily around the implementation of access control and information protection mechanisms. The topics of access control and information protection contain many difficult problems and have received considerable attention. On the other hand, certain topics relevant to program synthesis in an environment typical of a growing number of large direct access systems seem to have enjoyed relatively little exposure. Multics experience with program synthesis, however, has shown that the associated problems are easily underestimated and that solutions considered adequate for contemporary problems will not be adequate for complex information management problems anticipated in the near future.

This working paper covers program synthesis problems arising from the apparent duplication of program component names--procedure names and data base names--encountered when complexes of components are synthesized into potentially large programs. Some solutions to these problems are outlined. Also, a mechanism which permits a user to selectively replace components of a subsystem, shared concurrently by other users, with private components of his own is described and the value of such a mechanism is discussed.

The paper indicates at a fairly general level the approach taken by Multics to solve the naming problems associated with the synthesis and maintenance of shared subsystems residing in a tree-structured hierarchy.

THE ENVIRONMENT

Some systems[1,2,3] with large on-line storage capabilities permit their contained files or segments to be arranged into a tree-structured directory hierarchy consisting of non-terminal directories and terminal, non-directory files or segments. See Fig. 1. Each directory contains directory entries describing the attributes, such as length, location, access rights, etc., of all directly inferior directories or non-directories. Associated with each directory entry is a name called an entry name, uniquely identifying that entry among all other entries in the directory. Typically, each directory or non-directory in the directory hierarchy can be uniquely located by a multiple-component path name consisting of an ordered list of directory entry names, starting from the source or root of the hierarchy and leading to the item of interest. It is convenient to regard a directory as a device for grouping or "packaging" sets of related procedures into subsystems. Usually each system user will be assigned a single user directory in which all his files are described. In some systems, the user may also be permitted to add inferior directories containing descriptive information for additional files (or directories).

Note that, in such an environment, the entry name of a file does not uniquely identify or locate the file in the hierarchy; the path name, or some equivalent representation, is required instead.

PROGRAM SYNTHESIS

In most systems the synthesis of programs from their constituent data and procedure components consists of copying images of procedure and data files into a "core image" private to the user (relocating internal addresses if necessary), and of linking references between pairs of the component copies. In systems permitting the use of pure, shared procedures and shared data, copying and internal address relocation is unnecessary, but linking is still required by each user.[5] In either case, it is necessary to be able to establish a one-to-one correspondence or association between the procedure and data components required by a program and the named files in the system directory hierarchy. The establishment of an association between program component names and file path names is non-trivial and requires the cooperation of the programmer who must observe system conventions regarding this association.

If an external reference from a procedure, say a call to another procedure, is to uniquely specify the called procedure in a tree-structured directory hierarchy at linking time, then some way of determining the path name of the procedure to be linked to must be possible. One approach is to actually embed the path name of the called procedure in the calling procedure. Although this approach does have the advantage of resolving any possible ambiguity between name and object, it also suffers from several drawbacks. First, one must know the actual or intended path names of all procedures referenced by a procedure when coding the program. Secondly, if any procedures or data files are moved from one directory to another, or from one installation to another, special actions must be taken, such as recoding and recompilation of all procedures referencing the moved components. Thirdly, the substitution, by one user, of "private" replacement components for some of a collection of standard, shared components becomes extremely difficult.

An alternative method, and the one used in Multics, involves the more traditional approach of referencing external procedures and data using the usual single-component names. However, these names, called reference names, do not uniquely identify files in the directory hierarchy and hence are insufficient, taken by themselves, to locate files. The use of reference names rather than path names results in an apparent ambiguity which must somehow be resolved. In Multics, a set of system conventions is provided by which reference names are expanded to path names, thereby removing the ambiguity.

These conventions, called search rules[5], exist in some form in most systems and must be understood by the programmer if his programs are to be properly synthesized from referenced components located in the directory hierarchy.

SEARCH RULES--REFERENCE NAME EXPANSION

Perhaps the simplest example of search rules is to be found in batch systems in which card decks are read into the system to be combined subsequently with system library routines as appropriate. The search rules in this case are trivial: (1) for each external reference name, check to see if the reference name corresponds to one of the procedures read in from the card reader; if so, link to it. (2) Otherwise, search the system library for the referenced name.

An analogous set of search rules can be implemented for a directory hierarchy where the card deck is replaced by a user directory. In this environment, a user program would consist of components described in his user directory (but not inferior directories) plus, as a default, components in the system library. The search rules in this environment are implemented as follows. (1) For a given reference name indicating a procedure to be linked to, prefix the path name of this user's user directory to the reference name. If the file located by the resulting path name exists (in the user directory), complete the linkage. If not,(2) prefix the path name of the system library to the reference name and using the resulting path name, locate the indicated file. If no such file exists, the referenced file is considered to be non-existent.

This scheme has the drawback that it does not permit procedures in the user's directory to reference components in some other directory, with the exception, of course, of the system library.

The sharing or referencing of components described in a directory other than that containing the referencing procedure has been facilitated in some systems by the use of the file link. In Multics, a file link is a named directory entry which contains a path name to a file or segment described in some other directory rather than containing the attributes of some file or segment[4]. In Fig. 1., the file link with entry name d in the directory with path name root>w>y "points to" the file with path name root>x>z>d. Operationally, a file link has the property that when it is referenced by its path name, the reference is redirected to the file described by its contained path name and hence can be regarded as a form of indirect addressing. This indirect addressing capability provided by the file link permits the scope of the above search rule to be extended beyond the user directory and the system library. By placing a properly named link in the directory of a procedure referencing a program component in another directory, the attention of the search rules can be appropriately directed to the other directory.

For example (see Fig. 1), the procedure with entry name a in subsystem 1 can reference the component with entry name d in subsystem 2 with the help of the file link to root>x>z>d stored in the directory named root>w>y. Assuming that the user directory is named root>w>y, the user directory search rule, during the establishment of program linkage from a to d, would generate the path name root>w>y>d. This path name, specifying a file link, would then be replaced by the value of the file link--root>x>z>d--resulting in a search in subsystem 2 and the subsequent location of entry d in directory root>x>z as anticipated.

Designating the directory containing the referenced component as the target directory, we see that the use of file links is adequate unless a referenced procedure in a target directory in turn references further program components. If these further components do not reside in the original user directory or in the system library, the search rules thus far stated will fail to locate a component with the desired reference name or worse yet, may find by accident incorrect components which by coincidence have the required reference names.

An example of incorrect component selection arises when "duplicate" reference names exist. Continuing the above example, assume that procedure d in subsystem 2 calls procedure a, also in subsystem 2. See Fig. 1. The search rule, as currently stated, however, is based in the user directory, or in this case, subsystem 1 directory named root>w>y. Thus the search rule will generate a path name by prefixing root >w>y to a, thereby locating the wrong component. That is, the file with entry name a in subsystem 1 will be located and linked to from procedure d rather than the correct file with the "duplicate" entry name in subsystem 2. Clearly an improvement or extension over the user directory search rule is required if packages or subsystems of program components are to be properly synthesized into a single program.

One such extension is the caller directory search rule. The caller directory search rule states that the entry name for a referenced component should first be searched for in the directory containing the procedure originating the reference. In short, by convention, reference name scope is defined as being within the directory containing the "calling" procedure (except for the default scope of system library). The file link provides an "escape" mechanism to other directories or subsystems while still respecting the caller directory convention.

In order for subsystems, grouped by directory, to be unambiguously linked together during program synthesis it must be possible to redefine the current value of the caller directory each time the attention of the program linking facility is redirected to another directory. In Multics, the value of the caller directory is guaranteed current by being re-evaluated each time an external reference is linked by the program linking facility. Such frequent re-evaluation is required in Multics because program linkages are built dynamically at run time upon first reference to a program component. In a less dynamic environment, such as encountered when batch linking is utilized, the value of the caller directory might be redetermined only when the linkage facility encounters and follows a file link.

The search rules as so far described may be summarized as: (1) search the current caller directory; if this search fails, (2) search the system library. By the use of file links pointing to other subsystems, programs of arbitrary complexity can be synthesized. The caller directory search rule minimizes the need for programmers and users of mutually interacting subsystems to be concerned about the apparent duplication of reference names of components in different directories. With the exception of the procedures used to interface among the interacting subsystems, internal subsystem component names can be chosen with no fear of mutual conflict.

COMPONENT REPLACEMENT

Modification, upgrading and debugging of subsystems in an environment encouraging sharing have interesting implications. In some instances, the person performing the changes can make a private copy of the subsystem of interest and then work with the now unshared copy. In cases where the subsystem being modified is large, as are complicated language processors, comprehensive applications systems, or the operating system itself, copying is impractical. To permit system programmers to extend and maintain large, shared subsystems easily, the supervisor must provide a facility whereby each programmer can uniquely specify individual components to replace other uniquely identified components.

Initially, a partial solution to the substitution problem was implemented in Multics by introducing a third search rule to a user-specified directory called the working directory. This search was inserted between the caller directory search and the library search. By placing substitute versions of a library procedure in his working directory, the programmer receives his own private versions rather than the standard library versions. Such an approach does substitute working directory components for library components required by procedures in the user's user directory; however, it also performs the same substitution for library procedures referenced from other subsystems incorporated into his program. The latter substitutions introduce an undesirable element of uncertainty into the program since the user may well not be aware of the (perhaps invalid) substitutions performed on behalf of these "foreign" subsystems. Furthermore, the additional search rule does not permit substitution for non-library components.

An alternative "solution" to non-library substitution, consisting of interchanging the caller directory search rule and the working directory search rule, is quickly dispensed with upon considering the potential problems caused by components in "foreign" subsystems being replaced by working directory components. Since the working directory rule would precede and, therefore, override, the caller directory rule for all subsystems, any chance coincidence between unknown entry names in these other subsystems with names of user directory components to be substituted would result in effectively random and unwanted substitutions.

For example, imagine that a programmer wishes to test a new version of procedure a in subsystem 1 (see Fig. 1). To do so, he places this new version of a in some other directory, the hierarchy location of which is unimportant to this discussion, designated as his working directory. Upon linking from, say procedure b to procedure a in subsystem 1, the working directory rule is first envoked resulting in the improved version of a being used. However, when subsystem 2 is envoked in his program, an entirely unrelated procedure which by coincidence also has the reference name a is called, presumably unbeknown to our programmer. Unfortunately, the working directory is again used to perform the search prior to program linkage. This time an undesired substitution takes place, and the program yields erratic results.

To avoid such indiscriminate substitutions, it appears that a useful, unambiguous way to define each desired substitution is to specify path name pairs--one path name uniquely locating a component to be replaced and the other uniquely locating the replacement. These replacement pairs could, by convention, be placed in a file known both to the programmer and to the system and accessed by a substitution search rule. The search rules, during program synthesis, then become: (1) check all replacement pairs to see if a substitution is required; if not, (2) use the caller directory search rule; if this fails, (3) use the default search in the system library.

Neither substitution by replacement pairs or an equivalent method has yet been implemented in Multics. However, as the on-line upgrading and debugging of large, highly shared subsystems becomes a significant activity, such a facility will be provided.

CONCLUSIONS

The introduction of program components (procedures and data), mutually referenced by means of single-element reference names, into a tree-structured directory hierarchy in which elements are uniquely located by multiple-element path names allows the possibility of ambiguous referencing by name. System conventions must be established whereby reference names are "expanded" to path names when programs are synthesized from individual components. If complex subsystems are to be included in a program, search rules and packaging conventions must be provided to properly identify and utilize potentially ambiguous reference names.

Moreover, in large, utility-like systems with many users sharing subsystems or the system software itself, means for achieving selective component substitution, while the system runs unchanged for most users, is valuable if a smooth maintenance and improvement facility is to be provided. This capability is extremely useful in shared-access systems for the on-line maintenance and extensions, by system programmers, of system components proper. The need for these capabilities for user-provided subsystem maintenance will increase as shared-access system use becomes more intense and wide-spread.

Practical experience in Multics has shown that well-implemented, easily-understood search rules are essential if full advantage is to be taken of a large hierarchy of program components; conversely, poorly-implemented search rules can lead to confusion and bizarre accidents.

ACKNOWLEDGMENTS

I would like to acknowledge the helpful comments and discussions of A. Bensoussan, F. J. Corbató, R. C. Daley, J. W. Gintell and J. H. Saltzer.

REFERENCES

1. COMFORT, W. T. A Computing System Design for User Service. Proc. AFIPS 1965 Fall Joint Computer Conference Vol. 27, Pt. 1, Spartan Books, New York, pp.619-628.

2. CORBATÓ, F. J., and VYSSOTSKY, V. A. Introduction and Overview of the Multics System. Proc. AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Part 1. Spartan Books, New York, pp.185-196.

3. CRISMAN, P. A. ed. The Compatible Time-Sharing System: A Programmer's Guide, 2nd ed., MIT Press, Cambridge, Mass., 1965.

4. DALEY, R. C., and NEWMANN, P. G. A General-Purpose File System for Secondary Storage. Proc. AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Part 1. Spartan Books, New York, pp.213-229.

5. DALEY, R. C., and DENNIS, J. B. Virtual Memory, Processes, and Sharing in Multics. Comm. ACM 11, 5 (May 1968), 306-312.

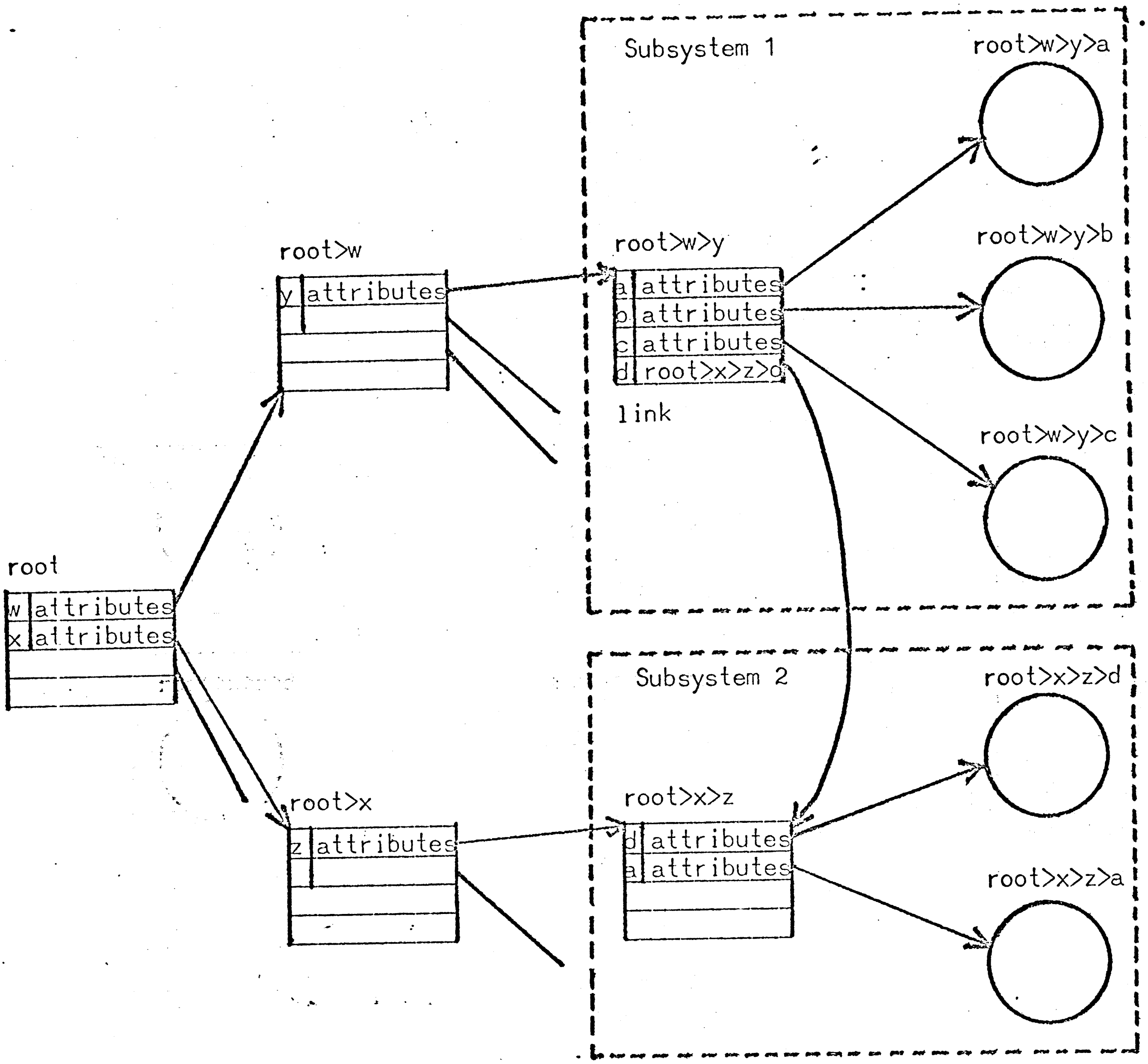

Figure 1. Directory Hierarchy Structure