1985

The TILDE File Naming Scheme

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Thomas P. Murtagh

Report Number:
85-507

The Tilde File Naming Scheme

Douglas Comer

Thomas P. Murtagh

CSD-TR-507

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

Hierarchical directory systems permit users of a computer system to organize files into meaningful sets. A key part of the directory's usefulness lies in the way it partitions names. A hierarchical directory system supports short mnemonic names by keeping the names of files in one directory independent of those in other directories. Most hierarchical directory systems, which were designed for single-processor time sharing systems, place files in a single directory tree. The disadvantages of the single tree scheme become apparent when two such systems are interconnected by a computer network, or when software is transported from one machine to another.

This report proposes an alternative to the single-tree naming scheme in which files are organized in a forest of tree-structured directories and each user choses his own view of the forest. It becomes necessary to divorce the directory tree from the processor in order to provide meaningful file naming in a distributed environment. Separating directory trees from processors also has advantages in single-processor systems.

This work was supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

Introduction

A general purpose operating system provides at least two interfaces to the machine on which it runs. One interface is through system calls that may be invoked by programs running on the machine. The other interface is the command language through which users interact with the system. The operating system must provide a way to name files through both of these interfaces.

Within the command language it should be possible to use names that are easy to express (e.g. short) and easy to read and remember (e.g. mnemonic). This essentially requires that users be capable of specifying local abbreviations for file names, since it is impossible to simultaneously provide distinct short mnemonic names for all the files in a large system. Within programs, on the other hand, it is frequently necessary to use file names that are unique within the system. Many programs access files that are directly associated with the program, rather than with the program's user. For example, an editor might reference a file of default key bindings that was created by the editor's author. The interpretation of such a file name must be independent of the abbreviations established by the program's current user, since it must be interpreted the same way for all users. To ensure this, most systems require that each file have a name that is unique within the system and require programmers to use such names when including file names in their code.

In most operating systems, file names are based on directories of files maintained by the system. While the directories in some systems are tied to physical storage volumes, the file systems of Multics[5] and Unix[7]† provide users with the ability to create directories and sub-directories independent of the physical structure of the system's secondary storage. Device independent directory organizations provide users with the ability to organize their

† Unix is a trademark of Bell Laboratories

files into directories more logically. Such systems are known as *hierarchical directory systems*.

A hierarchical directory system provides a naming mechanism that is excellent for interactive users. The concept of a *current working directory* allows users to work with short, mnemonic names easily. The user never has to explicitly specify abbreviations for file names. He simply move the current working directory to the directory containing those files with which he wishes to work. This action implicitly introduces abbreviations for the names of all the files in the directory. In addition, the abbreviations introduced in this way are cleanly integrated with the unique file names that programs must used to reliably identify files. Abbreviated names can be converted to full names and vice versa by simply adding or removing a prefix consisting of the full name of the current working directory.

Hierarchical directory systems, however, do little for the file naming problems of programmers. To identify a file reliably, a program must use a file name determined by the path through the directory tree from its root to the file. Such names are guaranteed to be unique throughout the system, but their use becomes a significant problem in a least two areas: the production of software for distribution to independent computer systems and the construction of a file naming scheme for use in a distributed computer system.

In this paper we present a modification of a conventional hierarchical directory scheme intended to address these problems. Our approach is based on two facts. First, we observe that the need of a program to reference files produced by its author does not imply that the names used for such files must be globally known and globally unique. It only implies that the names must be known in the environments of all users who execute the program and that users' local abbreviations for files must not interfere with these names. Accordingly, we give the user control over the specification of two collections of abbrevia-

tions that can be used for file names. One is just the current working directory, as found in conventional hierarchical systems. The other is a more stable collection of abbreviations that must be tailored to meet the naming requirements of all the programs the user references.

Second, we observe that while most systems organize files into a single directory tree, this tree is composed of many logically independent subtrees each of which is itself composed of related files. We call such subtrees *project sub-trees* or *tilde trees*. Requiring the user to maintain file abbreviations for all the programs he uses would be burdensome if it had to be done at the level of files. What we propose instead, is to have the user provide an environment through which programs can conveniently name tilde trees and then select files from within the trees.

In our system, file names are interpreted relative to a forest of trees. Each user's file names are interpreted in a distinct forest that the user controls by adding and removing tilde trees. The user can tailor his forest to provide the programs he executes with a reliable means to access the files they need without specifying globally unique names.

Our discussion of this naming scheme will be divided into 7 sections. In the next two sections, we will discuss the relationship between full path names and location dependency and explain how the notion of project sub-trees is related to this problem. In sections 3, 4, and 5 we discuss the details of our naming scheme and show how it reduces location dependency problems in both conventional and distributed computing environments. Sections 6 and 7 discuss remaining problem and work on a prototype implementation.

1. Full Path Names and Location Dependency

In some file systems, a file's name depends explicitly on the name of the device on which the file's data is stored[3]. The system maintains an independent directory of files on each physical volume. File names within each directory must be unique, and each volume is given a name which is unique within the system on which it is used. A unique name can be formed for any file by combining the name of the volume on which the file resides with the file's name within the volume.

This approach has a significant weakness. Because file names are location dependent, a file's name must be changed if it ever becomes necessary to move the file from one volume to another. When this happens, any other files containing program text or command language statements that refer to the moved file must be updated to reflect the change in its name.

Hierarchical file systems seem to eliminate this problem, because a path name does not explicitly depend on the volume on which a file is stored. In fact as long as one restricts one's attention to a single computing system, hierarchical file names can be location independent. If, however, one takes a larger view and recognizes that important files are likely to be transported to many different computer systems during their lifetimes, a new form of location dependency appears. The dependency is not as obvious as that associated with volume-specific file names. Usually, hierarchical file names do not explicitly include the name of the machine on which the file resides. However, they do depend on the structure of the directory tree of the machine on which the file resides.

When one creates a file in a hierarchical directory system its full path name is forced to depend on the structure of the upper levels of the directory tree on that system. It may be difficult or impossible to use the same name for the file when it is copied to another sys-

tem. For example, suppose user X moves from system A where his root directory is named '/usr/X'† to system B where the root directory '/usr/X' has already been assigned to another user. User X will be forced to accept a new home directory name and forced to have the names of all his files change as a consequence. Thus, while the user's names did not explicitly depend on the name of system A, they did depend on the structure of the directory hierarchy on A.

This form of location dependency is not quite as severe as that associated with file names that explicitly depend on device names. While file names that include device names always have to be changed when the file is moved, a file's path name only has to be changed if the upper levels of the directory tree in which it is stored are reorganized or the file is moved to a new system in which the original path name is not available. When the problem does occur, however, its consequences are just as serious.

To completely escape from such problems, one would have to gather all the files stored on all computing systems into one location-independent naming scheme. A universal hierarchy, however, is both technically impractical and politically impossible. Accordingly, we must assume that the files will be subdivided into groups for naming purposes and try to minimize the inherent location dependencies.

We believe that one important step toward minimizing location dependencies should be to carefully consider how we partition files up into groups for naming purposes. Obviously, just because two files have been placed on the same disk volume or even on the same computer system does not imply that they are so closely related that they belong in the same partition of a naming system. We believe that naming should be based on partitioning files into logically related groups so that when location dependencies do occur they are not

† The conventions used in examples of file names within this paper are those of the Unix operating system.

the consequence of storage allocation decisions that are unrelated to the functions of the files involved.

2. Project Sub-trees and Naming

In systems that provide hierarchical directories mechanisms, user's typically structure their directories in such a way that all of the files directly associated with a particular project can be found in one sub-tree of the system's directory hierarchy. This subtree itself may be divided into subtree's containing source files, object files, documentation file and data files. The important thing, however, is that all of these files can be found in one "project sub-tree".

When a program uses a full path name to identify a data file in its project's sub-tree, the path can be divided into two portions. The first specifies the path from the system's root to the root of the project's sub-tree. The remainder of the path identifies a particular file within the project's sub-tree.

When a program is distributed to other sites, a significant portion of its project subtree is likely to accompany it. Accordingly, the second portion of a full path name for a data file in the sub-tree is likely to remain accurate. On the other hand, the sub-tree's location within the directory structure of a new system is unlikely to be identical to its location on the original system. Thus, the first portion a full path name will become invalid on the new system, making full path names non-portable.

If a program could give file names relative to the root of its project sub-tree, programs could use just the second portions of full path names to identify files. This would obviously improve portability. Unfortunately, specifying names relative to a project sub-tree is not possible in current hierarchical directory systems. If a program does not depend on the current working directory it inherits, then it can avoid using many full path names by

making the root of its project subtree the current working directory. This operation itself, however, will involve the use of a non-portable full path name.

Two features of conventional hierarchical directory systems make them unable to support naming relative to project sub-trees. The first is that they provide no mechanism to distinguish a directory that is the root of a project sub-tree from any other directory. In conventional hierarchical directory systems, there are no distinguished directories. Thus, a program must give a path name to make its project subtree's root become the current working directory. The system has no way of automatically determining which directory is the root of a program's project subtree. Systems that view all directories uniformly have advantages, especially when compared to systems which treat certain directories (such as those containing commands) specially. We believe, however, that allowing users to indicate that certain directories are of special significance could strengthen the systems naming mechanisms.

The other difficulty is that conventional systems can not simultaneously support naming relative to more than one project sub-tree. That is, even if the project subtree's root is automatically made the current working directory, there is no way to make refences to files in another project's subtree without resorting to a full path name. Such references must be supported, because even if they do not occur explicitly in the code of a program, the program may make them when accessing files passed to it as arguments. For example, in UNIX, file names are passed as arguments rather than the files themselves. If one program passes a name specified relative to its project subtree to another program which immediately makes its own project subtree's root the current working directory the name will be misinterpreted.

Jones[4] has suggested that the problems causes by passing relative names into a new environment could be solved by automatically converting them into full path names as they are passed. We have rejected this approach for two reasons. First, it may significantly constrain the mechanisms for manipulating file name provided by a system, because it requires the ability to recognize file names in commands and data files. In a system like UNIX, it is currently possible to pass file names or parts of file names as strings and to even generate file names through string operations. Such features would have to be limited to make it possible to recognize all file names at the point where they should be expanded. We would prefer to devise a mechanism that would reduce file naming problems while leaving the systems designer more freedom to decide how file names should be manipulated. In addition, as noted above, file name parameters are just a special case of the problem of programs that reference files in a project subtree other than their own. We wish to address this more general problem.

3. The Tilde Naming Scheme

The Tilde naming system attempts to reduce location dependency problems by making all file names relative to project sub-trees. Instead of organizing the system's files into one directory tree, we partition the directory system into a set of logically independent directory trees called tilde trees. The intent is that most tilde trees will correspond to one project sub-tree.

We associate two names with each tilde tree. The first is the name that programmer's use to denote the project subtree; we call it the tree's *tilde name*. Files within a tree are accessed using the tree's tilde name and a path. Syntactically, a file name of the form:

$$\sim T / \alpha$$

is used to identify a file within a tilde tree, where T is the tree's tilde name and α is a

Unix-like path name relative to the tree named by T. A tree's tilde name need not be unique. The second name associated with each tilde tree is its absolute name. Each tree's absolute name must be unique. A set of absolute tree names is associated with each user. The system resolves a user's references to tilde names relative to his set of absolute names. The user can, but will rarely need to, change the set of absolute names that the system associates with him.

Independently, either of the names associated with a tilde tree is insufficient. Since tilde names are not unique, they cannot be reliably used to select a desired tilde tree from the system's collection of tilde trees. Since absolute names must be unique they will suffer from all the problems discussed in the preceding sections. They must either be long or non-mnemonic. They will be location dependent to some degree. Used together, however, these two forms of name can greatly reduce file naming problems.

3.1. Using Absolute Tree Names

In the Tilde system, the absolute names of tilde trees are not used to directly name files. Instead, they are only used to describe subsets of the collection of tilde trees accessible within the system. Such a subset is associated with each process running on the system. The set of tilde trees associated with a process is called the process' tilde environment. When a process manipulates (e.g. opens) a file with name

$$\sim x/\alpha$$

'x' is interpreted relative to the process' tilde environment. Thus, even if many trees have the same tilde name in the system, a tree's tilde name will generally be sufficient to identify it within a process' tilde environment. The system provides primitives that allows a process to interrogate and change its tilde environment. New processes inherit their initial environments from their creators.

We will not discuss the details of the format of absolute names or the primitives used to manipulate tilde environments until Section 5. However, there are some aspects of the use of these facilities that must be emphasized at this point. We anticipate that each user will maintain a persistent set of absolute tilde tree names. This set of absolute names will be used to establish a new environment for the user's root process (i.e. login process). The root process' environment will normally be inherited unchanged by all of that user's other processes. Thus, while each process technically has an independent tilde environment, we will often associate an environment with a user rather than a process.

A user may occasionally modify his set of absolute tree names to add or remove trees from his tilde environment. This, however, is the only time that absolute tree names will be used. All references to files within trees will assume that the desired tree has been included in the tilde environment and use a name relative to the tree's tilde name. As a result, the problems associated with absolute naming will be minimal. Location dependency is not as critical. If it is necessary to change a tree's absolute name, each user of the tree must update his specification of his tilde environment. It is not necessary, however, to search though programs, command scripts and data files for hidden references to the name. References to files within the tree that appear in programs, command scripts and data files will be made relative to the subtree's name within the user's tilde environment. These references will all function correctly, once the user updates the specification of his tilde environment. Thus, we have removed location dependent names from programs and collected them all in one location – the tilde environment.

3.2. Minimizing Tilde Name Conflicts

While changing a tilde trees absolute name only requires rather simple updates to user environment specifications, changing a tree's tilde name is just as troublesome as changing

the path to a subtree in a standard hierarchical directory system. The Tilde naming scheme does nothing to make the process of changing a tilde name easy. Instead, it has been designed to ensure that it will rarely be necessary to change a tilde name.

As suggested above, the use of a distinct environment for each user rather than a single environment for the entire system is one feature included to minimize the need to change tilde tree names. Even if many tilde trees in the system have the same tilde name there will be no need to even consider changing any of their names as long as no user needs to access more than one of them. Most project sub-trees will only be of interest to a single user or a small group of users. Having independent environments allows small groups of users to select tilde names without regard for how the name is used by other groups in the system.

Even in the case that a single user needs access to two or more trees with the same tilde name it is often possible to avoid renaming trees. The problem with allowing several trees in a single environment to have the same name is that it makes some file names ambiguous. Ambiguities are unlikely, however, as long as the number of trees with the same name is relatively small. Tilde tree names rarely appear alone; normally, they are used as the prefix of a name that identifies a file. When a tree name that is associated with more than one tree in an environment is used as the prefix of a file name, the suffix of the name can be used to resolve the ambiguity of the prefix. For example, even if two trees are named X, the file name

~ X/a/b/c

is unambiguous as long as the path "a/b/c" is only valid in one of the trees.

The Tilde system allows the user to include trees with identical tilde names in an environment and attempts to resolve ambiguous tree names based on context. In the case

that a file name cannot be resolved unambiguously, the system must either attempt to "guess" the right file or refuse to process the name.

We feel that such names will be sufficiently rare that it will be appropriate to select one of the named files heuristically. In particular, the system will keep trees with identical names within an environment in an ordered list. A reference to such a group of trees will always be resolved in the first tree in the list in which it can be resolved. The user will determine the initial ordering on the trees in his environment. The system may reorder the list in certain circumstances to increase its accuracy. For example, if a process is created to execute a program stored in a tilde tree whose tilde name conflicts with other trees in the creator's environment, the system will place the tree from which the file was selected to the head of the list of trees that share its name.

4. Tilde Names and Distributed Systems

As networking facilities become increasingly available and sophisticated it will become common for users to desire to access files on remote machines. To do so, they will need some way to name remote files. Distributed operating systems have taken two basic approaches to the question of how to name remote files. Some systems attempt to keep file naming completely separate from the physical structure of the underlying system. For example, in the LOCUS system[6], all files the system are organized into one tree structured directory system with with no constraints relating the machine on which a file is stored to its location within the directory system. Other systems explicitly include the name of the machine on which a file resides in its name [1,8].

Support for location transparent file naming within a distributed system complicates its implementation. Since the user cannot optimize file placement, the system must try to.

Since processing of all full path names involve references to the directory files in the upper levels of the directory tree, it is helpful to replicate these directories to reduce network traffic. Replication, however, requires synchronization mechanisms to maintain consistency as updates are applied. With the addition of such synchronization, care must be taken to ensure that these directories do not become a bottleneck. Even if these technical difficulties could be overcome, organizational considerations would make it impractical to attempt to combine all the systems that now provide one another with remote access to their files into one distributed system with location transparent file naming. Thus, at some level one must expect to find users providing location dependent names to access files over a network.

We assume, therefore, that a realistic view of distributed systems must deal with different levels of location transparency. It is likely that most machines will belong to tightly coupled systems of several processors in which all files names are location transparent. These system themselves, however, will form a more loosely coupled system in which some names must include location dependent information.

The Tilde naming scheme functions well in a distributed environment. The absolute names included in a process' tilde environment may be allowed to contain location dependent information when necessary. That is, each tilde tree's absolute name would identify the system on which it was stored. An absolute name would not identify the machine within a system on which the tree resided. It would only identify the system. Thus, the names of all the trees within a tightly coupled group of machines would be location transparent within that system.

This extension would allow the user to include both local and non-local tilde trees in his environment. Accordingly, all files, whether local or remote, could be identified using

names of one form, namely

$$\sim T / \alpha$$

This would have the obvious advantage of making most references to non-local files location independent. If the site on which a non-local file was stored changed, only its name in the tilde environment specification would need to be changed. In addition, it might provide some performance improvements for references to local names. As mentioned above, in a distributed system with a shared hierarchical directory system, the upper levels of the directory tree form a bottleneck. In the Tilde naming scheme, these upper levels are replaced by the mechanism used to resolve absolute tilde tree names. In a conventional system, the upper levels of the hierarchy must be traversed each time a file is opened. In the Tilde system, on the other hand, absolute Tilde names need only be resolved when a new environment is created - an event which normally occurs only when a user logs in.

5. Process Environment Control

Thus far, we have neglected to discuss the mechanisms used to change a process' environment, including the structure of absolute names. We have postponed this discussion, because the details of these mechanisms are irrelevant to our primary concerns. They have little effect on the location transparency problems we have been addressing. They will, however, have a significant effect on a user's perception of the system.

A user, in our system, will have to specify an environment that is appropriate for all the programs he normally expects to use. This could be a significant task. Therefore, we must make the process as convenient as possible. It must not present a constant burden to the user. Rather, it should require about the same effort as other actions performed to customize the user's computing environment, such as the specification of terminal characteristics in a profile script.

One essential is that the specification of a tilde environment be trivial for a user who only needs standard system software. The system administrator should be able to provide the names of all the standard tilde trees to such users. This should be done in such a way that users who need to add their own tilde trees to their environment can also depend on the system administrator for the specification of the standard trees. In particular, when the administrator adds a new standard tree, it should automatically be included in each user's forest, even if a user has added his own trees to his environment.

To meet these requirements, the specification of a process' tilde environment should be based on forming the union of sets of tilde trees. The system should provide commands that allow users and the system administrator to maintain named sets of tilde trees. The system administrator will maintain one or more sets containing the system's 'standard' tilde trees. Users who need access to tilde trees not included in the system administrator's sets will form their own sets. Then, a process can request a new environment by passing the system the names of the sets of tilde trees that should be combined to form the new environment. The ordering of the subset names will determine the order the system uses to resolve references to files in tilde trees with identical names.

A mechanism for specifying a tilde environment by naming sets of tilde trees will provide naive users with a 'default' tilde environment. Naive users will request this environment by simply passing the system the name of a standard set of trees maintained by the system administrator. At the same time, users who manage their own trees will also be able to use collections maintained by the administrator. Such users will pass the system a list naming one or more of their own sets of trees in addition to sets maintained by the system administrator. Conflicts between the names of the user's trees and the standard trees will be avoided by appropriately ordering the subsets specified. Changes made by the

administrator will be automatically reflected in the user's environment at his next log in.

This proposal now involves two naming schemes in addition to that for file naming – absolute tilde tree names and names for sets of absolute names. We assume that absolute tree names will consist of a network address for the system on which the tree resides and a unique name for the tree within that system. For naming sets of tilde tree names, a simple hierarchical naming scheme might be appropriate. In fact, set of absolute names might be kept in files in a distinguished tilde tree on each system.

6. Reliable File Naming

In our system, we assume that programs will depend on their users to establish an environment appropriate for their execution. This is not always a safe assumption. A program that executes with access privileges that exceed the privileges of its users must be very careful when it depends on the environment established by a user. For example, the Unix system provides a mechanism called *set-user-id* programs. A set-user-id program executes with the privileges of its owner rather than the privileges of its invoker. In addition, Unix interprets command names relative to a list of directories called the *command path*. Normally, a program uses the command path established by its invoker.

If a malicious user determines that a set-user-id program invokes a command named 'x' he can subvert the systems protection mechanisms by altering his command path in such a way that 'x' is interpreted as the name of his own program before invoking the set-user-id program. If the set-user-id program does not alter the command path established by its invoker, the reference to 'x' will cause the system to execute the user's program with the privileges of the owner of the set-user-id program. To avoid this insecurity, set-user-id programs in Unix typically change the command path before they begin to execute commands.

Just as they do in Unix, privileged programs will need to guarantee the correct resolution of names when run under the tilde system. The tilde environment will determine the interpretation of all file names – including command files. This implies that if a privileged program depends on the tilde environment provided by its user and references another program file, the user can lead the system to execute one of his own programs in a privileged mode.

Unfortunately, it is not as easy for a privileged program to avoid depending on its user's tilde environment as it is for a program in Unix to avoid depending on the command path. First, in order to specify a new tilde environment, a privileged program would have to supply the system with a list of absolute tree names. Unfortunately, adding absolute tree names to programs make them location dependent. This is not a new problem. The names a set-user-id program uses under Unix to change the command path are also location dependent. The use of location dependent names, however, is exactly what the Tilde naming scheme was designed to avoid.

Furthermore, even if one is willing to embed location dependent names in privileged programs, it may not be a simple matter to change the program's tilde environment. If the invoker passes file names as parameters to a privileged program, these names should be interpreted relative to the invoker's tilde environment. If the program attempts to use these names after changing the tilde environment to ensure its security, the system is likely to misinterpret the names.

In general, the problems involving name misinterpretation can be avoided by having privileged programs check the environments they inherit in the same sense that programs check their explicit parameters for errors. If the system provides a primitive that converts a tilde name into the absolute name of the tree associated with it in the current environment,

a privileged program could simply refuse to execute if the invoker's environment did not include the trees needed by the program. This approach, however, still requires embedding location dependent names in privileged programs.

Fortunately, we believe that in most cases the requirements privileged programs will place on their environments will be so simple that they can be verified without using absolute names. We suspect that most privileged programs will only need reliable access to two tilde trees – the tree in which the program itself is located and the tree containing the most basic system commands. Access to the tree in which the program is executed is guaranteed whenever the program is executing, because if that tree's name was not associated with the tree in the environment the program could not have been accessed. Access to the tree of basic system commands could be ensured either by having the system prevent users from changing the binding of this tree's name or by providing a primitive to determine that the current binding of the command tree's name was the standard binding. We hope to verify that these mechanisms would address the problem with privileged programs in our work with the prototype of the tilde naming system described in the next section.

7. A Prototype

A prototype of the naming mechanisms discussed in this paper is currently being developed[2]. The initial goal has been to provide an interface between the user and the Unix system that simulates Tilde naming. This interface is composed of a modified shell that recognizes tilde names and a library of procedures for manipulating files. The modified shell maintains a tilde environment that is inherited by processes it creates. The procedure library provides a Unix-like system call interface that interprets tilde names. A program must be re-linked with this library before it can use tilde names for files. Many of the standard system commands have already been re-linked on our system.

The current prototype uses Unix full path names as the absolute names of Tilde trees. A process can control its tilde environment by placing a list of absolute names into a Unix environment variable associated with the process. Sets of tilde names used to specify tilde environments are kept in files or other environment variables.

The current prototype does not fully implement the scheme described in this paper. In particular, whereas tilde trees in our scheme are logically independent, the use of full path names as absolute names allows users to specify tilde environments containing trees that actually overlap. Current plans call for a second implementation in which our scheme is actually incorporated into the system's kernel. In this version, we will be able to completely support the scheme we have described.

An area in which we hope the prototype will provide important insight is the design of the facilities for manipulating tilde environments. As stated above, these facilities will have a significant impact on users' perception of our system. With a working model we plan to experiment with several mechanisms for specifying and controlling tilde environments.

8. Summary

In this paper we have discussed the problems of location dependency associated with hierarchical file names and presented a new naming scheme intended to reduce these problems. Our scheme essentially uses a level of indirection in the name resolution process to separate location dependent aspects of naming from those closely related to a files function. The mechanisms we propose are carefully designed to make this indirection nearly transparent to the user, so that relative names are perceived as the primary file naming mechanism of the system.

We are currently implementing a prototype of this system, through which we expect to verify some of the assumptions concerning the use of file names made in our design and

to refine the design of the user interface to our mechanisms.

References

1. Brownbridge, D. R., L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!," *Software -- Practice and Experience* **12** pp. 1147-1162 (1982).
2. Comer, D. and R.E. Droms, "Tilde Trees in the Unix Environement," *Proceedings of the 1985 Winter Conference of the USENIX Association*, pp. 23-29 (January 1985).
3. IBM Corporation, "Operating System 360 Concepts and Facilities," pp. 598-646 in *Programming Systems and Languages*, ed. S. Rosen,McGraw-Hill, New York (1967).
4. Jones, Douglas W., "Improved Interpretation of Unix-like File Names Embedded in Data," *Communications of the ACM* 27(8) pp. 782-784 (August 1984).
5. Organick, E. I., *The Multics System: An Examination of its Structure.*, The MIT Press, Cambridge, Mass. (1972).
6. Popek, Gerald, Bruce Walker, Robert English, Charles Kline, and Greg Thiel, "The LOCUS Distributed Operating System," *Operating Systems Review* **17**(5) pp. 49-70 ACM, (Oct. 1983).
7. Ritchie, Dennis M. and Ken Thompson, "The Unix Time-Sharing System," *Communications of the ACM* **17**(7) pp. 365-375 (July 1974).
8. Tichy, Walter F. and Zuwang Ruan, "Towards a Distributed File System," *Proceedings of the Summer USENIX Conf.*, pp. 87-97 (June 1984).