Regular Paper

A Hierarchical-Keyword-based Naming Scheme in File Systems

Harumasa Tada,† Nobutoshi Todoroki,† Kazufumi Fukui† and Masahiro Higuchi††

Hierarchical naming has been used in file systems for the last several decades. As the number of files stored in file systems increases, the weakness of hierarchical naming is getting recognized. Some researchers have proposed hybrid naming schemes which introduce attribute-based naming into hierarchical naming. However, they are unsatisfactory or too complicated. In this paper, we propose a hybrid naming scheme, called HK (Hierarchical-Keyword-based) naming. In HK naming, a file is named by a set of keywords and keywords are organized hierarchically. It integrated hierarchical naming and attribute-based naming in a simple way. To show the practicality of our naming scheme, we implemented the prototype of the filename management system on UNIX file systems.

1. Introduction

With the advance of secondary storage devices, the number of files stored in a file system is rapidly increasing. It becomes a problem how to find a necessary file among a vast amount of files. On the other hand, files which contain non-text data, such as images or sounds, are popular nowadays. For such files, the content-based search, e.g., the search with the `grep` command in UNIX, is difficult or even impossible. Therefore, file organization by name is getting more important.

In traditional file systems, files are located in tree-structured directories. A user specifies a file by an explicit path name, i.e., an absolute path name or a relative path name from the current directory. Such a naming scheme is called *hierarchical naming* [1].

Hierarchical naming has some merits, e.g., familiarity, ease of implementation, etc. On the other hand, it has been pointed out by many researchers [1)~7)] that it is insufficient to organize a large number of objects. Because of this weakness, hierarchical naming is rarely used in database systems. In most of database systems, each data item is associated with some attributes instead of a path name. A user retrieves data items by queries [8]. The set of attributes of a data item can be regarded as its name and this type of naming is called *attribute-based naming*.

Some researchers introduced attribute-based naming into file systems [1)~3)]. Most of their approaches are hybrid naming scheme between hierarchical naming and attribute-based naming. In these approaches, integration of two naming schemes is rather awkward and resulting naming schemes are not necessarily easy to use. We consider that the problem lies in the metaphor for directories. A directory is a special file which contains information of other files. In hierarchical naming, directories are regarded as containers of files—they are represented as folders in most GUI systems. The problem of previous approaches is that they tried to introduce attributes while they preserve directories as file containers. In our opinion, the notion of file containers conflicts with file attributes and their integration results in mere composition of orthogonal name spaces.

We solved the problem by discarding the notion of file containers. The main components of our naming scheme are keywords. Instead of putting files in directories, we attach keywords to the files. Hierarchical structure is used for organizing keywords, not files.

We call our naming scheme *HK* (*Hierarchical-Keyword-based*) *naming*. In HK naming, a file is given a filename and a set of keywords. A user specifies files by an arbitrary ordered, possibly incomplete, list of keywords. A keyword is a string of characters such as `/image/photo` and `/article/sports/baseball`. Since the set of keywords form a keyword tree like a directory tree, such keywords are called *hierarchical keywords*. A user can find a keyword traversing the keyword tree.

In this paper, we show how files are organized and found in HK naming. To demonstrate the

† Graduate School of Engineering Science, Osaka University
†† School of Science and Engineering, Kinki University

practicality of HK naming, we implemented a prototype of a name management system based on HK naming on UNIX file systems. In our prototype, each hierarchical keyword is associated with a real directory. If a hierarchical keyword `/image/photo/child` exists, there is a directory `/image/photo/child`. It enables easy implementation and brings compatibility with usual UNIX file systems.

The rest of this paper is organized as follows. In Section 2, we describe advantages of hierarchical naming and attribute-based naming and review previously proposed hybrid naming schemes. Our naming scheme is presented in Section 3. Section 4 describes some issues of the implementation of HK naming on UNIX systems and our prototype system. The conclusion appears in Section 5.

2. Hierarchical Naming and Attribute-based Naming

2.1 Naming Schemes

We define a *name* as a string of characters which is used to specify a file or a set of files. In usual UNIX file systems, both absolute path names and relative path names are names. A *filename* is a string of characters, such as `main.tex` and `fig.jpg`, which is assigned to a file so that users can identify it. In this paper, we distinguish between names and filenames.

A *naming scheme* consists of syntactic representation and semantic interpretation of names. A set of names complying with a naming scheme forms a *name space*[9].

In traditional file systems, files are organized in tree-structured directories. A file (possibly a directory) has a filename which is unique in the directory which stores it. A name is defined as a path name which is a sequence of filenames. A path name describes where the file is located in the directory tree. The set of these names forms a hierarchical name space. This naming scheme is called *hierarchical naming*[1].

In database systems, on the other hand, tree-structured directories do not exist. Each data item is associated with some attributes. Each attribute is a pair of an attribute-name and a value such as "$author : tada$". A user retrieves data items by queries such as $(author = tada) \wedge (type = text)$. In this case, a query can be considered as a name which specifies a set of data items. The set of such names forms a kind of non-hierarchical name space. Such a naming scheme is called *attribute-based naming*[1].

Keyword-based naming is a special case of attribute-based naming. Each object is associated with several keywords. A keyword is an attribute of the object but it does not have an attribute-name. A name is a list of keywords such as "`tada,text,report`". Keyword-based naming inherits most advantages of attribute-based naming. We adopted keyword-based naming as the basis of our naming scheme.

2.2 Advantages of Hierarchical Naming

Hierarchical naming and attribute-based naming have their own advantages. In Ref. 1), advantages of hierarchical naming are mentioned. They are summarized as follows.

Analogy with Real World We can explain hierarchical structure by analogy with traditional paper-based information management in which a paper (file) is put in a folder (directory) which is stored in a cabinet (parent directory). A user can understand the structure of the name space intuitively.

Name Scope Each directory forms a self-contained naming context. The interpretation of a component of a name is determined solely by the context in which it occurs, as specified by the previous part of the name. Therefore, the size of the search space in name resolution is small. It enables efficient implementation of the name management system.

Sense of Place The current directory enables users to specify files using relative path names. It is useful to specify files concisely. Moreover, it gives users the sense of place. The current directory is recognized as the directory where a user is visiting.

Navigation When a user does not know the name of the file he needs, he can find it by traversing the directory tree. When a user lists the contents of a directory, the contents of subdirectories are hidden. This is the essence of the navigational nature of hierarchical naming. If files are well-organized in the directory tree, he can reach the file easily. In attribute-based naming, since attributes are not organized, it is difficult to recall which attributes are associated to the file.

In addition to above advantages, we consider that the following is important.

Organization of Directories In attribute-based naming, files are organized by at-

tributes. However, attributes are not organized. If a file is attached an attribute such as "*theme* : *horse*", there are no information about "horse" itself. In hierarchical naming, directories are organized hierarchically as well as files. For example, a directory name `/animal/herbivore/horse` represents that the horse is a herbivore which is a kind of animal.

2.3 Advantages of Attribute-based Naming

Advantages of attribute-based naming mentioned in Ref. 1) are as follows.

Independence of Attributes In hierarchical naming, file characteristics must be arranged in fixed order even if they are logically independent. Sometimes it brings inconveniences. For example, suppose that directory *monochrome* is a subdirectory of */home/tada/photo/child*. Though all photos of children can be listed, there is no straightforward way to list the set of all monochrome photos. In attribute-based naming, all attributes are handled independently.

Detailed Classification A very long path name such as `/home/tada/images/photograph/landscape/beach/summer/yacht/child.jpg` is inconvenient in hierarchical naming. To specify a file, a user must visit the directory which stores the file or describe its path name. In attribute-based naming, on the other hand, only a part of attributes are required to specify files. Hence any characteristics can be attached to files as attributes. It enables detailed classification of files.

In addition, we point out following ones.

Rapid Finding of Files Tree structure of hierarchical naming encourages navigation when a user is uncertain of a path name of a file. On the other hand, if a user remembers some attributes about the file, it is troublesome to traverse the directory tree. In attribute-based naming, he can use attributes he remembers to specify a set of likely files. In this case, he can find a file more rapidly than in hierarchical naming.

Appropriate Organization In hierarchical naming, there is a case in which files cannot be organized appropriately. For example, suppose that two directories */home/tada/photo/dog/* and */home/tada/photo/child/* exist. If there exists a photograph of children playing with a dog, which directory should it be located in? In this case, it is difficult to store the file appropriately. It is true that a user can make the file accessible from both directories using (symbolic or hard) links. However, management of such links is troublesome because each link is independent, i.e., it knows nothing about other links of the file. In order to remove a file, for example, a user should find and remove its all links by himself. In attribute-based naming, on the other hand, any attributes can be attached freely to the file. It enables to organize files appropriately.

Flexible Organization In hierarchical naming, if a file is moved from a directory to its subdirectory for more detailed classification, its path name will change. This causes a problem especially when files are shared by two or more persons. In attribute-based naming, even if new attributes were added to files, the files can be specified as before.

2.4 Hybrid Naming Schemes

Some researchers have tried to integrate hierarchical naming with attribute-based naming.

Semantic File System (SFS) [2] extends usual tree-structured file system to provide attribute-based access to files. In SFS, files are stored in usual tree-structured directories. Moreover, they have some attributes and can be accessed by queries. Attributes are automatically extracted from files by programs called *transducers*. SFS achieves syntactic compatibility with existing path names by introducing the concept of a *virtual directory*. Virtual directory names are interpreted as queries. For example, a query such as $(author = tada) \wedge (category = paper)$ is represented by the path name like `/author:/tada/category:/paper`. Queries of SFS are limited to conjunctions of atomic expressions like "*attribute* = *value*". The main feature of SFS is that it enables attribute-based access through native directory commands such as `ls` and `cd`. The drawback is that each file characteristic should be represented as either a directory name in a hierarchy or an attribute in a flat name space. For example, consider a file which contains a photograph of a mouse on a table. Now we note the characteristic "mouse". If "mouse" is represented as an attribute such as "*subject* : *mouse*", there is no way to distinguish between a small ro-

dent and a popular pointing device. If "mouse" is represented as a directory name such as `/photo/animal/mouse/furniture/table`, we can recognize that the file contains a photograph of a small creature. On the other hand, some fixed order is imposed on "animal" and "furniture" which are logically independent. Though SFS provides a unified syntax, it merely provides a composition of two orthogonal name spaces.

In Prospero File System [3], each file has some attributes like SFS. Prospero allows each user to have his own tree-structured name space which is called a *virtual file system*. The content of a directory is a collection of links. A link maps a single component of a name to a file or a directory. Each link may have an associated function, called a *filter*, which yields a virtual directory. A filter corresponds to a query in SFS. However, a filter is an arbitrary program written by users. It enables more flexible construction of virtual directory than SFS whose queries are conjunctions of atomic expressions. Though Prospero provides more flexibility than SFS, it is also a composition of two orthogonal name spaces. The cost of writing filters is the major drawback of Prospero.

Sechrest, et al. tried to integrate hierarchical naming with attribute-based naming using rule-based framework in their multi-structured naming [1]. They consider component names in a path name as attributes attached to a file with a set of constraints upon their use in names, e.g. they should be written in fixed order. In hierarchical naming, these constraints are implicit in the hierarchical structure of a name space. In multi-structured naming, on the other hand, a user can express the constraints that determine name space structure as explicit rules. For example, consider the name hierarchy shown in **Fig. 1**. Using scope rules, it is possible to specify that `male` and `child` do not introduce a new naming context. In this case, the names `/photo/portrait/male/child/cry.jpg` and `/photo/portrait/child/male/cry.jpg` refer to the same file. In addition to scope rules, there are some types of rules, i.e., implicit value rules, aliasing rules, etc. The details of the way to specify such rules is not described in Ref. 1). It is true that multi-structured naming provides enough flexibility. However, it seems to be too complicated for users to use. We consider that the user interface is an important and difficult issue of multi-structured naming.

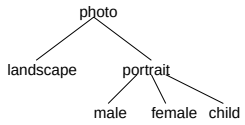


Fig. 1 The name hierarchy.

In multi-structured naming, the set of names forms a kind of name tree as usual hierarchical naming. Unlike hierarchical naming, constraints of hierarchical structure of the tree can be arbitrarily relaxed by users. It is a problem what commands should be provided to traverse such a name tree. It also seems to be difficult for users to describe appropriate rules for each attribute. In Ref. 1), only the framework is described and the user interface is not mentioned.

3. HK Naming

Existing hybrid naming schemes have shortcomings as mentioned above. All of them cling the notion that directories, whether real or virtual, are containers of files. We feel that it is not reasonable to use nested file containers and attributes together. To solve this problem, we consider that the notion of file containers should be discarded. Thus we propose another hybrid naming scheme.

We chose keyword-based naming as the basis of our naming scheme. It provides great flexibility in finding and organizing files. Nevertheless, we can point out two drawbacks of keyword-based naming. First, keywords are not organized, e.g., we cannot distinguish between the dining "table" and the multiplication "table". Second, keyword navigation is not provided. If we cannot recall what keywords are attached to the file which we needed, there is no way to navigate us to such keywords. As mentioned in Section 2.2, hierarchical structure provides these features. This is why we introduced hierarchical structure to organize keywords. We call our naming scheme *HK* (*Hierarchical-Keyword-based*) *naming*.

3.1 Hierarchical Keywords

In HK naming, a file is given a filename and attached some *hierarchical keywords*. A *component keyword* is a nonnull string of characters. A hierarchical keyword includes one or more component keywords k_1, k_2, $\cdots$, k_n and is denoted as $/k_1/k_2/\cdots/k_n$. The order of component keywords in a hierarchical keyword is meaningful, e.g., `/people/child` and `/child/people` are regarded as distinct

ones. If $/k_1/k_2/\cdots/k_n$ is a hierarchical keyword, $/k_1/k_2/\cdots/k_{n-1}$ is also a hierarchical keyword. In this case, $/k_1/k_2/\cdots/k_{n-1}$ is a *parent* of $/k_1/k_2/\cdots/k_n$ and $/k_1/k_2/\cdots/k_n$ is a *child* of $/k_1/k_2/\cdots/k_{n-1}$. The *ancestor-descendant relation* is the transitive-reflexive closure of the parent-child relation. The set of hierarchical keywords forms a hierarchical structure which we call the *keyword tree*.

3.2 Names

A file is given a filename and attached some hierarchical keywords. A name is a comma-separated list of hierarchical keywords, possibly followed by a filename after "`//`". In names, the order of hierarchical keywords is meaningless. For example, the meanings of two names "`/people/child,/animal/dog//f1.jpg`" and "`/animal/dog,/people/child//f1.jpg`" are the same.

A *full name* of a file is a name which consists of a filename and all hierarchical keywords attached to the file. Since hierarchical keywords can be listed in any order, the full name of a file is not unique.

A hierarchical keyword *matches* all descendants of it. For example, `/animal` matches `/animal`, `/animal/cat`, `/animal/dog/poodle`, etc.

A name N matches a file f if every hierarchical keyword in N matches at least one of hierarchical keywords attached to f and the filename in N is the same as f's filename. If N does not include a filename, N matches all files which the keyword list in N matches. A name specifies all files which it matches. If a name which is an argument for a command matches multiple files, all of them are passed to the command as the case that "`*`" is used in the UNIX shell.

When a user describe a name in shells, he can use meta characters such as "`*`" like in usual UNIX shells. In addition, we introduce another meta character "`@`" which matches any sequences of component keywords. For example, `/image/@/dog` matches `/image/dog`, `/image/photo/animal/dog`, etc. More precisely, a string "`/@`" can be replaced with a null string or a single slash ("`/`") followed by any slash-separated sequence of one or more component keywords. "`/@`" at the last of a hierarchical keyword such as `/dog/@` has no effect because a hierarchical keyword matches its all descendants by default. `/@/dog` matches any hierarchical keywords which includes a component keyword `dog`. A name can include multiple "`@`"s such as `/@/dog/@/white`.

Table 1 Sample files.

file	full name
f_1	`/sports//fig.jpg`
f_2	`/sports/skate//fig.jpg`
f_3	`/woman,/sports//fig.jpg`

3.3 Uniqueness

In file systems, it is required that every file should be specified uniquely. In hierarchical naming, this uniqueness is guaranteed by absolute path names. In HK naming, it is guaranteed by full names. It is prohibited that multiple files have the same full name. However, there are cases when the full name of a file cannot be specify the file uniquely. For example, consider three files shown in **Table 1**. f_1's full name "`/sports//fig.jpg`" cannot be specify f_1 uniquely because it also matches f_2 and f_3.

In order to solve the problem, we introduced notations using "`$`" and "`!`". Hierarchical keywords followed by "`$`" does not match their children. For example, `/animal$` matches `/animal` only, while `/animal` matches `/animal/cat`, `/animal/dog/poodle`, etc. The notation using "`!`" is used to exclude unrelated keywords. A hierarchical keyword k is called *unrelated* to a name n when k is not a descendant of any hierarchical keyword in n. A name whose keyword list ends with "`!`" does not match any file which is attached a keyword unrelated to the name. For example, the name "`/animal,/image!//fig.jpg`" does not match the file whose full name is "`/animal/dog,/image,/child//fig.jpg`" because it is attached `/child` which is unrelated to the name. Using these notations, a user can always specify any file uniquely. f_1 in Table 1 is specified with "`/sports$!//fig.jpg`".

3.4 Current Kw-List

In hierarchical naming, a user can use a relative path name from the current directory. In HK naming, we introduce the *current kw-list* which is similar to the current directory in hierarchical naming. It is a list of hierarchical keywords which specifies the set of files which a user is currently interested in. The set is called the *current file set*.

The current kw-list enables specification of files with filenames. When the current kw-list is "`/image/photo,/sports/baseball`", a filename `fig.jpg` is interpreted as "`/image/photo,/sports/baseball//fig.jpg`". To add the current kw-list to names which include hierar-

chical keywords, "," is placed at the head of them, e.g., ",/player/pitcher//fig.jpg".

3.5 File Organization and Search

Here we describe how to organize and find files using HK naming.

Files are organized by attaching (or detaching) hierarchical keywords to them. A hierarchical keyword should be created before it is attached to files. Hierarchical keywords which are no longer needed can be removed. For these purposes, we introduce following commands.

- `mkkw` *keyword...* (MaKe KeyWord)
  `mkkw` creates a new hierarchical keyword.
- `rmkw` *keyword...* (ReMove KeyWord)
  `rmkw` removes a hierarchical keyword.
- `atkw` *list name...* (AtTach KeyWords)
  `atkw` attaches hierarchical keywords in *list* to files specified by *name*.
- `dtkw` *list name...* (DeTach KeyWords)
  `dtkw` detaches hierarchical keywords in *list* from files specified by *name*.

When a user does not know the accurate name of the file which he needs, he searches the file by changing the current kw-list. Commands used to find files are summarized as follows.

- `akl` *list* (Append to Kw-List)
  `akl` appends hierarchical keywords in *list* to the current kw-list.
- `rkl` *list* (Remove from Kw-List)
  `rkl` removes hierarchical keywords in *list* from the current kw-list.
- `lsf` [*name*] (LiSt Files)
  `lsf` lists filenames of files specified by *name*. If *name* is omitted, the current kw-list is used as an argument.
- `pkl` (Print Kw-List)
  `pkl` shows the current kw-list.

When a user cannot recall what keywords are attached to the file which he needed, he searches a keyword traversing the keyword tree. The keyword tree can be traversed like a directory tree. The *current keyword* indicates the current place in the keyword tree. There are commands `ckw`, `lskw` and `pkw` which corresponds to `cd`, `ls` and `pwd` respectively. Commands for traversal of the keyword tree are summarized as follows.

- `ckw` *keyword* (Change KeyWord)
  `ckw` changes the current keyword to *keyword*.
- `lskw` [*keyword*] (LiSt KeyWords)
  `lskw` lists children of *keyword*. If *keyword* is omitted, the current keyword is used as an argument.
- `pkw` (Print KeyWord)
  `pkw` shows the current keyword.

```
% akl /@/koala,/image/photo                    (1)
Keyword "/image/photo" does not exist
% pkl                                          (2)
/@/koala
% lsf                                          (3)
koala-life.ps  koala.txt(3)
koala-map.gif  voice-koala.mp3
% rkl /@/koala                                 (4)
% pkw                                          (5)
/
% lskw                                         (6)
animal  image  sound  text
% akl /animal,/image                           (7)
% lsf                                          (8)
koala.jpg      lion.jpg   zebra.jpg
koala-map.gif  panda.jpg
% xv koala.jpg                                 (9)
[ File koala.jpg is displayed. ]
% atkw /animal/koala koala.jpg                 (10)
% ckw image                                    (11)
% lskw                                         (12)
map  painting
% mkkw photo                                   (13)
% atkw photo *.jpg                             (14)
% lsf --full koala.jpg                         (15)
/animal/koala,/image/photo//koala.jpg
```

Fig. 2 The sample session.

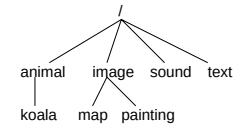


Fig. 3 The keyword tree.

We did not provide a command to create a file explicitly. A new file is created automatically when a user specified a nonexistent name as the output file of a command or an application.

Fig. 2 shows an example of file searching and organizing. The goal of the user is to find a file which contains a photograph of a koala. Suppose that the initial keyword tree is like **Fig. 3**. At first, he tried to append keywords `/@/koala` and `/image/photo` to the current kw-list (in line 1). Since `/image/photo` does not exist in the keyword tree, only `/@/koala` was appended to the current kw-list (in line 2). Though he listed files which matches `/@/koala`, the file he required was not listed (in line 3)☆. Then he deleted `/animal/koala` from the current kw-list (in line 4) and searched other keywords

☆ The number after `koala.txt` shows that there are three files which have the same filename `koala.txt`.

(in line 5 and 6). He appended `/animal` and `/image` to the current kw-list (in line 7) and found the file (in line 8). He displayed the file using `xv` (in line 9). For the convenience of future search, he attached hierarchical keywords to files (in line 10 and 14). Since `/image/photo` did not exist in the keyword tree, he created it beforehand (from line 11 to 13). The "`--full`" option of `lsf` is to list full names instead of filenames (in line 15).

3.6 Properties of HK Naming

HK naming integrated hierarchical naming and attribute-based naming in a relatively simple way. Since it is basically an attribute-based naming, it inherits all advantages of attribute-based naming mentioned in Section 2.3. In addition, it contains organizational and navigational features of hierarchical naming. On the other hand, some advantages of hierarchical naming are lost.

First, since HK naming discarded the notion of file containers, the analogy with real world are found no longer. In other hybrid naming schemes, this analogy is preserved. In order to exploit attributes, however, a user must understand attribute-based search to which he is not familiar in real world.

Second, the sense of place is lost. However, the current kw-list enables a user to specify the naming context in which he is interested. It is a generalization of the current directory in hierarchical naming. Hierarchical keywords can be searched in the same way as the directory search.

Third, the name resolution of HK naming costs more than that of hierarchical naming because of lack of the name scope. This problem has been tackled in area of database systems and many optimization methods based on indexing have been proposed, for example, Ref. 10)∼12). We expect that such indexing can be applied to improve name resolution time. A rapid advance of computer hardware also diminishes the effect of the size of the search space.

3.7 Name Spaces

The set of names attached to files forms a name space. In hierarchical naming, one name space (a directory tree) is shared by all users. In general, a user cannot modify the name of other users' files. In HK naming, each user has his own name space. Usually other users' name spaces are invisible. A user is free to attach and detach any hierarchical keyword to any file in his name space. He cannot modify other users' name spaces. Consider two users A and B. A can attach `/animal/dog` to B's file `/photo/child//img.jpg` in A's name space without any permission of B. After A attached it, B cannot specify the file with `/animal/dog//img.jpg` because A's name space is invisible to B. If necessary, a user can include (or exclude) other users' name spaces with explict declaration. A user can set the permission to his name space to limit inclusion by other users. `root` is also a user and has his own name space. Including the `root`'s name space, a user can access shared files such as shared libraries.

4. Implementation on UNIX Systems

In this section, we mention some issues which will arise when HK naming is implemented on UNIX environment and describe our prototype of the name management system.

4.1 System Calls and Libraries

UNIX provides system calls for low-level file I/O, i.e., `open`, `read`, `write` and `close`☆. Since HK naming uses lists of keywords instead of path names, these system calls should be modified.

Before a file is accessed, `open` is used to open (or possibly create) the file and bind it to a file descriptor. Since `read`, `write` and `close` access the file using the file descriptor instead of the path name, they need not to be modified to adapt to HK naming. That is, only `open` should be modified to accept names which include hierarchical keywords.

Existing `open` always takes a path name of a single file, while the modified `open` takes a name which may specify multiple files. For compatibility with existing applications, `open` should always return a single file. For this reason, the modified `open` should behave as follows. When a name which includes no filenames, such as "`/animal/mouse,/image/photo`", is given, `open` creates a temporary directory which contains all files which matches the name and returns the file descriptor of the directory. When a name which includes a filename, such as "`/animal/mouse,/image/photo//f1.jpg`", is given, `open` returns the file descriptor of the file which matches the name. If multiple files match the name, one of them is arbitrarily chosen.

☆ `creat` which is used to create files is made obsolete by `open`.

This adaptation of `open` system call involves modification of OS kernels and its efficient implementation is a difficult issue.

Most applications do not call above system calls directly. They access files using functions in the `stdio` library such as `fopen`, `fscanf`, `fclose`, etc. Since only `fopen` takes a path name as an argument, it is possible to adapt `stdio` library to HK naming by modifying `fopen`.

4.2 The Search Path for Commands

In UNIX, command names are filenames of executable files. In usual UNIX system, the search path for commands is a colon-separated list of directory names like "`/usr/X11R6/bin:/usr/local/bin:/usr/bin:/bin`". Path names of all directories which stores executable files should appear in the search path. In HK naming, the search path is a colon-separated list of names. For example, "`/local,/script:/shared,/binary`" means that files which "`/local,/script`" matches are searched before those which "`/shared,/binary`" matches. If the same hierarchical keyword such as `/executable` (or its descendant) is attached to all executable files, a very simple search path like "`/executable`" becomes possible. On the other hand, there is a case that the given command name matches multiple files simultaneously. For example, a command name `test` matches "`/local,/executable/script//test`" and "`/shared,/executable/binary//test`". In this case, any of them are not executed immediately and a user should specify the file he want to execute using additional keywords. To avoid this, filenames of executable files should be unique. Of course, adding other names to the search path like "`/local,/executable:/executable`", it is possible to specify the search order.

4.3 File Selection in Applications

There are many interactive applications which allow a user to select a file to open. For example, Emacs is one of such applications. Generally, Emacs users do not use shells to search a file to edit.

In order to adapt such applications to HK naming, modification of system calls mentioned in Section 4.1 is not enough. Each application itself needs to be modified to select a file using HK naming. In the case of Emacs, this file selection mechanism is easily implemented using Emacs Lisp. For many applications, however, this adaptation involves source code modification and recompile.

Some applications, e.g., tgif, xv, ghostview, etc., has their own GUI-based file selectors based on hierarchical naming. For such applications, a common file selector based on HK naming is desirable. We recognize that it is difficult for GUI in HK naming to use a real world metaphor. Development of an appropriate GUI is one of our future works.

4.4 Prototype on UNIX File System

To show the practicality of our naming scheme, we implemented a prototype of the name management system using HK naming on UNIX file systems.

As mentioned in Section 4.1, the true implementation of HK naming involves modification of OS kernels. We implemented the prototype system without modifying the underlying operating system. Though it enables to easily apply to existing UNIX environment, it is unfavorable from the viewpoint of performance. We are now studying the efficient implementation of HK naming in operating system level.

4.4.1 Architecture

The prototype system runs on an ordinary UNIX shell. A user describes names as command arguments. The system receives commands and arguments in place of the shell. Then it translates arguments into absolute path names and passes them with commands to the underlying shell. The prototype system is written in Perl and the program size is about 500 lines.

4.4.2 Management of Keywords

In our prototype system, a keyword tree is implemented as a directory tree. That is, each hierarchical keyword is associated with a real directory. If a hierarchical keyword `/image/photo/child` exists, there is a directory `/image/photo/child`. Each hierarchical keyword has a unique ID called a *keyword ID*. Keyword IDs are managed using a table called a *keyword table*. An example is shown in **Table 2**.

We show how hierarchical keywords are man-

Table 2 The keyword table.

ID	hierarchical keyword
1	`/image`
2	`/device`
3	`/animal`
4	`/animal/mouse`
5	`/image/photo`
6	`/device/mouse`

aged in our prototype system by an example. Suppose that the keyword table is as Table 2.

Consider a file f_1 whose path name is `/image/photo/fig.jpg`. Though f_1 is placed in the directory */image/photo*, it does not mean that f_1 is attached a hierarchical keyword `/image/photo`. To attach a keyword `/image/photo` to f_1, a symbolic link to f_1 is placed in the directory */image/photo*. In the rest of this paper, we refer to such a link as *file-link*. In this case, the name of the file-link is "`.fig.jpg#5#`". The number "5" is the keyword ID of `/image/photo` (see Table 2). When a new keyword `/animal/mouse` is attached to f_1, a new file-link "`.fig.jpg#4,5#`" is placed in the directory */animal/mouse*. "`.fig.jpg#5#`" in */image/photo* is renamed "`.fig.jpg#4,5#`". In this way, for any file f, f's file-link exists in every directory which corresponds to f's hierarchical keyword. All f's file-links have the same name which includes all IDs of f's hierarchical keywords. Because of this, "`--full`" option of `lsf` (mentioned in Section 3.5) is easily implemented, i.e., it just converts keyword IDs in file-link to hierarchical keywords referring the keyword table.

4.4.3 Temporary Directory

In our prototype system, files are handled as file-links. It causes a problem with some existing applications which limit filenames of input files. For example, filenames of input files of LaTeX should end with the extension "`.tex`". LaTeX does not accept file-links like "`.main.tex#14,25#`" as inputs. In order to deal with such cases, file-links are not passed to commands directly. A file-link such as "`.main.tex#14,25#`" is copied to the temporary directory with its filename "`main.tex`" and the copy is passed to the command.

4.4.4 Name Resolution

A conversion from a name of files into file entities is called *name resolution*. Suppose that the keyword table is as Table 2 and a name "`/@/animal,/image`" is specified by a user. Our prototype system convert it to a set of files as follows.

(1) The system chooses a hierarchical keyword in the name. Suppose that `/@/animal` was chosen. The system refers the keyword table to find hierarchical keywords which `/@/animal` matches. In this case, only `/animal` was found.
(2) The system opens all directories in the directory subtree rooted */animal* and makes a set L of all file-links in them. At this time, L contains all files that any descendant of `/animal` is attached to. If `/@/animal` matches multiple keywords, this step is iterated for each of them and L becomes the union.
(3) Referring the keyword table, the system makes a set of keyword IDs of `/image`'s all descendants. Then file-links which include no IDs in the set are excluded from L. If the name includes more keywords, this step is iterated for each of the rest of keywords in the name. If the name includes a filename such as `fig.jpg`, each file-link in L is checked whether its filename is `fig.jpg`.
(4) For the reason mentioned in Section 4.4.3, all file-links in the resulting L are copied to the temporary directory such as */temp/14340* (14340 is the process ID of the undelying shell).
(5) Finally, the path name "`/temp/14340/*`" is passed to the underlying shell.

4.4.5 Limitation of Prototype System

As mentioned in Section 4.3, many application need to be modified to use with our prototype system. The current system is severely limited in usage of existing applications. Another limitation of the prototype system is concerned with command arguments. If a command has names of files as arguments, they are translated into UNIX path names before execution of the command. Sometimes it causes a problem because not all command arguments specify files. For example, the argument of `man` command is not a name of a file but a command name. Since which argument specifies files is determined by each application, there is no way for the prototype system to know it. The prototype system regards arguments which starts with "`/`" or "`,`" (see Section 3.4) as names of files, translates them into path names and passes the path names to the command. Other arguments are passed to the command as it is. Therefore, each argument which starts with "`/`" or "`,`" and is not a name of files should be quoted.

4.4.6 Experiments

Using our prototype, we organized all files stored in home directories of 15 users. Absolute path names are converted into full names. For example, an absolute path name `/home/tada/TeX/macros/abc.sty` is converted to a full name "`/home/tada/TeX/macros//abc.sty`". It can be specified by names such as "`/home/tada//abc.sty`", "`/@/macros,/@/TeX//abc.sty`", "`/home/@/TeX//abc.sty`", etc. Though multiple hierarchical keywords can be

Table 3 Response time and number of files.

Name	Time (sec.)	Files
/home/higuchi	10.0	15,173
/home/higuchi/Mail	3.9	6,458
/home/tada/TA/database	0.3	13
/@/database	5.5	3,751
/@/Bthesis	4.8	4,174
/@/error	0.4	30
/@/Bthesis,/@/database	3.5	6
/@/Bthesis,/@/error	0.4	15

assigned to a file, it is not performed in the experiment.

The total number of files is 101,863. There are 2,582 hierarchical keywords. There are cases that the same component keywords may appear in different places in the keyword tree. For example `bin` appears in `/home/tada/bin`, `/home/higuchi/bin`, etc. There are 1,233 different component keywords.

4.4.7 Performance

As mentioned in Section 3.5, the `lsf` command lists filenames of files specified with an argument. We measured the response time for `lsf`. It depends on the name given as the argument. In the experiment, files and the keyword table are stored in the same host running Linux 2.0.36 (CPU: Pentium III 600 MHz, Memory: 128 MB). In the experiment, `lsf` for all hierarchical keywords are consecutively executed. It is probable that the some directories searched in name resolution are stored in caches. Without directory caches, the performance shown below is expected to decline.

Table 3 shows response times of `lsf` and the numbers of specified files for some names. For names which consist of only one hierarchical keyword, the number of specified files affects to the response time. Since "`/home/higuchi`" specifies so many files, its response time is very long. The results of "`/home/tada/TA/database`" and "`/@/database`" show that the use of "`@`" notation may cause long response times. In case of names including multiple hierarchical keywords, the number of specified files does not necessarily affect to the response time. For example, "`/@/Bthesis,/@/database`", which specifies only 6 files, takes long time because both "`/@/Bthesis`" and "`/@/database`" specifies many files. In this way, names with "`@`" notation tend to take longer time to execute `lsf`.

The reason why the response time of "`lsf /@/Bthesis,/@/database`" is shorter than that of "`lsf /@/Bthesis`" or "`lsf /@/database`" is as follows. For simplicity, we denote "a set of files which a name N matches" as $[N]$. In order to make [`/@/Bthesis,/@/database`], the prototype system does not make [`/@/Bthesis`] and [`/@/database`] respectively. Instead, it makes [`/@/Bthesis`] first and then files which "`/@/database`" matches are picked up from [`/@/Bthesis`] (Step (2) in Section 4.4.4). Thus the time to make [`/@/Bthesis,/@/database`] is shorter than the sum of the time to make [`/@/Bthesis`] and [`/@/database`]. The time to copy all file-links to the temporary directory (Step (4) in Section 4.4.4) seriously affects the execution time. This time depends on the number of file-links in the set. The number of file-links in [`/@/Bthesis,/@/database`] is much less than that of [`/@/Bthesis`] or [`/@/database`]. As a result, the overall execution time of "`lsf /@/Bthesis,/@/database`" is shorter than that of "`lsf /@/Bthesis`" or "`lsf /@/database`".

Table 4 Distribution of response time.

Time (sec.)	Frequency	Ratio (%)
$0 \sim 1$	1,147	93.0
$1 \sim 2$	61	4.9
$2 \sim 5$	14	1.1
$5 \sim$	11	0.9

For any component keyword k, we executed "`lsf /@/`k" and measured the response time. **Table 4** shows the distribution of the response times. As shown in the table, for about 98% of all cases, `lsf` responds in less than 2 seconds. Though this result is owing to directory caches as mentioned above, it is generally acceptable in interactive use. That is, we can conclude that our prototype system is barely useful in relatively small-scaled environment as the example. In a larger environment which consists of more than a hundred users, its performance is expected to become impractical. For such environments, implementation which involves OS kernel modification seems to be essential.

5. Conclusion

Some researchers have proposed hybrid naming schemes between hierarchical naming and attribute-based naming. However, they are less flexible or difficult to use because two conflicting notions, file containers and file attributes, coexist in them. To solve the problem, we introduced the notion of hierarchical keywords in-

stead of file containers and proposed HK naming. We implemented a prototype of filename management system on a UNIX file system and showed that it is practical for a relatively small-scaled environment.

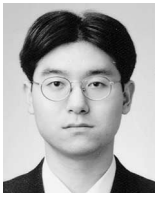
Our current prototype is designed for attaching more importance of ease of implementation than to efficiency of name resolution. We plan to study more efficient implementation of the name management system. The user interface for HK naming (including GUI) is another interesting topic.

References

1) Sechrest, S. and McClennen, M.: Blending Hierarchical and Attribute-Based File Naming, *Proc. 12th IEEE Intl. Conf. on Distributed Computing Systems*, Yokohama, Japan, IEEE, pp.572–580 (1992).
2) Gifford, D., Jouvelot, P., Sheldon, M. and O'Toole, J.: Semantic File Systems, *Proc. 13th ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, ACM, pp.16–25 (1991).
3) Neuman, B.: The Prospero File System: A Global File System Based on the Virtual System Model, *Proc. USENIX Workshop on File Systems* (1992).
4) Gopal, B. and Manber, U.: Integrating Content-Based Access Mechanisms with Hierarchical File Systems, *Proc. 3rd Symp. on Operating Systems Design and Implementation*, New Orleans, LA, USENIX, pp.265–278 (1999).
5) Bowman, M., Spasojevic, M. and Spector, A.: File System Support for Search, Technical Report, Transarc Corporation, Pittsburgh, PA (1994).
6) Freeman, E. and Gelernter, D.: Lifestreams: A Storage Model for Personal Data, *ACM SIGMOD Record*, Vol.25, No.1, pp.80–86 (1996).
7) Neufeld, G.: Descriptive Names in X.500, *Proc. SIGCOMM '89 Symposium, Communications, Architectures and Protocols*, Austin, Texas, ACM, ACM PRESS, pp.64–71 (1989).
8) Korth, H. and Silberschatz, A.: *Database System Concepts*, McGraw-Hill, New York, NY (1991).
9) Terry, D.: Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments, Technical Report CSL-85-1, Xerox Palo Alto Research Center (1985).
10) Faloutsos, C.: Access Methods for Text, *ACM Comput. Surv.*, Vol.17, No.1, pp.49–74 (1985).
11) Sacks-Davis, R., Kent, A. and Ramamohanarao, K.: Multikey Access Methods Based on Superimposed Coding Techniques, *ACM Trans. Database Syst.*, Vol.12, No.4, pp.655–696 (1987).
12) Moffat, A. and Zobel, J.: Self-Indexing Inverted Files for Fast Text Retrieval, *ACM Trans. Information Syst.*, Vol.14, No.4, pp.349–379 (1996).

(Received November 17, 2000)
(Accepted June 19, 2001)

Harumasa Tada received his B.E., M.E. and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1993, 1995 and 1998, respectively. He is currently a Research Assistant of Graduate School of Engineering Science, Osaka University. His current research interests are file naming schemes and distributed programming environment.



Nobutoshi Todoroki received his B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1999 and 2001, respectively.

Kazufumi Fukui received his B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1999 and 2001, respectively.

Masahiro Higuchi received the B.E., M.E. and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1983, 1985 and 1995, respectively. In 1985 $\sim$ 1990, he worked for Fujitsu Laboratories LTD. In 1991 he joined the faculty of Osaka University. In 1995 $\sim$ 1999, he was an Assistant Professor of Department of Information and Computer Sciences, Osaka University. Since 2000 he has been an Associate Professor of School of Science and Engineering, Kinki University. His current research interests include problems of distributed system design, and methods for protocol specification, testing and verification.