| 6.826 Principles of Computer Systems | Fall Semester, 1993 |
| --- | --- |

# Lecture 10: October 18, 1993

*Lecturer: Butler Lampson* *Scribe: Ekrem Soylemez*[1]

## 1 Overview

In this lecture, name related issues, atomicity, aliases, and links in the file system are discussed. It is the first of two discussing issues related to naming. The emphasis is on a weak specification, which can be very powerful. In this lecture provides an overview and specification of a vanilla file system.

## 2 What do we mean by name?

In the real world, it is not always true that a name always maps to a single object. It is unrealistic to expect a name like Mike to map to a single person. However, in the world of computers, it is usually true that a name always map either to one element or to nothing. We will assume that this is the case with our file system.

With this in mind, we can define name as follows:

```
TYPE
    N = STRING              % Name
    PN = SEQ[N]             % Path name
```

Note that PN is an abstraction of hierarchical name. Though they may use different separator to separate two names in the path name, many of the file naming schemes that we know are the implementations of this abstraction.(e.g. Unix, VMS, DOS). Furthermore, although file systems provide the canonical example of hierarchical names, they appear in many other situations. Some examples of hierarchical names outside of file systems are Internet addresses, IP addresses and page tables in virtual memory. Also note that the string need not be a string, but might be a number. This is the case, for example, in IP addresses.

## 3 What do we do with names?

Different uses of names have different requirements. However, the following list is common to many name spaces.

- Read: look it up and return the value. In the Unix file system, this corresponds to finding a file.

[1] These notes borrow some lines from the notes prepared by Yasuhiro Endo last fall, and Kah-Kay Sung the fall before that.

- Write: add a new name. This may not always be a possibility. In the Unix file system, this is creating a file.
- Enumerate: although this is often overlooked, it is very important. In the Unix file system, it is `ls`.
- Rename: change the name of a value. Like write, this is not always available. In the Unix file system, it is `mv`.

# 4 Specification

Clearly there must be some state in which names make sense. That is to say, names must map to some value, and this mapping function can be seen as the state. If we assume the state is much like the simple memory we discussed earlier this term, then a specification might be:

```
VAR d: PN -> V %V is the datapath.
```

Notice that the chief difference from the old simple memory is the address has become much different. The data is fairly similar. For example, in the Unix file system, it is the inode.

Given the new definition, the following specifications can be written:

```
Read(d, pn) -> V raises {Undefined} =
    <<d!pn => RET d(pn) [*] RAISE Undefined>>

Write(d, pn, v) =
    <<d(pn) := v>>

Remove(d, pn) =
    <<d := d{pn->}>>

Rename(d, pn1, pn2) =
    << DO pn | d!(pn1+pn) =>        % for each pn that is under pn1
          d(pn2+pn) := d(pn1+pn);   % associate pn2+pn with the value of pn1+pn
          d := d{pn1+pn->} OD>>     % undefine pn1+pn

Next(d, pn)->PN =                   % Instead of enumerate.
    << VAR pn', pn'' |
           (pn' > pn) /\            % Define an ordering on pn's.
           ~(EXISTS pn'' > pn /\ pn' > pn'')
           => RET pn' >>            % Return next higher one.
```

These specifications are somewhat coarse and unrealistic. For example, there is no obvious way to report a directory in this specification. Also, we need to introduce an invariant to make sure that if the path name has a value none of its predecessor has a value. But these specifications represent fairly solid idea of a file system that we can refine.

# 5 Implementation

One question which arises from this specification is how to implement it. The most straightforward way is simply recording the (PN, D) pairs. This has two major disadvantages: the data structure may become excessively large if the names can be long, and searching for a name must be done linearly. The first of these can be eliminated by removing similar prefixes between consecutive entries.

Another possible representation is a B-tree. This can have difficulties with renaming directories, however. Consider, for example, the following directory structure:

```
/a/b/c/1
/a/b/c/2
/a/b/c/......
/a/b/d/1
/a/b/d/2
/a/b/d/......
```

Now, renaming `/a/b` to `/a/x` will require making a change to every child of `/a/b`.

# 6 Object Oriented/Local Approach

Another way to implement the state is to view each level as an object. This allows the user to traverse the name space as a tree. This is the view normally thought of as file systems, although there is some issue as to whether the name space will actually be a tree, as there may be links to other directories, and possibly even cycles. (Both of these can be done in Unix with `ln` and `ln -s`.) Furthermore, the object oriented view solves the difficulties mentioned at the end of the specification.

We implement this idea by modifying the definition of `D` as follows:

```
TYPE
    D = N->(V+D)
```

Using this new definition, we rewrite the specification as follows:

```
Read(d, pn) -> V raises {Undefined} =
    VAR d', n | (d', n) := GetDir(d, pn);
    d'!n /\ d'(n) is  V => RET d(n) [*] RAISE Undefined

Write(d, pn, v) raises {Undefined} =
    VAR d', n | (d', n) := GetDir(d, pn);
    ~d'!n \/ d'(n) is V => d'(n) := v [*] RAISE Undefined

Mkdir(d, pn) raises {Undefined} =
    VAR d', n | (d', n) := GetDir(d, pn);
    ~d'!n => d'(n) := {} [*] RAISE Undefined
```

```
% finds d′ and n so that d′(n) = some value
GetDir(d, pn) -> (D, N) raises {Undefined} =
    pn = {} \/ ˜d!pn.head => RAISE Undefined
  [*] pn.size = 1 => RET(d, pn.head)
  [*] RET GetDir(d(pn.head), pn.tail))
```

One consequence of this idea is that it allows for decentralized implementations. In other words, each directory may have its own method of doing things. This idea is not very well understood in older name hierarchies. For example, the original reason for the @ symbol in Internet addresses is to tell the implementation that a different lookup method is needed, as this is a user rather than a machine. Similarly, in the original BSD Unix, a path name was `machine:/a/b/c`. The colon in this name provided the information that the section before it was a machine name, and needed to be treated differently. By contrast, the Andrew file system (AFS) allows a machine to be accessed as just another directory. The different lookup methods are transparent to the user.

# 7 Looking up names

With the object oriented approach, one can easily imagine starting lookups in different places, depending on the context. In Unix, a file can be accessed in many ways, including:

- `a` The file `a` in the current directory.
- `/a` The file `a` in the root directory.
- `lampson/a` The file `a` in the user lampson's home directory.
- `../a` The file `a` in the parent directory.

In DOS, there is another level of naming: the drive. For example `C:a` refers to the file `a` in the current directory on the C drive.

When designing and using a name space, there is always a tension between knowing the details of what is going on in your space, and the ability to explore the world. For example, yesterday you might have read a file on a machine somewhere on the Internet, and the machine might have been renamed by someone else today. However, this sort of behavior probably does not happen on your local hard drive. One possible solution to this is to have absolute addresses. The problem is that it is hard to deal with locating a specific name. The solution to this is to segment the name, and provide a forwarding server which keeps track of all the names regardless of how they have changed. With the exponential increase in RAM available for a relatively small cost, this idea is becoming increasingly practical.

# 8 Possible Augmentations to the Vanilla File System

The operations needed, and the significance of their semantics vary greatly with the task. For example, if one is browsing the Worldwide Web, the semantics of write are not very important. However, to RCS/SCCS, the semantics are almost the entire point. With this in mind, the following are some additional operations which may be available in some situations, and not in others.

## 8.1 Another `Enumerate`

Here is an `Enumerate` which just returns the set of all names in a directory. In some cases this may be totally inappropriate, as the set may be too large to be returned in one shot. In these cases, the `Next` operation would be more useful.

```
Enumerate(d, pn) -> SET N Raises{error} =
    << VAR d′, n| (d′, n) := GetDir(d, pn);
           RET {n′| d′(n) ! n′}>>
```

## 8.2 `Union`

Another sometimes useful operation is `Union`. This can be specified by

```
Union(d1, d2) -> D =
    RET lambda m -> (D + V) = d1!m => RET d1(m) [*] RET d2(m)
```

This is very similar to search paths in the Unix file system.

## 8.3 `Mount`

Another possible augmentation to the file system is `Mount`, which overrides the definition of a directory in an original structure with a new subtree.

```
Mount(d1, m, d2) -> D =
    RET lambda m′ -> (D + V) = (m = m′) => RET d1(m) [*] RET d2(m)
```

## 8.4 Aliases and Links

Aliases and links both allow an object to have more than one name associated with it. That is, if `/b/y` is aliased to `/a/x`, `write(D, /b/y, V)` also changes the value of `/a/x`. The difference between aliases and links is that an alias directly references the shared object while a link references the object through its existing path name. Thus, if `/b/y` is linked to `/a/x`, and `/a/x` is removed later, then `/b/y` becomes a dangling pointer. But if `/b/y` is aliased to `/a/x`, then `/b/y` will make sense even if `/a/x` is removed. Aliases are similar to "hard links" in the Unix file system, while links are similar to "soft links." We change the definition of D to allow links:

```
TYPE
  i = integer
  link = pn
  D = N->(V+i+link)       % i is an offset into a sequence of D,
```

### 8.5 Virtual Nodes

One problem is that not all “files” are really that. For example, devices, and remote directories can be viewed as entries in the local directory structure. With the object oriented methods, these entries can trigger a virtual node `vnode` switch in the kernel. This makes a call to an appropriate method which handles the implementation of the virtual node.

This allows the idea of the file system to be extended almost arbitrarily. For example, in *Plan 9*, almost everything is treated as a file. This includes all devices and memory.

## 9 Example Application of Hierarchical File System - Relational Database

One can imagine implementing a relational database using a hierarchical file system. Given the primary key of records, one can construct the database as follows:

```
SSN(Primary Key)        NAME        Salary    ...
123-456-7891            XXX         25000     ...
234-567-8901            YYY         35000     ...
...                     ...         ...       ...
```

An example of a valid path name for this is `/sys/data/123-456-7890/NAME`, and its value is `XXX`.

One can also imagine more complex and intelligent ways of enumeration. For example, it is possible to consider an enumeration on a condition; e.g., `/sys/data/salary>50000`. In fact, a file system named *Semantic File System* that allows this has been developed by D. Gifford and his group at MIT.