12. Naming

Any problem in computing can be solved by another level of indirection.
David Wheeler

Introduction

This handout is about orderly ways of naming complicated collections of objects in a computer system. A basic technique for understanding a big system is to describe it as a collection of simple parts. Being able to name these parts is a necessary aspect of such a description, and often the most important aspect.

The basic idea can be expressed in two ways that are more or less equivalent:

> Identify values by variable length names called *path names* that are sequences of simple names that are strings. Think of all the names with the same prefix (for instance, `/udir/lampson` and `/udir/lynch`) as being grouped together. This grouping induces a tree structure on the names. Non-leaf nodes in the tree are *directories*.
>
> Make a tree of nodes with simple names on the arcs. The leaf nodes are values and the internal nodes are *directories*. A node is named by a path through the tree from the root; such a name is called a *path name*.

Thus `/udir/lampson/pocs/handouts/12` is a path name for a value (perhaps the text of this handout), and `/udir/lampson/pocs/handouts` is a path name for a directory (other words for directory are folder, context, closure, environment, binding, and dictionary). The collection of all the path names that make sense in some situation is called a *name space*. Viewing a name space as a tree gives us the standard terminology of parents, children, ancestors, and descendants.

Using path names to name values (or objects, if you prefer) is often called 'hierarchical naming' or 'tree-structured naming'. There are a lot of other names for it that are used in special situations: mounting, search paths, multiplexing, device addressing, network references. An important reason for studying naming in general is that you don't have to start from scratch in understanding all those other things.

Path names are good because:

- The name space can grow indefinitely, and the growth can be managed in a decentralized way. That is, the authority to create names in one part of the space can be delegated, and thereafter there is no need for synchronization. Names that start `/udir/lampson` are independent of names that start `/udir/rinard`.
- Many kinds of data can be encapsulated under this interface, with a common set of operations. Arbitrary operations can be encoded as reads and writes of suitably chosen names.

As we have seen, a path name is a sequence of simple names. We use the types `N = String` for a simple name and `PN = SEQ N` for a path name. It is often convenient to write a path name as a string. The syntax of these strings is not important; it is just a convention for encoding the path names. Here are some examples:

`/udir/lampson/pocs/handouts/12`	Unix path name
`lampson@mediaone.net`	Internet mail address. The path name is `{"net", "mediaone", "lampson"}`
`16.23.5.193`	IP network address (fixed length)

We will normally write path names as Unix file names, rather than as the sequence constructors that would be correct Spec. Thus `a/b/c/1026` instead of `PN{"a","b","c","1026"}`.

People often try to distinguish a name (what something is) from an address (where it is) or a route (how to find it). This is a matter of levels of abstraction and must not be taken as absolute. At a given level of abstraction we tend to identify objects at that level by names, the lower-level objects that code them by addresses, and paths at lower levels by routes. Examples:

```
microsoft.com -> 207.46.130.149 -> SEQ [router output port, LAN address]
a/b/c/1026 -> INode/1026 -> DA/2 -> [cylinder, head, sector, byte 2]
```

Sometimes people talk about "descriptive names", which are queries in a database. We will see that these are readily encompassed within the framework of path names. That is a formal relationship, however. There is an important practical difference between a *designator* for a single entity, such as `lampson@mediaone.net`, and a *description* or query such as "everyone at MIT's LCS whose research involves parallel computing". The difference is illuminated by the comparison between the name `eecsfaculty@eecs.mit.edu` and the query "the faculty members in MIT's EECS department". The former name is probably maintained with some care; it's anyone's guess how reliable the answer to the query is. When using a name, it is wise to consider whether it is a designator or a description.

This is not to say that descriptions or queries are bad. On the contrary, they are very valuable, as any one knows who has ever used a web search engine. However, they usually work well only when a person examines the results with some care.

In the remainder of this handout we examine the specs for the two ways of describing a name space that we introduced earlier: as a memory addressed by path names, and as a tree (or more generally a graph) of directories. The two ways are closely related, but they give rise to somewhat different specs. Then we study the recursive structure of name spaces and various ways of inducing a name space on a collection of values. This leads to a more abstract analysis of how the spec for a name space can vary, depending on the properties of the underlying values. We conclude our general treatment by examining how to name a name space. Finally, we give a large number of examples of name spaces; you might want to look at these first to get some more context.

Name space as memory

We can view a name space as an example of the memory abstraction we studied earlier. Recall that a memory is a partial map `M = A -> V`. Here we take `A = PN` and replace `M` with `D` (for

directory). This kind of memory differs from the byte-addressable physical memory of a computer in several ways[1]:

- The map is partial.
- The domain is changing.
- The current value of the domain (that is, which names are defined) is interesting.
- `PN`'s with the same prefix are related (though not as much as in the second view of name spaces).

Here are some examples of name spaces that can naturally be viewed as memories:

> The Simple Network Management Protocol (SNMP) is used to manage components of the Internet. It uses path names (rooted in IP addresses) to name values, and the basic operations are to read and write a single named value.
>
> Several file systems use a single large table to map the path name of a file to the extents that represent it.

```
MODULE MemNames0[V] EXPORT Read, Write, Remove, Enum, Next, Rename =

TYPE N         =  String                          % Name
     PN        =  SEQ N WITH {"<<=":=PNLE}        % Path Name
     D         =  PN -> V                         % Directory

VAR d          := D{}                             % the state

FUNC PNLE(pn1, pn2) -> Bool = pn1.LexLE(pn2, N."<=")   % pn1 <<= pn2
```

Here are the familiar `Read` and `Write` procedures; `Read` raises `error` if `d` is undefined at `pn`, for consistency with later specs. In this basic spec none of the other procedures raises `error`; this innocence will not persist when things get more complicated. It's common to also have a `Remove` procedure for making a `PN` undefined; note that unlike a file system, this `Remove` does not erase the values of longer names that start with `PN`. This is because, unlike a file system, this spec does not ensure that every prefix of a defined `PN` is defined.

```
FUNC Read(pn) -> V RAISES {error} = RET d(pn) [*] RAISE error

APROC Write(pn, v) = << d := d{pn -> v} >>

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

The body of `Write` is usually written `d(pn) := v`.

It's important that the map is partial, and that the domain changes. This means that we need operations to find out what the domain is. Simply returning the entire domain is not practical, since it may be too big, and usually only part of it is of interest. There are two schools of thought

[1] It differs much less from the virtual memory, in which the map may be partial and the domain may change as new virtual memory is assigned or files are mapped. Actually these things can happen to physical memory as well, especially in the part of it implemented by I/O devices.

about what form these operations should take, represented by the functions `Enum` and `Next`; only one of these is needed.

> `Enum` returns all the simple names that can lead to a value starting from `pn`; another way of saying this is that it returns all the names bound in the directory named `pn`. By recursively applying `Enum` to `pn + n` for each simple name `n` that `Enum` returns, you can explore the entire tree.
>
> On the other hand, if you keep feeding `Next` its own output, starting with `{}`, it walks the tree of defined names depth-first, returning in turn each `PN` that is bound to a `V`. It finishes with `{}`.

Note that what `Next` does is not the same as returning the results of `Enum` one at a time, since `Next` explores the entire tree, not just one directory. Thus `Enum` takes the organization of the name space into directories more seriously than does `Next`.

```
FUNC Enum(pn) -> SET N = RET {pn1 | d!(pn + pn1) | pn1.head}

FUNC Next(pn) -> PN    = VAR later := {pn' | d!pn' /\ pn <= pn'} |
     RET later.fmin(PN."<<=") [*] RET {}              % {} if later is empty
```

A separate issue is arranging to get a reasonable number of results from one of these procedures. If the directory is large, `Enum` as defined here may return an inconveniently large set, and we may have to call `Next` inconveniently many times. In real life we would make either routine return a sequence of `N`'s or `PN`'s, usually called a 'buffer'. This is a standard use of batching to reduce the overhead of invoking an operation, without allowing the batches to get too large. We won't add this complication to our specs.

Finally, there is a `Rename` procedure that takes directories quite seriously. It reflects the idea that all the names which start the same way are related, by changing all the names that start with `from` so that they start with `to`. Because directories are not very real in the representation, this procedure has to do a lot of work. It erases everything that starts with either argument, and then copies everything in the original `d` that starts with `from` to the corresponding path name that starts with `to`. Read `x <= y` as "`x` is a prefix of `y`".

```
APROC Rename(from: PN, to: PN) RAISES {error} = << VAR d0 := d |
     IF   from <= to => RAISE error                   % can't rename to a descendant
     [*]  DO VAR pn :IN d.dom | (to <= pn \/ from <= pn) => d := d{pn -> } OD;
          DO VAR pn | d(to + pn ) # d0(from + pn) => d(to + pn) := d0(from + pn) OD
     FI >>

END MemNames0
```

Here is a different version of `Rename` that makes explicit the relation between the initial state `d` and the final state `d'`. Read `x >= y` as "`x` is a suffix of `y`".

```
APROC Rename(from: PN, to: PN) RAISES {error} = <<
     IF VAR d' |
          (ALL x: PN, y: PN | (    x >= from                  => ~ d'!x
                             [*] x = to + y /\ d!(from + y) => d'(x) = d(from + y
                             [*] ~ x >= to /\ d!x           => d'(x) = d(x)
                             [*] ~ d'!x )
```

```
        => d := d'
    [*]  RAISE error FI >>
```

There is often a rule that a name can be bound to a directory or to a value, but not both. For this we need a slightly different spec that marks a name as bound to a directory by giving it the special value `isD`, with a separate procedure for making an empty directory. To enforce the new rule every routine can now raise `error`, and `Remove` erases the whole sub-tree. As usual, boxes mark the changes from `MemNames0`.

```
MODULE MemNames[V] EXPORT Read, Write, MakeD, Remove, Enum, Rename =

TYPE Dir         =  ENUM[isDir]
     D           =  PN -> (V + Dir) SUCHTHAT (\d| d({}) IS Dir)    % root a Dir

VAR  d           := D{{} -> isDir}

% INVARIANT (ALL pn, pn' | d!pn' /\ pn' > pn => d(pn) = isDir

FUNC Read(pn) -> V RAISES {error} = d(pn) IS V => RET d(pn) [*] RAISE error

FUNC Enum(pn) -> SET N RAISES {error} =
      d(pn) IS Dir => RET {pn1 | d!(pn + pn1) | pn1.head}    [*] RAISE error

APROC Write(pn, v) RAISES {error}  = << Set(pn, v) >>
APROC MakeDir(pn)  RAISES {error}  = << Set(pn, isDir) >>

APROC Remove(pn) =                                  % Erase everything with pn prefix.
    << DO VAR pn' :IN d.dom | (pn <= pn') => d := d{pn' -> } OD >>

APROC Rename(from: PN, to: PN) RAISES {error} = << VAR d0 := d |
    IF   from <= to => RAISE error                  % can't rename to a descendant
    [*]  DO VAR pn :IN d.dom | (to <= pn \/ from <= pn) => d := d{pn -> } OD;
         DO VAR pn | d(to + pn ) # d0(from + pn) =>
              d(to + pn) := d0(from + pn) OD
    FI >>

APROC Set(pn, y: (V + D) RAISES {error} =
    << pn # {} /\ d(pn.reml) IS D => d(pn) := y [*] RAISE error >>

END MemNames
```

A file system usually forbids overwriting a file with a directory (for no obvious reason) or overwriting a non-empty directory with anything (because a directory is precious and should not be clobbered wantonly), but these rules are rather arbitrary, and we omit them here.

Exercise: write a version of `Rename` that makes explicit the relation between the initial state `d` and the final state `d'`, in the style of the second `Rename` of `MemNames0`.

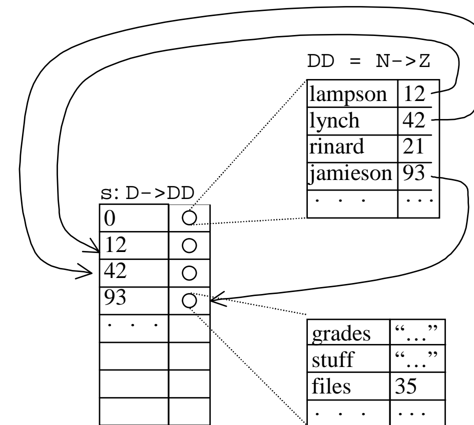
The `MemNames` spec is basically the same as the simple `Memory` spec. Complications arise because the domain can change, and because of the distinction between directories and values. The specs in the next section take this distinction much more seriously.

Name space as graph of directory objects

These specs are reasonably simple, but they are clumsy for operations on directories such as `Rename`. More fundamentally, they don't handle aliasing, where the same object has more than one name. The other (and more usual) way to look at a hierarchical name space is to think of each directory as a function that maps a simple name (not a path name) to a value or another directory, rather than thinking of the entire tree as a single `PN -> V` map. This tree (or general graph) structure maps a `PN` by mapping each `N` in turn, traversing a path through the graph of directories; hence the term 'path name'. We continue to use the type `D` for a directory.

Our eventual goal is a spec for a name space as graph that is 'object-oriented' in the sense that you can supply different code for each directory in the name space. We will begin, however, with a simpler spec that is equivalent to `MemNames`, evolve this to a more general spec that allows aliases, and finally add the object orientation.

The obvious thing to do is to make a `D` be a function `N -> Z`, where `Z = (D + V)` as before, and have a state variable `d` which is the root of the tree. Unfortunately this completely functional structure doesn't work smoothly, because there's no way to change the value of `a/b/c/d` without changing the value of `a/b/c` so that it contains the new value of `a/b/c/d`, and similarly for `a/b` and `a` as well.[2]



We solve this problem in the usual way with another level of indirection, so that the value of a directory name is not a `N -> Z` but some kind of reference or pointer to a `N -> Z`, as shown in the figure. This reference is an 'internal name' for a directory. We use the name `DD` for the actual

[2] The method of explicitly changing all the functions up to the root has some advantages. In particular, we can make several changes to different parts of the name space appear atomically by waiting to rewrite the root until all the changes are made. It is not very practical for a file system, though at least one has been built this way: H.E. Sturgis, *A Post-Mortem for a Time-sharing System*, PhD thesis, University of California, Berkeley, and Report CSL 74-1, Xerox Research Center, Palo Alto, Jan 1974. It has also been used in database systems to atomically change the entire database state; in this context it is called 'shadowing'. See Gray and Reuter, pp 728-732.

function `N -> Z` and introduce a state variable `s` that holds all the `DD` values; its type is `D->DD`. A `D` is just the internal name of a directory, that is, an index into `s`. We take `D = Int` for simplicity, but any type with enough values would do; in Unix `D = INo`. You may find it helpful to think of `D` as a pointer and `s` as a memory, or of `D` as an inode number and `s` as the inodes. Later sections explore the meaning of a `D` in more detail, and in particular the meaning of `root`.

Once we have introduced this extra indirection the name space does not have to be a tree, since two `PN`'s can have the same `D` value and hence refer to the same directory. In a Unix file system, for example, every directory with the path name `pn` also has the path names `pn/.`, `pn/./.`, etc., and if `pn/a` is a subdirectory, then the parent also has the names `pn/a/..`, `pn/a/../a/..`, etc. Thus the name space is not a tree or even a DAG, but a graph with cycles, though the cycles are constrained to certain stylized forms involving '`.`' and '`..`'. This means, of course, that there are defined `PN`'s of unbounded length; in real life there is usually an arbitrary upper bound on the length of a defined `PN`.

The spec below does not expose `D`'s to the client, but deals entirely in `PN`'s. Real systems often do expose the `D` pointers, usually as some kind of capability (for instance in a file system that allows you to open a directory and obtain a file descriptor for it), but sometimes just as a naked pointer (for instance in many distributed name servers). The spec uses an internal function `Get`, defined near the end, that looks up a `PN` in a directory; `GetD` is a variation that raises `error` if it can't return a `D`.

```
MODULE ObjNames0[V] EXPORT Read, Write, MakeD, Remove, Enum, Rename =

TYPE D         =  Int                          % just an internal name
     Z         =  (V + D)                      % the value of a name
     DD        =  N -> Z                       % a Directory

VAR  root      :  D := 0
     s         := (D -> DD){}{root -> DD{}}    % initially empty root

FUNC Read(pn) -> V RAISES {error} = VAR z := Get(root, pn) |
     IF z IS V => RET z [*] RAISE error FI

FUNC Enum(pn) -> SET PN RAISES {error} = RET s(GetD(root, pn)).dom
% Raises error if pn isn't a directory, like MemNames.
```

A write operation on the name `a/b/c` has to change the `d` component of the directory `a/b`; it does this through the procedure `SetPN`, which gets its hands on that directory by invoking `GetD(root, pn.reml)`.

```
APROC Write(pn, v) RAISES {error} = << SetPN(pn, v) >>
APROC MakeD(pn)  RAISES {error} = << VAR d := NewD() | SetPN(pn, d)  >>

APROC Remove(pn)   RAISES {error} =
     << VAR d := GetD(root, pn.reml) | >>

APROC Rename(from: PN, to: PN) RAISES {error} = <<
     IF   (to = {}) \/ (from <= to) => RAISE error      % can't rename to a descendant
     [*]  VAR fd := GetD(root, from.reml),               % know from, to # {}
              td := GetD(root, to  .reml) |
              s(fd)!(from.last) =>
                 s(td) := s(td)(to  .last -> s(fd)(from.last));
                 s(fd) := s(fd){from.last -> }
     [*]  RAISE error
     FI >>
```

The remaining routines are internal. The main one is `Get(d, pn)`, which returns the result of starting at `d` and following the path `pn`. `GetD` raises `error` if it doesn't get a directory. `NewD` creates a new, empty directory.

```
FUNC Get(d, pn) -> Z RAISES {error} =
% Return the value of pn looked up starting at z.
     IF   pn = {} => RET d
     [*]  VAR z :=s(d)(pn.head) | z IS D => RET Get(z, pn.tail)
     [*]  RAISE error
     FI

FUNC GetD(d, pn) -> D RAISES {error} = VAR z := Get(d, pn) |
     IF z IS D => RET z [*] RAISE error FI

APROC SetPN(pn, z) RAISES {error} =
     << VAR d := GetD(root, pn.reml) | s(d)(pn.last) := z >>

APROC NewD() -> D = << VAR d | ~ s!d => s(d) := DD{}; RET d >>

END ObjNames0
```

As we did with the second version of `MemNames0.Rename`, we can give a definition of `Get` in terms of a predicate. It says that there's a sequence `p` of directories starting at `d` and ending at the result of `Get`, such that the components of `pn` select the corresponding components of `p`; if there's no such sequence, raise `error`.

```
FUNC Child(z1, z2) -> Bool = z1 IS D /\ s!z1 /\ z2 IN s(z1).rng

FUNC Get(d, pn) -> Z RAISES {error} = <<
     IF VAR p :IN Child.paths |
         p.head = d /\ (ALL i :IN pn.dom | p(i+1) = s(p(i)(pn(i))) => RET p.last
     [*] RAISE error
     FI >>
```

`ObjNames0` is equivalent to `MemNames`. The abstraction function from `ObjNames0` to `MemNames` is

```
     MemNames.d = (\ pn | G(pn) IS V => G(pn) [*] G(pn) IS D => isD)
```

where we define a function `G` which is like `Get` on `root` except that it is undefined where `Get` raises `error`:

```
FUNC G(pn) -> Z = RET Get(root, pn) EXCEPT error => IF false => SKIP FI
```

The `EXCEPT` turns the `error` exception from `Get` into an undefined result for `G`.

Exercise: What is the abstraction function from `MemNames` to `ObjNames0`.

Objects, aliases, and atomicity

This spec makes clear the basic idea of interpreting a path name as a path through a graph of directories, but it is unrealistic in several ways:

The operations for changing the value of the `DD` functions in `s` may be very different from the `Write` and `MakeD` operations of `ObjNames0`. This happens when we impose the naming abstraction on a data structure that changes according to its own rules. SNMP is a good example; the values of names changes because of the operation of the network. Later in this handout we will explore a number of these variations.

There is often an 'alias' or 'symbolic link' mechanism which allows the value of a name `n` in context `d` to be a *link* `(d', pn)`. The meaning is that `d(n)` is a synonym for `Get(d', pn)`.

The operations are specified as atomic, but this is often too strong.

Our next spec, `ObjNames`, reflects all these considerations. It is rather complicated, but the complexity is the result of the many demands placed on it; ideas for simplifying it would be gratefully received. `ObjNames` is a fairly realistic spec for a naming system that allows for both symbolic links and extensible code for directories.

A `ObjNames.D` has `get` and `set` methods to allow for different code, though for now we don't take any advantage of this, but use the fixed code `GetFromS` and `SetInS`. In the section on object-oriented directories below, we will see how to plug in other versions of `D` with different `get` and `set` methods. The section on coherence below explains why `get` is a procedure rather than a function. These methods map undefined values to `nil` because it's tricky to program with undefined in this general setting; this means that `Z` needs `Null` as an extra case.

`Link` is another case of `Z` (the internal value of a name), and there is code in `Get` to follow links; the rules for doing this are somewhat arbitrary, but follow the Unix conventions. Because of the complications introduced by links, we usually use `GetDN` instead of `Get` to follow paths; this procedure converts a `PN` relative to `root` into a directory `d` and a name `n` in that directory. Then the external procedures read or write the value of that name.

Because `Get` is no longer atomic, it's no longer possible to define it in terms of a path through the directories that exists at a single instant. The section on atomicity below discusses this point in more detail.

```
MODULE ObjNames[V] EXPORT ... =

TYPE D          =  Int                                       % Just an internal name
                   WITH {get:=GetFromS, set:=SetInS}         % get returns nil if undefined
     Link       =  [d: (D + Null), pn]                       % d=nil for 'relative': the containing D
     Z          =  (V + D + Link + Null)                     % nil means undefined
     DD         =  N -> Z

CONST root      :  D := 0
VAR  s          := (D -> DD){}{root -> DD{}}                 % initially empty root

APROC GetFromS(d, n) -> Z =                                  % d.get(n)
    << RET s(d)(n) [*] RET nil >>

APROC SetInS  (d, n, z)   =                                  % d.set(n, z)
% If z = nil, SetInS leaves n undefined in s(d).
    << IF z # nil => s(d)(n) := z [*] s(d) := s(d){n -> } FI >>
```



```
PROC Read   (pn)      -> V     RAISES {error} = VAR z := Get(root, pn) |
    IF z IS V => RET z [*] RAISE error FI

PROC Enum   (pn)      -> SET N RAISES {error} =
% Can't just write RET GetD(root, pn).get.dom as in ObjNames0, because get isn't a function.
% The lack of atomicity is on purpose.
    VAR d := GetD(root, pn), ns: SET N := {}, z |
        DO VAR n | << z := d.get(n); ~ n IN ns /\ z # nil => ns + := {n} >> OD;
        RET ns

PROC Write  (pn, v)            RAISES {error} =                 SetPN(pn,v, true)

PROC MakeD(pn)                 RAISES {error} = VAR d := NewD() | SetPN(pn,d, false)

PROC Rename(from: PN, to: PN) RAISES {error} = VAR d, n, d', n' |
    IF  (to = {}) \/ (from <= to) => RAISE error      % can't rename to a descendant
    [*] (d, n) := GetDN(from, false); (d', n') := GetDN(to, false);
        << d.get!n => d'.set(n', d.get(n)); d.set(n, nil) >>
    [*] RAISE error
    FI
```

This version of `Rename` imposes a different restriction on renaming to a descendant than real file systems, which usually have a notion of a distinguished parent for each directory and disallow `ParentPN(d) <= ParentPN(d')`. They also usually require `d` and `d'` to be in the same 'file system', a notion which we don't have. Note that `Rename` does its two writes atomically, like many real file systems.

The remaining routines are internal. `Get` follows every link it sees; a link can appear at any point, not just at the end of the path. `GetDN` would be just

```
    IF pn = {} => RAISE error [*] RET (GetD(root, pn.reml), pn.last) FI
```

except for the question of what to do when the value of this `(d, n)` is a link. The `followLastLink` parameter says whether to follow such a link or not. Because this can happen more than once, the body of `GetDN` needs to be a loop.

```
PROC Get(d, pn) -> Z RAISES {error} = VAR z := d |
% Return the value of pn looked up starting at d.
    DO << pn # {} => VAR n := pn.head, z' |
        IF  z IS D =>                                 % must have a value for n.
            z' := z.get(n);
            IF  z' # nil =>
                % If there's a link, follow it. Otherwise just look up n.
                IF  (z, pn') := FollowLink(z, n); pn := pn' + pn.tail
                [*] z        := z'              ; pn :=       pn.tail
                FI
            [*] RAISE error
            FI
        [*] RAISE error
        FI
    >> OD; RET z

PROC GetD(d, pn) -> D RAISES {error} = VAR z := Get(d, pn) |
    IF z IS D => RET z AS D [*] RAISE error FI
```

```
PROC GetDN(pn, followLastLink: Bool) -> (D, N) RAISES {error} = VAR d := root |
% Convert pn into (d, n) such that d.get(n) is the item that pn refers to.
    DO  IF  pn = {} => RAISE error
        [*] VAR n := pn.last, z |
            d := Get(d, pn.reml);
            % If there's a link, follow it and loop. Otherwise return.
            << followLastLink => (d, pn) := FollowLink(d, n) [*] RET (d, n) >>
        FI
    OD
```

```
APROC FollowLink(d, n) -> (D, PN) = <<
% Fail if d.get(n) not Link. Use d as the context if the link lacks one.
    VAR l := d.get(n) | l IS Link => RET ((l.d IS D => l.d [*] d), l.pn) >>
```

```
PROC SetPN(pn, z, followLastLink: Bool) RAISES {error} =
    VAR d, n | (d, n) := GetDN(pn, followLastLink); d.set(n, z)

APROC NewD() -> D = << VAR d | ~ s!d => s(d) := D{}; RET d >>

END ObjNames
```

Object-oriented directories

Although `D` in `ObjNames` has `get` and `set` methods, they are the same for all `D`'s. To encompass the full range of applications of path names, we need to make a `D` into a full-fledged 'object', in which different instances can have different `get` and `set` operations (yet another level of indirection). This is the essential meaning of 'object-oriented': the type of an object is a record of routine types which defines a single interface to all objects of that type, but every object has its own values for the routines, and hence its own code.

To do this, we change the type to:

```
TYPE D      =  [get: APROC (n) -> Z, set: PROC (n, z) RAISES {error}]
     DR     =  Int                          % what D used to be; R for reference
```

keeping the other types from `ObjNames` unchanged:

```
     Z      =  (V + D + Link + Null)        % nil means undefined
     DD     =  N -> Z
```

We also need to change the state to:

```
CONST root  := NewD()
      s     := (DR -> DD){root -> DD{}}     % initially empty root
```

and to provide a new version of the `NewD` procedure for creating a new standard directory. The routines that `NewD` assigns to `get` and `set` have the same bodies as the `GetFromS` and `SetInS` routines.

A technical point: The reason for not writing `get:=s(dr)` in `NewD` is that this would capture the value of `s(dr)` at the time `NewD` is invoked; we want the value at the time `get` is invoked, and this is what we get because of the fact that Spec functions are functions on the global state, rather than pure functions.

```
APROC NewD() -> D = << VAR dr | ~ s!dr =>
    s(dr) := DD{};
    RET D{ get := (\ n | s(dr)(n)),
```



```
           set := (PROC (n, z) = IF  z # nil => s(dr)(n) := z
                                 [*] s(dr) := s(dr){n -> } FI) }
```

```
PROC SetErr(n, z) RAISES {error} = RAISE error
% For later use as a set proc if the directory is read-only
```

We don't need to change anything else in `ObjNames`.

We will see many other examples of `get` and `set` routines. Note that it's easy to define a `D` that disallows updates, by making `set` be `SetErr`.

Views and recursive structure

In this section we examine ways of constructing name spaces, and in particular ways of building up directories out of existing directories. We already have a basic recursive scheme that makes a set of existing directories the children of a parent directory. The generalization of this idea is to define a function on some state that returns a `D`, that is, a pair of `get` and `set` procedures. There are various terms for this:

'encapsulating' the state,

'embedding' the state in a name space,

'making the state compatible' with a name space interface,

defining a 'view' on the state.

We will usually call it a view. The spec for a view defines how the result of `get` depends on the state and how `set` affects the state.

All of these terms express the same idea: make the state behave like a `D`, that is, abstract it as a pair of `get` and `set` procedures. Once packaged in this way, it can be used wherever a `D` can be used. In particular, it can be an argument to one of the recursive views that make a `D` out of other `D`'s: a parent directory, a link, or the others discussed below. It can also be the argument of tools like the Unix commands that list, search, and manipulate directories.

The read operations are much the same for all views, but updates vary a great deal. The two simplest cases are the one we have already seen, where you can set the value of a name just as you write into a memory location, and the even simpler one that disallows updates entirely; the latter is only interesting if `get` looks at global state that can change in other ways, as it does in the `Union` and `Filter` operations below. Each time we introduce a view, we will discuss the spec for updating it.

In the rest of this section we describe views that are based on directories: links, mounting, unions, and filters. The final section of the handout gives many examples of views based on other kinds of data.

Links and mounting

The idea behind links (called 'symbolic links' in Unix, 'shortcuts' in Windows, and 'aliases' in the Macintosh) is that of an alias (another level of indirection): we can define the value of a name

in a directory by saying that it is the same as the value of some other name in some other directory. If the value is a directory, another way of saying this is that we can represent a directory `d` by the link `(d', pn')`, with `d(pn) = d'(pn')(pn)`, or more graphically `d/pn = d'/pn'/pn`. When put in this form it is usually called *mounting* the directory `d'(pn')` on `pn0`, if `pn0` is the name of `d`. In this language, `pn0` is called a 'mount point'. Another name for it is 'junction'.

We have already seen code in `ObjNames` to handle links. You might wonder why this code was needed. Why isn't our wonderful object-oriented interface enough? The reason is that people expect more from aliases than this interface can deliver: there can be an alias for a value, not only for a directory, and there are complicated rules for when the alias should be followed silently and when it should be an object in its own right that can be enumerated or changed

Links and mounting make it possible to give objects the names you want them to have, rather than the ones they got because of defects in the system or other people's bad taste. A very down-to-earth example is the problems caused by the restriction in standard Unix that a file system must fit on a single disk. This means that in an installation with 4 disks and 12 users, the name space contains `/disk1/john` and `/disk2/mary` rather than the `/udir/john` and `/udir/mary` that we want. By making `/udir/john` be a link to `/disk1/john`, and similarly for the other users, we can hide this annoyance.

Since a link is not just a `D`, we need extra interface procedures to read the value of a link (without following it automatically, as `Read` does), and to install a link. We call the install procedure `Mount` to emphasize that a mount point and a symbolic link are essentially the same thing. The `Mount` procedure is just like `Write` except for the second argument's type and the fact that it doesn't follow a final link in `pn`.

```
PROC ReadLink(pn) -> Link RAISES {error} = VAR d, n |
     (d, n) := GetDN(pn, false);
     VAR z | z := d.get(n); IF z IS Link => RET z [*] RAISE error FI

PROC Mount(pn, link) -> DD = SetPN(pn, link, false)
```

The section on roots below discusses where we might get the `D` in the `link` argument of `Mount`. In the common case of a link to someplace in the same name space, we have:

```
PROC MakeLink(pn, pn', local: Bool) =
     Mount(pn, Link{d := (local => nil [*] root), pn := pn'})
```

Updating (with `Write`, for instance) makes sense when there are links, but there are two possibilities. If every link is followed then a link never gets updated, since `GetDN` never returns a reference to a link. If a final link is not followed then it can be replaced by something else.

What is the relation between these links and what Unix calls 'hard links'? A Unix hard link is an inode number, which you can think of as a direct pointer to a file; it corresponds to a `D` in `ObjNames`. Several directory entries can have the same inode number. Another way to look at this is that the inodes are just another kind of name of the form `inodeRoot/2387754`, so that a hard link is just a link that happens to be an inode number rather than an ordinary path name. There is no provision for making the value of an inode number be a link (or indeed anything except a file), so that's the end of the line.

Unions

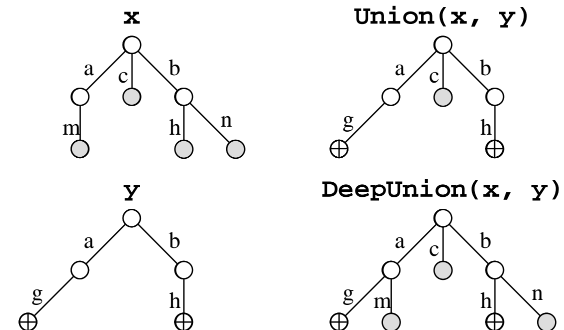
Since a directory is a function `N -> Z`, it is natural to combine two directories with the `"+"` overlay operator on functions[3]. If we do this repeatedly, writing `d1 + d2 + d3`, we get the effect of a 'search path' that looks at `d3` first, then `d2`, and finally `d1` (in that order because `"+"` gives preference to its second argument, unlike a search path which gives preference to its first argument). The difference is that this rule is part of the name space, while a search path must be coded separately in each program that cares. It's unclear whether an update of a union should change the first argument, change the second argument, do something more complicated, or raise an error. We take the last view for simplicity.

```
FUNC Union(d1, d2) -> D = RET D{get := d1.get + d2.get, set := SetErr}
```
[4]

Another kind of union combines the name spaces at every level, not just at the top level, by merging directories recursively. This is the most general way to combine two trees that have evolved independently.

```
FUNC DeepUnion(d1, d2) -> D = RET D{
     get := (\ n |
            (    d1.get(n) IS D /\ d2.get(n) IS D => DeepUnion(d1.get(n), d2.get(n))
             [*] (d1.get + d2.get)(n) )),
     set := SetErr}
```

This is a spec, of course, not efficient code.



Filters and queries

Given a directory `d`, we can make a smaller one by selecting some of `d`'s children. We can use any predicate for this purpose, so we get:

```
FUNC Filter(d, p: (D, N) -> Bool) -> D =
     RET D{get := ( \ n | (p(d, n) => d.get(n)) [*] nil ), set := SetErr}
```

[3] See section 9 of the Spec reference manual.
[4] This is a bit oversimplified, since `get` is an `APROC` and hence doesn't have `"+"`defined. But the idea should be clear. Plan 9 (see the examples at the end) implements unions.

Examples:

Pattern match in a directory: `a/b/*.ps`. The predicate is true if `n` matches `*.ps`.

Querying a table: `payroll/salary>25000/name`. The predicate is true if `Get(d, n/salary) > 25000`. See the example of viewing a table in the final section of examples.

Full text indexing: `bwl/papers/word:naming`. The predicate is true if `d.get(n)` is a text file that contains the word `naming`. The code could just search all the text files, but a practical one will probably involve an auxiliary index structure that maps words to the files that contain them, and will probably not be perfectly coherent.

See the 'semantic file system' example below for more details and a reference.

Variations

It is useful to summarize the ways in which a spec for a name space might vary. The variations almost all have to do with the exact semantics of updates:

What operations are updates, that is, can change the results of `Read`?

Are there aliases, so that an update to one object can affect the value of others?

Are the updates atomic, or it is possible for reads to see intermediate states? Can an update be lost, or partly lost, if there is a crash?

Viewed as a memory, is the name space *coherent*? That is, does every read that follows an update see the update, or is it possible for the old state to hang around for a while?

How much can the set of defined `PN`'s change? In other words, is it useful to think about a *schema* for the name space that is separate from the current state?

Updates

If the directories are 'real', then there will be non-trivial `Write`, `MakeD`, and `Rename` operations. If they are not, these operations will always raise `error`, there will be operations to update the underlying data, and the view function will determine the effects of these updates on `Read` and `Enum`. In many systems, `Read` and `Write` cannot be modeled as operations on memory because `Write(a, r)` does not just change the value returned by `Read(a)`. Instead they must be understood as methods of (or messages sent to) some object.

The earliest example of this kind of system is the DEC Unibus, the prototype for modern I/O systems. Devices on such an I/O bus have 'device registers' that are named as locations in memory. You can read and write them with ordinary load and store instructions. Each device, however, is free to interpret these reads and writes as it sees fit. For example, a disk controller may have a set of registers into which you can write a command which is interpreted as "read `n` disk blocks starting at address `da` into memory starting at address `a`". This might take three writes, for the parameters `n`, `da`, and `a`, and the third write has the side effect of starting execution of the command.

The most recent well-known incarnation of this idea is the World Wide Web, in which read and write actions (called `Get` and `Post` in the protocol) are treated as messages to servers that can search databases, accept orders for pizza, or whatever.

Aliases

We have already discussed this topic at some length. Links and unions both introduce aliases. There can also be 'hard links', which are several occurrences of the same `D`. In a Unix file system, for example, it is possible to have several directory entries that point to the same file. A hard link differs from a soft link because the connection it establishes between a name and a file cannot be broken by changing the binding of some other name. And of course a view can introduce arbitrarily complicated aliasing. For example, it's fairly common for an I/O device that has internal memory to make that memory addressable with two control registers `a` and `v`, and the rule that a read or write of `v` refers to the internal memory location addressed by the current contents of `a`.

Atomicity

The `MemNames` and `ObjNames0` specs made all the update operations atomic. For code to satisfy these specs, it must hold some kind of lock on every directory touched by `GetDN`, or at least on the name looked up in each such directory. This can involve a lot of directories, and since the name space is a graph it also introduces the danger of deadlock. It's therefore common for systems to satisfy only the weaker atomicity spec of `ObjNames`, which says that looking up a simple name is atomic, but the entire lookup process is not.

This means that `Read(/a/x)` can return `3` even though there was never any instant at which the path name `/a/x` had the value `3`, or indeed was defined at all. To see how this can happen, suppose:

initially `/a` is the directory `d1` and `/b` is undefined;

initially `x` is undefined in `d1`;

concurrently with `Read(/a/x)` we do `Rename(/a, /b); Write(/b/x, 3)`.

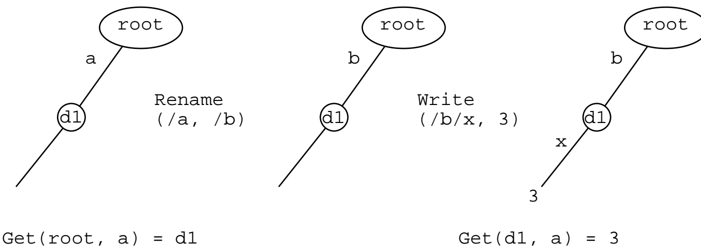
The following sequence of actions yields `Read(/a/x) = 3`:

In the `Read` , `Get(root, a) = d1`

`Rename(/a, /b)` makes `/a` undefined and `d1` the value of `/b`

`Write(/b/x, 3)` makes `3` the value of `x` in `d1`

In the `Read`, `RET d1.get(x)` returns `3`.



Obviously, whether this possibility is important or not depends on how clients are using the name space.

Coherence

Other things being equal, everyone prefers a coherent or 'sequentially consistent' memory, in which there is a single order of all the concurrent operations with the property that the result of every read is the result that a simple memory would return after it has done all the preceding writes in order. Maintaining coherence has costs, however, in the amount of synchronization that is required if parts of the memory are cached, or in the amount of availability if the memory is replicated. We will discuss the first issue in detail at the end of the course. Here we consider the availability of a replicated memory.

Recall the majority register from the beginning of the course. It writes a majority of the replicas and reads from a majority, thus ensuring that every read must see the most recent write. However, this means that you can't do either a read or a write unless you can talk to a majority. There we used a general notion of majority in which the only requirement is that every two majorities have a non-empty intersection. Applying this idea, we can define separate read and write *quorums*, with the property that every read quorum intersects every write quorum. Then we can make reads more available by making every replica a read quorum, at the price of having the only write quorum be the set of all replicas, so that we have to do every write to all the replicas.

An alternative approach is to weaken the spec so that it's possible for a read to see old values. We have seen a version of this already in connection with crashes and write buffering, where it was possible for the system to revert to an old state after a crash. Now we propose to make the spec even more non-deterministic: you can read an old value at any time, and the only restriction is that you won't read a value older than the most recent `Sync`. In return, we can now have much more availability in the code, since both a read and a write can be done to a single replica. This means that if you do `Write(/a, 3)` and immediately read `a`, you may not get `3` because the `Read` might use a different replica that hasn't seen the `Write` yet. Only `Sync` requires communication among the replicas.

We give the spec for this as a variation on `ObjNames`. We allow `nil` to be in `dd(n)`, representing the fact that `n` has been undefined in `dd`.


```
TYPE DD          =  N -> SEQ Z                              % remember old values

APROC GetFromS(d, n) -> Z = <<                              % we write d.get(n)
% The non-determinism wouldn't be allowed if this were a function
    VAR z | z IN s(d)(n) => RET z [*] RET nil >>            % return any old value

PROC  SetToS(d, n, z) =                                     % we write d.set(n, z)
    s(d)(n) := ((s(d)!n => s(d)(n) [*] {}) + {z}            % add z to the state

PROC Sync(pn) RAISES {error} =
    VAR d, n, z |
        (d, n) := GetDN(pn, true); z := s(d)(n).last;
        IF z # nil => s(d)(n) := {z} [*] s(d) := s(d){n -> } FI
```

This spec is common in the naming service for a distributed system, for instance in the Internet's DNS or Microsoft's Active Directory. The name space changes slowly, it isn't critical to see the very latest value, and it *is* critical to have high availability. In particular, it's critical to be able to look up names even when network partitions make some working replicas unreachable.

Schemas

In the database world, a schema is the definition of what names are defined (and usually also of the type of each name's value).[5] Network management calls this a 'management information base' or MIB. Depending on the application there are very different rules about how the schema is defined.

In a file system, for example, there is usually no official schema written down. Nonetheless, each operating system has conventions that in practice have the force of law. A Unix system without `/bin` and `/etc` will not get very far. But other parts of the name space, especially in users' private directories, are completely variable.

By contrast, a database system takes the schema very seriously, and a management system takes at least some parts of it seriously. The choice has mainly to do with whether it is people or programs that are using the name space. Programs tend to be much less flexible; it's a lot of work to make them adapt to missing data or pay attention to unexpected additional data

Minor issues

We mention in passing some other, less fundamental, ways in which the specs for name spaces differ.

> *Rules about overwriting*. Some systems allow any name to be overwritten, others treat directories, or non-empty directories, specially to reduce the consequences of careless errors.
>
> *Access control.* Many systems enforce rules about which users or programs are allowed to read or write various parts of the name space.

[5] Gray and Reuter, *Transaction Processing*, Morgan Kaufmann, 1993, pp 768-786.

Resource control. Writes often consume resources that are expensive or in fixed supply, such as disk blocks. This means that they can fail if the resources are exhausted, and there may also be a quota system that limits the resource consumption of users or programs.

Roots

It's not turtles all the way down.

Anonymous

So far we have ducked the question of how the `root` is represented, or the `D` in a link that plays a similar role. In `ObjNames0` we said `D = Int`, leaving its interpretation entirely to the `s` component of the state. In `ObjNames` we said `D` is a pair of procedures, begging the question of how the procedures are represented. The representation of a root depends entirely on the implementation. In a file system, for instance, a root names a disk, a disk partition, a volume, a file system exported from a server, or something like that. Thus there is another name space for the roots (another level of indirection). It works in a wide variety of ways. For example:

In MS-DOS. you name a physically connected disk drive. If the drive has removable media and you insert the wrong one, too bad.

On the Macintosh. you use the string name of a disk. If the system doesn't know where to find this disk, it asks the user. If you give the same name to two removable disks, too bad.

On Digital VMS. disks have unique identifiers that are used much like the string names on the Macintosh.

For the NFS network file system, a root is named by a host name or IP address, plus a file system name or handle on that host. If that name or address gets assigned to another machine, too bad.

In a network directory a root is named by a unique identifier. There is also a set of servers that might store replicas of that directory.

In the secure file system, a root is named by the hash of a public encryption key. There's also a network address to help you find the file system, but that's only a hint.[6]

In general it is a good idea to have absolute names (unique identifiers) for directories. This at least ensures that you won't use the wrong directory if the information about where to find it turns out to be wrong. A UID doesn't give much help in locating a directory, however. The possibilities are:

Store a set of places to look along with the UID. The problem is keeping this set up to date.

Keep another name space that maps UID's to locations (yet another level of indirection). The problem is keeping this name space up to date, and making it sufficiently available. For the former, every location can register itself periodically. For the latter, replication is good. We will talk about replication in detail later in the course.

Search some ad-hoc set of places in the hope of finding a copy. This search is often called a 'broadcast'.

[6] Mazières, Kaminsky, Kaashoek, and Witchel, Separating key management from file system security. *Proc. 17th ACM Symposium on Operating Systems Principles*, Dec. 1999. www.pdos.lcs.mit.edu/papers/sfs:sosp99.pdf.

We defined the interface routines to start from a fixed `root`. Some systems, such as Unix, have provisions for changing the root; the `chroot` system call does this for a process. In addition, it is common to have a more local context (called a 'working directory' for a file system), and to have syntax to specify whether to start from the root or the working directory (presence or absence of an initial '`/`' for a Unix file system).

Examples

These are to expand your mind and to help you recognize a name space when you come across it under some disguise.

File system directory

Example: `/udir/lampson/pocs/handouts/12-naming`

Not a tree, because of `.` and `..`, hard links, and soft links.
Devices, named pipes, and other things can appear as well as files.
Links and mounting are important for assembling the name space you want.
Files may have attributes, which are a little directory attached to the file.
Sometimes resources, fonts, and other OS rigmarole are stored this way.

inodes

There is a single inode directory, usually coded as a function rather than a table: you compute the location of the inode on the disk from the number.
For system-wide inodes, prefix a system-wide file system or volume name.

Plan 9[7]

This operating system puts all its objects into a single name space: files, devices, pipes, processes, display servers, and search paths (as union directories).

Semantic file system[8]

Not restricted to relational databases.

Free-text indexing: `~lampson/Mail/inbox/(word="compiler")`

Program cross-reference: `/project/sources/(calls="DeleteFile")`

Table (relational data base)

Example:

ID no (key)	*Name*	*Salary*	*Married?*
1432	Smith	21,000	Yes
44563	Jones	35,000	No
8456	Brown	17,000	Yes

We can view this as a naming tree in several ways:

`#44563/Name = Jones` key's value is a `D` that defines `Name`, `Salary`, etc.
`Name/#44563 = Jones` key's value is the `Name` field of its row

The second way, `cat Name/*` yields

```
Smith Jones Brown
```

[7] Pike et al., The use of name spaces in Plan 9, *ACM Operating Systems Review* **27**, 2, Apr. 1993, pp 72-76.
[8] Gifford et al., Semantic file systems, *Proc. 13th ACM Symposium on Operating System Principles*, Oct. 1991, pp 16-25 (handout 13).

Network naming[9] Example: `theory.lcs.mit.edu`

Distributed code. Can share responsibility for following the path between client and server in many ways.
A directory handle is a machine address (interpreted by some communication network), plus some id for the directory on that machine.
Attractive as top levels of complete naming hierarchy.

E-mail addresses Example: `rinard@lcs.mit.edu`

This syntax patches together the network name space and the user name space of a single host. Often there are links (called forwarding) and directories full of links (called distribution lists).

SNMP[10] Example: Router with circuits, packets in circuits, headers in packets, etc.

Internet Simple Network Management Protocol
Roughly, view the state of the managed entity as a table, treating it as a name space the way we did earlier. You can read or write table entries.
The `Next` action allows a client to explore the name space, whose structure is read-only. Ad hoc `Write` actions are sometimes used to modify the structure, for instance by adding a row to a table.

Page tables Divide up the virtual address, using the first chunk to index a first level page table, later chunks for lower level tables, and the last chunk for the byte in the page.

I/O device addressing Example:
Memory bus.
SCSI controller, by device register addresses.
SCSI device, by device number `0..7` on SCSI bus.
Disk sector, by disk address on unit.

Usually there is a pure read/write interface to the part of the I/O system that is named by memory addresses (the device registers in the example), and a message interface to the rest (the disk in the example).

Multiplexing a channel Examples:
Node-node network channel → *n* process-process channels.
Process-kernel channel → *n* inter-process channels.
ATM virtual path → *n* virtual circuits.

Given a channel, you can multiplex it to get sub-channels.
Sub-channels are identified by addresses in messages on the main channel.
This idea can be applied recursively, as in all good name spaces.

LAN addresses 48-bit ethernet address. This is flat: the address is just a UID.

[9] B. Lampson, Designing a global name service, *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp 1-10. RFC 1034/5 for DNS.
[10] M. Rose, *The Simple Book*, Prentice-Hall, 1990.

Hierarchical network addresses[11] Example: `16.24.116.42` (an IP address).

An address in a big network is hierarchical.
A router knows its parents and children, like a file directory, and also its siblings (because the parent might be missing)
To route, traverse up the name space to least common ancestor of current place and destination, then down to destination.

Network reference[12] Example: `6.24.116.42/11234/1223:44 9 Jan 1995/item 21`

Network address + port or process id + incarnation + more multiplexing + address or export index.
Some applications are remote procedure binding, network pointer, network object

Abbreviations A, talking to B, wants to pass a big value `V`, say a font or security credentials.
A makes up a short name `N` for `V` (sometimes called a 'cookie', though it's not the same as a Web cookie) and passes that.
If B doesn't know `N`'s value `V`, it calls back to A to get it, and caches the result.
Sometimes A tells `V` to B when it chooses `N`, and B is expected to remember it.
This is not as good because B might run out of space or fail and restart.

World Wide Web Example: `http://ds.internic.net/ds/rfc-index.html`

This is the URL (Uniform Resource Locator) for Internet RFCs.
The Web has a read/write interface.

Spec names Example: `ObjNames.Enum`

Telephone numbers Example: `1-617-253-6182`

Postal addresses Example:

```
Prof. Butler Lampson
Room 43-535
MIT
Cambridge, MA 02139
```

[11] R. Perlman, *Connections*, Prentice-Hall, 1993.
[12] Andrew Birrell et al., Network objects, *Proc. 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993 (handout 25).

13. Paper: Semantic File Systems

The attached paper by David Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole was presented at the 13th ACM Symposium on Operating Systems Principles, 1991, and appeared in its proceedings, *ACM Operating Systems Review*, Oct. 1991, pp 16-25.

Read it as an adjunct to the lecture on naming

Semantic File Systems

David K. Gifford, Pierre Jouvelot[1],
Mark A. Sheldon, James W. O'Toole, Jr.

Programming Systems Research Group
MIT Laboratory for Computer Science

Abstract

A *semantic file system* is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers*. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Compatiblity with existing file system protocols is provided by introducing the concept of a *virtual directory*. Virtual directory names are interpreted as queries, and thus provide flexible associative access to files and directories in a manner compatible with existing software. Rapid attribute-based access to file system contents is implemented by automatic extraction and indexing of key properties of file system objects. The automatic indexing of files and directories is called "semantic" because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Experimental results from a semantic file system implementation support the thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming.

1 Introduction

We would like to develop an approach for information storage that both permits users to share information more effectively, and provides reductions in programming effort and program complexity. To be effective this new approach must be used, and thus an approach that provides a transition path from existing file systems is desirable.

In this paper we explore the thesis that *semantic file systems* present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. A semantic file system is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers*. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Automatic indexing is performed when files or directories are created or updated.

The automatic indexing of files and directories is called "semantic" because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Through the use of specialized transducers, a semantic file system "understands" the documents, programs, object code, mail, images, name service databases, bibliographies, and other files contained by the system. For example, the transducer for a C program could extract the names of the procedures that the program exports or imports, procedure types, and the files included by the program. A semantic file system can be extended easily by users through the addition of specialized transducers.

Associative access is designed to make it easier for users to share information by helping them discover and locate programs, documents, and other relevant objects. For example, files can be located based upon transducer generated attributes such as author, exported or imported procedures, words contained, type, and title.

A semantic file system provides both a user interface and an application programming interface to its associative access facilities. User interfaces based upon browsers [Inf90, Ver90] have proven to be effective for query based access to information, and we expect browsers to be offered by most semantic file system implementations. Application programming interfaces that permit remote access include specialized protocols for information retrieval [NIS91], and remote procedure call based interfaces [GCS87].

It is also possible to export the facilities of a semantic file system without introducing any new interfaces. This can be accomplished by extending the naming semantics of files and directories to support associative access. A benefit of this approach is that all existing applications, including user interfaces, immediately inherit the benefits of associative access.

A semantic file system integrates associative access into a tree structured file system through the concept of a *virtual directory*. Virtual directory names are interpreted as queries and thus provide flexible associative access to files and directories in a manner compatible with existing software.

For example, in the following session with a semantic

This research was funded by the Defense Advanced Research Projects Agency of the U.S. Department of Defense and was monitored by the Office of Naval Research under grant number N00014-89-J-1988.

[1] Also with CRI, Ecole des Mines de Paris, France.

file system we first locate within a library all of the files that export the procedure lookup_fault, and then further restrict this set of files to those that have the extension c:

```
% cd /sfs/exports:/lookup_fault
% ls -F
virtdir_query.c@        virtdir_query.o@
% cd ext:/c
% ls -F
virtdir_query.c@
%
```

Semantic file systems can provide associative access to a group of file servers in a distributed system. This distributed search capability provides a simplified mechanism for locating information in large nationwide file systems.

Semantic file systems should be of use to both individuals and groups. Individuals can use the query facility of a semantic file system to locate files and to provide alternative views of data. Groups of users should find semantic file systems an effective way to learn about shared files and to keep themselves up to date about the status of group projects. As workgroups increasingly use file servers as shared library resources we expect that semantic file system technology will become even more useful.

Because semantic file systems are compatible with existing tree structured file systems, implementations of semantic file systems can be fully compatible with existing network file system protocols such as NFS [SGK$^+$85, Sun88] and AFS [Kaz88]. NFS compatibility permits existing client machines to use the indexing and associative access features of a semantic file system without modification. Files stored in a semantic file system via NFS will be automatically indexed, and query result sets will appear as virtual directories in the NFS name space. This approach directly addresses the "dusty data" problem of existing UNIX file systems by allowing existing UNIX file servers to be converted transparently to semantic file systems.

We have built a prototype semantic file system and run a series of experiments to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. We tried to locate various documents and programs in the file system using unmodified NFS clients. The results of these experiments suggest that semantic file systems can be used to find information more quickly than is possible using ordinary file systems, and add expressive power to command level programming languages.

In the remainder of the paper we discuss previous research (Section 2), introduce the interface and a semantics for a semantic file system (Section 3), review the design and implementation of a semantic file system (Section 4), present our experimental results (Section 5) and conclude with observations on other applications of virtual directories (Section 6).

2 Previous Work

Associative access to on-line information was pioneered in early bibliographic retrieval systems where it was found to be of great value in locating information in large databases [Sal83]. The utility of associative access motivated its subsequent application to file and document management. The previous research we build upon includes work on personal computer indexing systems, information retrieval systems, distributed file systems, new naming models for file systems, and wide-area naming systems:

- Personal computer indexing systems such as On Location [Tec90], Magellan [Cor], and the Digital Librarian [NC89b, NC89a] provide window-based file system browsers that permit word-based associative access to file system contents. Magellan and the Digital Librarian permit searches based upon boolean combinations of words, while On Location is limited to conjunctions of words. All three systems rank matching files using a relevance score. These systems all create indexes to reduce search time. On Location automatically indexes files in the background, while Magellan and the Digital Librarian require users to explicitly create indexes. Both On Location and the Digital Librarian permit users to add appropriate keyword generation programs [Cla90, NC89b] to index new types of files. However, Magellan, On Location, and the Digital Librarian are limited to a list of words for file description.

- Information retrieval systems such as Basis [Inf90], Verity [Ver90], and Boss DMS [Log91] extend the semantics of personal computer indexing systems by adding field specific queries. Fields that can be queried include document category, author, type, title, identifier, status, date, and text contents. Many of these document relationships and attributes can be stored in relational database systems that provide a general query language and support application program access. The WAIS system permits information at remote sites to be queried, but relies upon the user to choose an appropriate remote host from a directory of services [KM91, Ste91]. Distributed information retrieval systems [GCS87, DANO91] perform query routing based upon database content labels to ensure that all relevant hosts are contacted in response to a query.

- Distributed file systems [Sun89, Kaz88] provide remote access to files with tree structured names. These systems have enabled file sharing among groups of people and over wide geographic areas. Existing UNIX tools such as grep and find [Gro86] are often used to perform associative searches in distributed file systems.

- New naming models for file systems include the Portable Common Tool Environment (PCTE) [GMT86], the Property List DIRectory system (PLDIR) [Mog86], Virtual Systems [Neu90] and Sun's Network Software Environment (NSE) [SC88]. PCTE provides an entity-relationship database that models the attributes of objects including files. PCTE has been implemented as a compatible extension to UNIX. However, PCTE users must use specialized tools to query the PCTE database, and thus do not receive the benefits of associative access via a file system interface. The Property List DIRectory system implements a file system model designed around file properties and offers a Unix front-end user interface. Similarly, Virtual Systems permit users to hand-craft customized views of services, files, and directories. However, neither system provides automatic attribute extraction (although [Mog86] alludes to it as a possible extension) or attribute-based access to their contents. NSE is a network transparent software development tool that allows different views of

a file system hierarchy called *environments* to be defined. Unlike virtual directories, these views must be explicitly created before being accessed.

- Wide-area naming systems such as X.500 [CCI88], Profile [Pet88], and the Networked Resource Discovery Project [Sch89] provide attribute-based access to a wide variety of objects, but they are not integrated into a file system nor do they provide automatic attribute-based access to the contents of a file system.

Key advances offered by the present work include:

- Virtual directories integrate associative access into existing tree structured file systems in a manner that is compatible with existing applications.
- Virtual directories permit unmodified remote hosts to access the facilities of a semantic file system with existing network file system protocols.
- Transducers can be programmed by users to perform arbitrary interpretation of file and directory contents in order to produce a desired set of field-value pairs for later retrieval. The use of fields allows transducers to describe many aspects of a file, and thus permits subsequent sophisticated associative access to computed properties. In addition, transducers can identify entities within files as independent objects for retrieval. For example, individual mail messages within a mail file can be treated as independent entities.

Previous research supports our view that overloading file system semantics can improve system uniformity and utility when compared with the alternative of creating a new interface that is incompatible with existing applications. Examples of this approach include:

- Devices in UNIX appear as special files [RT74] in the `/dev` directory, enabling them to be used as ordinary files from UNIX applications.
- UNIX System III named pipes [Roc85, p. 159f] appear as special files, enabling programs to rendezvous using file system operations.
- File systems appear as special directories in Automount daemon directories [CL89, Pen90, PW90], enabling the binding of a name to a file system to be computed at the time of reference.
- Processes appear as special directories in Killian's process file system [Kil84], enabling process observation and control via file operations.
- Services appear as special directories in Plan 9 [PPTT90], enabling service access in a distributed system through file system operations in the service's name space.
- Arbitrary semantics can be associated with files and directories using Watchdogs [BP88], Pseudo Devices [WO88], and Filters [Neu90], enabling file system extensions such as terminal drivers, network protocols, X servers, file access control, file compression, mail notification, user specific directory views, heterogeneous file access, and service access.
- The ATTIC system [CG91] uses a modified NFS server to provide transparent access to automatically compressed files.

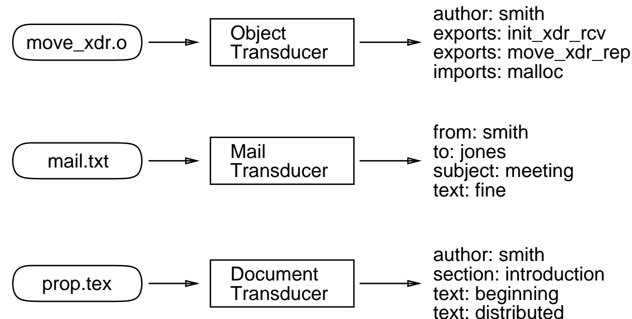


Figure 1: Sample Transducer Output

3 Semantic File System Semantics

Semantic file systems can implement a wide variety of semantics. In this section we present one such semantics that we have implemented. Section 6 describes some other possibilities.

Files stored in a semantic file system are interpreted by file type specific transducers to produce a set of descriptive attributes that enable later retrieval of the files. An *attribute* is a *field-value* pair, where a *field* describes a property of a file (such as its author, or the words in its text), and a *value* is a string or an integer. A given file can have many attributes that have the same field name. For example, a text file would have as many `text:` attributes as it has unique words. By convention, field names end with a colon.

A user extensible *transducer table* is used to determine the transducer that should be used to interpret a given file type. One way of implementing a transducer table is to permit users to store subtree specific transducers in the subtree's parent directory, and to look for an appropriate transducer at indexing time by searching up the directory hierarchy.

To accommodate files (such as mail files) that contain multiple objects we have generalized the unit of associative access beyond whole files. We call the unit of associative access an *entity*. An entity can consist of an entire file, an object within a file, or a directory. Directories are assigned attributes by directory transducers.

A transducer is a filter that takes as input the contents of a file, and outputs the file's entities and their corresponding attributes. A simple transducer could treat an input file as a single entity, and use the file's unique words as attributes. A complex transducer might perform type reconstruction on an input file, identify each procedure as an independent entity and use attributes to record their reconstructed types. Figure 1 shows examples of an object file transducer, a mail file transducer, and a TEX file transducer.

The semantics of a semantic file system can be readily extended because users can write new transducers. Transducers are free to use new field names to describe special attributes. For example, a CAD file transducer could introduce a `drawing:` field to describe a drawing identifier.

The associative access interface to a semantic file system is based upon queries that describe desired attributes of entities. A *query* is a description of desired attributes that permits a high degree of selectivity in locating entities of interest. The result of a query is a set of files and/or directories that contain the entities described. Queries are

boolean combinations of attributes, where each attribute describes the desired value of a field. It is also possible to ask for all of the values of a given field in a query result set. The values of a field can be useful when narrowing a query to eliminate entities that are not of interest.

A semantic file system is *query consistent* when it guarantees query results that correspond to its current contents. If updates cease to the contents of a semantic file system it will eventually be query consistent. This property is known as convergent consistency. The rate at which a given implementation converges is administratively determined by balancing the user benefits of fast convergence when compared with the higher processing cost of indexing rapidly changing entities multiple times. It is of course possible to guarantee that a semantic file system is always query consistent with appropriate use of atomic actions.

In the remainder of this section we will explore how conjunctive queries can be mapped into tree-structured path names. As we mentioned earlier, this is only one of the possible interfaces to the query capabilities of a semantic file system. It is also possible to map disjunction and negation into tree-structured names, but they have not been implemented in our prototype and we will not discuss them.

Queries are performed in a semantic file system through use of virtual directories to describe a desired view of file system contents. A virtual directory is computed on demand by a semantic file system. From the point of view of a client program, a virtual directory is indistinguishable from an ordinary directory. However, unlike ordinary directories, virtual directories do not have to be explicitly created to be accessed.

The query facilities of a semantic file system appear as virtual directories at each level of the directory tree. A *field virtual directory* is named by a field, and has one entry for each possible value of its corresponding field. Thus in `/sfs`, the virtual directory `/sfs/owner:` corresponds to the `owner:` field. The field virtual directory `/sfs/owner:` would have one entry for each owner that has written a file in `/sfs`. For example:

```
% ls -F /sfs/owner:
jones/          root/           smith/
%
```

The entries in a field virtual directory are value virtual directories. A *value virtual directory* has one entry for each entity described by a field-value pair. Thus the value virtual directory `/sfs/owner:/smith` contains entries for files in `/sfs` that are owned by Smith. Each entry is a symbolic link to the file. For example:

```
% ls -F /sfs/owner:/smith
bio.txt@        paper.tex@      prop.tex@
%
```

When an entity is smaller than an entire file, a view of the file can be presented by extending file naming semantics to include view specifications. To permit the conjunction of attributes in a query, value virtual directories contain field virtual directories. For example:

```
% ls -F /sfs/owner:/smith/text:/resume
bio.txt@
%
```

A pleasant property of virtual directories is their synergistic interaction with existing file system facilities. For example, when a symbolic link names a virtual directory the link describes a computed view of a file system. It is also possible to use file save programs, such as `tar`, on virtual directories to save a computed subset of a file system. It would be possible also to generalize virtual directories to present views of file systems with respect to a certain time in the past.

A semantic file system can be overlaid on top of an ordinary file system, allowing all file system operations to go through the SFS server. The overlaid approach has the advantage that it provides the power of a semantic file system to a user at all times without the need to refer to a distinguished directory for query processing. It also allows the server to do indexing in response to file system mutation operations. Alternatively, a semantic file system may create virtual directories that contain links to the files in the underlying file system. This means that subsequent client operations bypass the semantic file system server.

When an overlaid approach is used field virtual directories must be invisible to preserve the proper operation of tree traversal applications. A directory is *invisible* when it is not returned by directory enumeration requests, but can be accessed via explicit lookup. If field virtual directories were visible, the set of trees under `/sfs` in our above example would be infinite. Unfortunately making directories invisible causes the UNIX command `pwd` to fail when the current path includes an invisible directory. It is possible to fix this through inclusion of unusual `..` entries in invisible directories.

The distinguished `field:` virtual directory makes field virtual directories visible. This permits users to enumerate possible search fields. The `field:` directory is itself invisible. For example:

```
% ls -F /sfs/field:
author:/     exports:/    owner:/      text:/
category:/   ext:/        priority:/   title:/
date:/       imports:/    subject:/    type:/
dir:/        name:/
% ls -F /sfs/field:/text:/semantic/owner:/jones
mail.txt@        paper.tex@     prop.tex@
%
```

The syntax of semantic file system path names is:

```
<sfs-path>   ::=  /<pn> | <pn>
<pn>         ::=  <name> | <attribute>
                  <field-name> | <name>/<pn>
                  <attribute>/<pn>
<attribute>  ::=  field: | <field-name>/<value>
<field-name> ::=  <string>:
<value>      ::=  <string>
<name>       ::=  <string>
```

The semantics of semantic file system path names is:

- The universe of entities is defined by the path name prefix before the first virtual directory name.
- The contents of a field virtual directory is a set of value virtual directories, one for each value that the field describes in the universe.

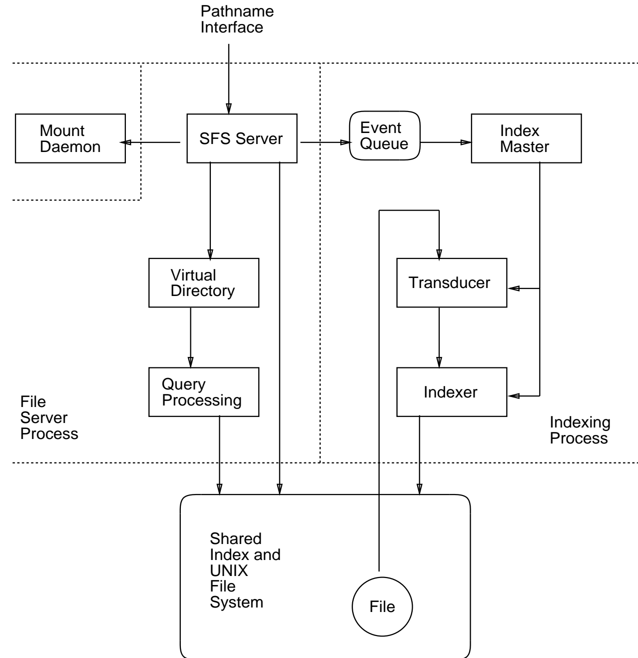


Figure 2: SFS Block Diagram

- The contents of a value virtual directory is a set of entries, one for each entity in the the universe that has the attribute described by the name of the value virtual directory and its parent field virtual directory. The contents of a value virtual directory defines the universe of entities for its subdirectories. In the absence of name conflicts, the name of an entry in a value virtual directory is its original entry name. Entry name conflicts are resolved by assigning nonce names to entries.

- The contents of a `field:` virtual directory is the set of fields in use.

4 Semantic File System Implementation

We have built a semantic file system that implements the NFS [SGK+85, Sun89] protocol as its external interface. To use the search facilities of our semantic file system, an Internet client can simply mount our file system at a desired point and begin using virtual directory names. Our NFS server computes the contents of virtual directories as necessary in response to NFS `lookup` and `readdir` requests.

A block diagram of our implementation is shown in Figure 2. The dashed lines in the figure describe process boundaries. The major processes are:

- The *client* process is responsible for generating file system requests using normal NFS style path names.

- The *file server process* is responsible for creating virtual directories in response to path name based queries. The SFS Server module implements a user level NFS server and is responsible for implementing the NFS interface to the system. The SFS Server uses *directory faults* to request computation of needed entries by the Virtual Directory module. A faulting mechanism is used because the SFS Server caches virtual directory results, and will only fault when needed information is requested the first time or is no longer cached. The Virtual Directory module in turn calls the Query Processing module to actually compute the contents of a virtual directory.

  The file server process records file system modification events in a write-behind log. The modification log eliminates duplicate modification events.

- The *indexing process* is responsible for keeping the index of file system contents up-to-date. The Index Master module examines the modification log generated by the file server process every two minutes. The indexing process responds to a file system modification event by choosing an appropriate transducer for the modified object. An appropriate transducer is selected by determination of the type of the object (e.g. C source file, object file, directory). If no special transducer is found a default transducer is used. The output of the transducer is fed to the Indexer module that inserts the computed attributes into the index. Indexing and retrieval are based upon Peter Weinberger's BTree package [Wei] and an adapted version of the `refer` [Les] software to maintain the mappings between attributes and objects.

- The *mount daemon* is contacted to determine the root file handle of the underlying UNIX file system. The file server process exports its NFS service using the same root file handle on a distinct port number.

- The *kernel* implements a standard file system that is used to store the shared index. The file server process could be integrated into the kernel by a VFS based implementation [Kle86] of an semantic file system. We chose to implement our prototype using a user level NFS server to simplify development.

Instead of computing all of the virtual directories that are present in a path name, our implementation only computes a virtual directory if it is enumerated by a client `readdir` request or a `lookup` is performed on one of its entries. This optimization allows the SFS Server to postpone query processing in the hope that further attribute specifications will reduce the amount of work necessary for computation of the result set. This optimization is implemented as follows:

- The SFS Server responds to a `lookup` request on a virtual directory with a `lookup_not_found` fault to the Virtual Directory module. The Virtual Directory module checks to make sure that the virtual directory name is syntactically well formed according to the grammar in Section 3. If the name is well formed, the directory fault is immediately satisfied by calling the `create_dir` procedure in the SFS Server. This procedure creates a placeholder directory that is used to satisfy the client's original `lookup` request.

- The SFS Server responds to a `readdir` request on a virtual directory or a `lookup` on one of its entries with a `fill_directory` fault to the Virtual Directory module. The Virtual Directory module collects all of the attribute specifications in the virtual directory path

name and passes them to the Query Processing module. The Query Processing module uses simple heuristics to reorder the processing of attributes to optimize query performance. The matching entries are then materialized in the placeholder directory by the Virtual Directory module that calls the `create_link` procedure in the SFS Server for each matching file or directory.

The transducers that are presently supported by our semantic file system implementation include:

- A transducer that describes New York Times articles with `type:`, `priority:`, `date:`, `category:`, `subject:`, `title:`, `author:`, and `text:` attributes.
- A transducer that describes object files with `exports:` and `imports:` attributes for procedures and global variables.
- A transducer that describes C, Pascal, and Scheme source files with `exports:` and `imports:` attributes for procedures.
- A transducer that describes mail files with `from:`, `to:`, `subject:`, and `text:` attributes.
- A transducer that describes text files with `text:` attributes. The text file transducer is the default transducer for ASCII files.

In addition to the specialized attributes listed above, all files and directories are further described by `owner`, `group`, `dir`, `name`, and `ext` attributes.

At present, we only index publicly readable files. We are investigating indexing protected files as well, and limiting query results to entities that can be read by the requester. We are in the process of making a number of improvements to our prototype implementation. These enhancements include 1) full support for multi-host queries using query routing, 2) an enhanced query language, 3) better support for file deletion and renaming, and 4) integration of views for entities smaller than files. Our present implementation deals with deletions by keeping a table of deleted entities and removing them from the results of query processing. Entities are permanently removed from the database when a full reindexing of the system is performed. We are investigating performing file and directory renames without reindexing the underlying files.

5 Results

We ran a series of experiments using our semantic file system implementation to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. All of the experimental data we report are from our research group's file server using a semantic file system. The server is a Microvax-3 running UNIX version 4.3bsd. The server indexes all of its publicly readable files and directories.

To compact the indexes our prototype system reconstructs a full index of the file system contents every week. On 23 July 1991, full indexing of our user file system processed 68 MBytes in 7,771 files (Table 5).[1] Indexing the resulting 1 million attributes took 1 hour and 36 minutes (Table 2). This works out to an indexing rate of 712 KBytes/minute.

[1] The 162 MBytes in publicly readable files that were not processed were in files for which transducers have not yet been written: executable files, PostScript files, DVI files, tar files, image data, etc.

Total file system size	326	MBytes
Amount publicly readable	230	MBytes
Amount with known transducer	68	MBytes
Number of distinct attributes	173,075	
Number of attributes indexed	1,042,832	

Type	Number of Files	KBytes
Object	871	8,503
Source	2,755	17,991
Text	1,871	20,638
Other	2,274	21,187
Total	7,771	68,319

Table 1: User File System Statistics for 23 July 1991

Part of index	Size in KBytes
Index Tables	6,621
Index Trees	3,398
Total	10,019

Phase	Time (hh:mm)
Directory Enumeration	0:07
Determine File Types	0:01
Transduce Directory	0:01
Transduce Object	0:08
Transduce Source	0:23
Transduce Text	0:23
Transduce Other	0:24
Build Index Tables[2]	1:22
Build Index Trees	0:06
Total	1:36

Table 2: User FS Indexing Statistics on 23 July 1991

File system mutation operations trigger incremental indexing. In update tests simulating typical user editing and compiling, incremental indexing is normally completed in less than 5 minutes. In these tests, only 2 megabytes of modified file data were reindexed. Incremental indexing is slower than full indexing in the prototype system because the incremental indexer does not make good use of real memory for caching. The full indexer uses 10 megabytes of real memory for caching; the incremental indexer uses less than 1 megabyte.

The indexing operations of our prototype are I/O bound. The CPU is 60% idle during indexing. Our measurements show that transducers generate approximately 30 disk transfers per second, thereby saturating the disk. Indexing the resulting attributes also saturates the disk. Although the transducers and the indexer use different disk drives, the transducer-indexer pipeline does not allow I/O operations to proceed in parallel on the two disks. Thus, we feel that we could double the throughput by improving the pipeline's

[2] in parallel with Transduce

structure.

We expect our indexing strategy to scale to larger file systems because indexing is limited by the update rate to a file system rather than its total storage capacity. Incremental processing of updates will require additional read bandwidth approximately equal to the write traffic that actually occurs. Past studies of Unix file system activity [OCH+85] indicate that update rates are low, and that most new data is deleted or overwritten quickly; thus, delaying slightly the processing of updates might reduce the additional bandwidth required by indexing.

To determine the increased latency of overlaid NFS operations introduced by interposing our SFS server between the client and the native file system, we used the `nhfsstone` benchmark [Leg89] at low loads. The delays observed from an unmodified client machine were smaller than the variation in latencies of the native NFS operations. Preliminary measurements show that `lookup` operations are delayed by 2 ms on average, and operations that generate update notifications incur a larger delay.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for information sharing:

- The typical response time for the first `ls` command on a virtual directory is approximately 2 seconds. This response time reflects a substantial time savings over linear search through our entire file system with existing tools. In addition, subsequent `ls` commands respond immediately with cached results.

  We ran a series of experiments to test how the number of attributes in a virtual directory name altered the observed performance of the `ls` command on a virtual directory. Attributes were added one at a time to arrive at the final path name:

  ```
  /sfs/text:/virtual/
       text:/directory/
       text:/semantic/
       ext:/tex/
       owner:/gifford
  ```

  The two properties of a query that affect its response time are the number of attributes in the query and the number of objects in the result set. The effect of an increase in either of these factors is additional disk accesses. Figure 3 illustrates the interplay of these factors. Each point on the response time graph is the average of three experiments. In a separate experiment we measured an average response time of 5.4 seconds when the result set grew to 545 entities.

- We began to use the semantic file system as soon as it was operable to help coordinate the production of this paper and for a variety of other everyday tasks. We have found the virtual directory interface to be easy to use. (We were immediately able to use the GNU Emacs directory editor DIRED [Sta87] to submit queries and browse the results. No code modification was required.) At least two users in our group reexamined their file protections in view of the ease with which other users could locate interesting files in the system.

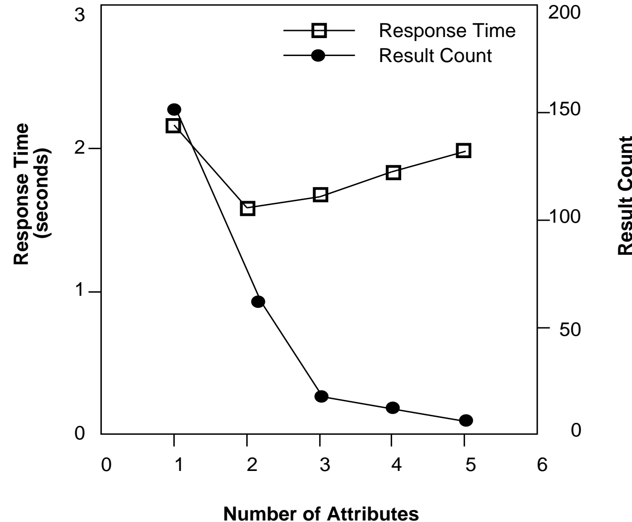


Figure 3: Plot of Number of Attributes vs. Response Time and Number of Results

- Users outside our research group have successfully used the query interface to locate information, including newspaper articles, in our file system.

- Users outside our research group have failed to find files for which no transducer had yet been installed. We are developing new transducers in response to these failed queries.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for command level programming:

- The UNIX shell pathname expansion facilities integrate well with virtual directories. For example, it is possible to query the file system for all `dvi` files owned by a particular user, and to print those whose names begin with a certain sequence of characters.

- Symbolic links have proven to be an effective way to describe file system views. The result of using such a symbolic link as a directory is a dynamically computed set of files.

6 Conclusions

We have described how a semantic file system can provide associative attribute-based access to the contents of an information storage system with the help of file type specific transducers. We have also discussed how this access can be integrated into the file system itself with virtual directories. Virtual directories are directories that are computed upon demand.

The results to date are consistent with our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. We plan to conduct further experiments to explore this thesis in further detail. We plan also to examine how virtual directories can directly benefit application programmers.

Our experimental system has tested one semantics for virtual directories, but there are many other possibilities. For example:

- The virtual directory syntax can be extended to support a richer query language. Disjunctive queries would permit users to use "or" in their queries, and would also offer the ability to search on multiple network semantic file systems concurrently.

- Users could assign attributes to file system entities in addition to the attributes that are automatically assigned by transducers.

- Transducers could be created for audio and video files. In principle this would permit access by time, frame number, or content [Nee91].

- The data model underlying a semantic file system could be enhanced. For example, an entity-relationship model [Cat83] would provide more expressive power than simple attribute based retrieval.

- The entities indexed by a semantic file system could include a wide variety of object types, including I/O devices and file servers. Wide-area naming systems such as X.500 [CCI88] could be presented in terms of virtual directories.

- A confederation of semantic file systems, possibly numbering in the thousands, can be organized into an *semantic library system.* A semantic library system exports the same interface as an individual semantic file system, and thus a semantic library system permits associative access to the contents of its constituent servers with existing file system protocols as well as with protocols that are designed specifically for content based access. A semantic library system is implemented by servers that use content based routing [GLB85] to direct a single user request to one or more relevant semantic file systems.

  We have already completed the implementation of an NFS compatible query processing system that forwards requests to multiple hosts and combines the results.

- Virtual directories can be used as an interface to other systems, such as information retrieval systems and programming environment support systems, such as PCTE. We are exploring also how existing applications could access object repositories via a virtual directory interface. It is possible to extend the semantics of a semantic file system to include access to individual entities in a manner suitable for an object repository [GO91].

- Relevance feedback and query results could be added by introducing new virtual directories.

The implementation of real-time indexing may require a substantial amount of computing power at a semantic file server. We are investigating how to optimize the task of real-time indexing in order to minimize this load. Another area of research is exploring how massive parallelism [SK86] might replace indexing.

An interesting limiting case of our design is a system that makes an underlying tree structured naming system superfluous. In such a system all directories would be computed upon demand, including directories that correspond to traditional tree structured file names. Such a system might help us share information more effectively by encouraging query based access that would lead to the discovery of unexpected but useful information.

Acknowledgments

We would like to thank Doug Grundman, Andrew Myers, and Raymie Stata, for their various contributions to the paper and the implementation. The referees provided valuable feedback and concrete suggestions that we have endeavored to incorporate into the paper. In particular, we very much appreciate the useful and detailed comments provided by Mike Burrows.

References

[BP88] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. In *USENIX Association 1988 Winter Conference Proceedings*, pages 267–275, Dallas, Texas, February 1988.

[Cat83] R. G. G. Cattell. Design and implementation of a relationship-entity-datum data model. Technical Report CSL-83-4, Xerox PARC, Palo Alto, California, May 1983.

[CCI88] CCITT. The Directory - Overview of Concepts, Models and Services. Recommendation X.500, 1988.

[CG91] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–211, Santa Clara, California, April 1991. ACM.

[CL89] Brent Callaghan and Tom Lyon. The automounter. In *USENIX Association 1989 Winter Conference Proceedings*, 1989.

[Cla90] Claris Corporation, Santa Clara, California, January 1990. News Release.

[Cor] Lotus Corporation. Lotus Magellan: Quick Launch. Product tutorial, Lotus Corporation, Cambridge, Massachusetts. Part number 35115.

[DANO91] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. Technical Report USC-TR 91-06, University of Southern California, Computer Science Department, 1991.

[GCS87] David K. Gifford, Robert G. Cote, and David A. Segal. Walter user's manual. Technical Report MIT/LCS/TR-399, M.I.T. Laboratory for Computer Science, September 1987.

[GLB85] David K. Gifford, John M. Lucassen, and Stephen T. Berlin. An architecture for large scale information systems. In *10th Symposium on Operating System Principles*, pages 161–170. ACM, December 1985.

[GMT86] Ferdinando Gallo, Regis Minot, and Ian Thomas. The object management system of PCTE as a software engineering database management system. In *Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12–15. ACM, December 1986.

[GO91] David K. Gifford and James W. O'Toole. Intelligent file systems for object repositories. In *Operating Systems of the 90s and Beyond*, Saarbrücken, Germany, July 1991. Internationales Begegnales- und Forschungszentrum für Informatik, Schloss Dagstuhl-Geschäftsstelle. To be published by Springer-Verlag.

[Gro86] Computer Systems Research Group. UNIX User's Reference Manual. 4.3 Berkeley Software Distribution, Berkeley, California, April 1986. Virtual VAX-11 Version.

[Inf90] Information Dimensions, Inc. BASISplus. The Key To Managing The World Of Information. Information Dimensions, Inc., Dublin, Ohio, 1990. Product description.

[Kaz88] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Association 1988 Winter Conference Proceedings*, pages 31–43, 1988.

[Kil84] T. J. Killian. Processes as files. In *USENIX Association 1984 Summer Conference Proceedings*, Salt Lake City, Utah, 1984.

[Kle86] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association 1986 Winter Conference Proceedings*, pages 238–247, 1986.

[KM91] Brewster Kahle and Art Medlar. An information system for corporate users: Wide area information servers. Technical Report TMC-199, Thinking Machines, Inc., April 1991. Version 3.

[Leg89] Legato Systems, Inc. Nhfsstone. Software package. Legato Systems, Inc., Palo Alto, California, 1989.

[Les] M. E. Lesk. Some applications of inverted indexes on the UNIX system. UNIX Supplementary Document, Section 30.

[Log91] Boss Logic, Inc. Boss DMS development specification. Technical documentation, Boss Logic, Inc., Fairfield, IA, February 1991.

[Mog86] Jeffrey C. Mogul. Representing information about files. Technical Report 86-1103, Stanford Univ. Department of CS, March 1986. Ph.D. Thesis.

[NC89a] NeXT Corporation. 1.0 release notes: Indexing. NeXT Corporation, Palo Alto, California, 1989.

[NC89b] NeXT Corporation. Text indexing facilities on the NeXT computer. NeXT Corporation, Palo Alto, California, 1989. from 1.0 Release Notes.

[Nee91] Roger Needham, 1991. Personal communication.

[Neu90] B. Clifford Neuman. The virtual system model: A scalable approach to organizing large systems. Technical Report 90-05-01, Univ. of Washington CS Department, May 1990. Thesis Proposal.

[NIS91] Ansi z39.50 version 2. National Information Standards Organization, Bethesda, Maryland, January 1991. Second Draft.

[OCH+85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2bsd file system. In *Symposium on Operating System Principles*, pages 15–24. ACM, December 1985.

[Pen90] Jan-Simon Pendry. Amd — an automounter. Department of Computing, Imperial College, London, May 1990.

[Pet88] Larry Peterson. The Profile Naming Service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. UK UUG proceedings, 1990.

[PW90] Jan-Simon Pendry and Nick Williams. Amd: The 4.4 BSD automounter reference manual, December 1990. Documentation for software revision 5.3 Alpha.

[Roc85] Marc J. Rochkind. *Advanced UNIX Programming.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[RT74] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Comm. ACM*, 17(7):365–375, July 1974.

[Sal83] Gerard Salton. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[SC88] Sun Corporation. The Network Software Environment. Technical report, Sun Computer Corporation, Mountain View, California, 1988.

[Sch89] Michael F. Schwartz. The Networked Resource Discovery Project. In *Proceedings of the IFIP XI World Congress*, pages 827–832. IFIP, August 1989.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX Association 1985 Summer Conference Proceedings*, pages 119–130, 1985.

[SK86] C. Stanfill and B. Kahle. Parallel Free-Text Search on the Connection Machine System. *Comm. ACM*, pages 1229–1239, December 1986.

[Sta87] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, March 1987. Sixth Edition, Version 18.

[Ste91] Richard Marlon Stein. Browsing through terabytes: Wide-area information servers open a new frontier in personal and corporate information services. *Byte*, pages 157–164, May 1991.

[Sun88] Sun Microsystems, Sunnyvale, California. *Network Programming*, May 1988. Part Number 800-1779-10.

[Sun89] NFS: Network file system protocol specification. Sun Microsystems, Network Working Group, Request for Comments (RFC 1094), March 1989. Version 2.

[Tec90] ON Technology. ON Technology, Inc. announces On Location for the Apple Macintosh computer. News Release ON Technology, Inc., Cambridge, Massachusetts, January 1990.

[Ver90] Verity. Topic. Product description, Verity, Mountain View, California, 1990.

[Wei] Peter Weinberger. CBT Program documentation. Bell Laboratories.

[WO88] Brent B. Welch and John K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *USENIX Association 1988 Summer Conference Proceedings*, pages 37–49, San Francisco, California, June 1988.