12. Naming

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

David Wheeler

It's not turtles all the way down.

Anonymous

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 that are sequences of simple names. 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.

- 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, collection, 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 $N = \text{String}$ for a simple name and $PN = \text{SEQ} \ N$ 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:

<table>
<thead>
<tr>
<th>Unix path name</th>
<th>Internet mail address</th>
<th>IP network address (fixed length)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/udir/lampson/pocs/handouts/12</td>
<td><a href="mailto:lampson@mediaone.net">lampson@mediaone.net</a></td>
<td>16.23.5.193</td>
</tr>
</tbody>
</table>

In this handout we will write path names as Unix file names, rather than as the sequence of simple name constructors that would be correct Spec. Thus $a/b/c/1026$ instead of $PN{a, \text{"b"}, \text{"c"}, 1026}$.

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 implement 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 (currently undefined) name faculty.eecs@mit.edu and the query “the faculty members in MIT’s EECS department”. The former name, if defined at all, 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 will 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 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 \( M = A \rightarrow D \). Here we take \( A = PN \), replace \( D \) with \( V \) (for value), and replace \( M \) with \( D \) (for directory). This kind of memory differs from the byte-addressable memory of a computer in several ways:

- 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 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 EXPORT Read, Write, Remove, Enum, Next, Rename =

 TYPE N = String % Name
 PN = SEQ N % Path Name
 D = PN -> V % Directory

 VAR d := D{} % the state

 Here are the familiar Read and Write procedures; Read raises error if pn is undefined, for consistency with later specs. Note that 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 this Remove does not erase the values of longer names that start with PN.

 FUNC Read(pn) -> V RAISES {error} = RET d(pn) [*] RAISE error
 APROC Write(pn, v) = << d(pn) := v >>
 APROC Remove(pn) = << d := d{pn -> } >>

 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 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.

 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 and finishing 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.

 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.

 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 and final states. Read x >= y as ”x is a suffix of y”.

 APROC Rename(from: PN, to: PN) RAISES {error} = <<
 IF VAR d0 |
 (ALL x: PN, y: PN | { x >= from 
 [*] x = to + y \ d!(from + y) => d0(x) = d(from + y)
 [*] - x >= to \ d!x => d0(x) = d(x)
 [*] - d0!x } )
 => d := d0
 [*] 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 isDir, with a separate procedure for making an empty directory. We present this spec without extensive comments, marking with boxes the changes from MemNames0. To enforce the new rule every routine can now raise error, and Remove erases the whole sub-tree.
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.

The MemNames spec is basically the same as the simple Mem spec. Complication 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

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 \( PN \to V \) 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 a different implementation 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 \( N \to Z \), where \( Z = (D + V) \) as before, and have a state variable \( s \) 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.\(^1\)

We solve this problem with another level of indirection, so that the value of a directory name is \( a/b/c \), and similarly for \( a/b/c/d \), etc., but some kind of reference or pointer to a name’ for a directory. We use the name \( DD \) for the actual function \( N \to Z \) and introduce a state variable \( s \) that holds all the \( DD \) values; its type is \( D \to DD \).

\[ \text{APROC } Read(pn) \to V \text{ RAISES (error) = d(pn) IS V } \rightarrow \text{ RET } d(pn) \text{ [*] RAISE error} \]

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.

The MemNames spec is basically the same as the simple Mem spec. Complication 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

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 \( PN \to V \) 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 a different implementation 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 \( N \to Z \), where \( Z = (D + V) \) as before, and have a state variable \( s \) 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.\(^1\)

We solve this problem with another level of indirection, so that the value of a directory name is \( a/b/c \), and similarly for \( a/b/c/d \), etc., but some kind of reference or pointer to a \( N \to Z \). This reference is an ‘internal name’ for a directory. We use the name \( D \) for the actual function \( N \to Z \) and introduce a state variable \( s \) that holds all the \( D \) values; its type is \( D \to DD \). A \( D \) is just the internal name of a directory, that is, an index into \( s \). We take \( D = \text{Int} \) for simplicity, but any type with enough values would do: in Unix \( D = \text{INO} \). 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 \( s \).

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 \( PN/a/... \), \( PN/a/.. \), 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

\[^1\]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, \textit{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.
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**

EXPORT Read, Write, MakeDir, Remove, Enum, Rename =

**TYPE D = Int**  % just an internal name

**Z = (V + D)**  % the value of a name

**DD = N -> Z**

**CONST**

root : D := 0

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

**FUNC Get(d, pn) -> Z RAISES {error} = <<
IF VAR x: SEQ D | [\*] RAISE error  FI >>

CONSTRUCTION GetD(root, pn) -> D RAISES {error} = VAR z := Get(d, pn) |

IF z IS V => RET z [\*] RAISE error FI

**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 [\*] RAISE 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 MakeDir(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),

td := GetD(root, to.reml) |

s(td) := s(td)(z => 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} = VAR z := d | % Return the value of pn looked up starting at z.

DO pn # {} => VAR n := pn.head |

IF z IS D => RAISE error [\*] RAISE error FI

OD; RET z

**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) := s(d) z => s(d) from.last; RET d >>

**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 x of directories from a to the returned value, such that the components of pn select the corresponding components of x; if there’s no such sequence, raise error.

**FUNC Get(d, pn) -> Z RAISES {error} = <<
IF VAR x: SEQ D |

x.size = pn.size /

\( x(0) = d \)

\( (\text{ALL } i : \text{IN } x \text{.reml} . \text{dom} | x(i+1) = s(x(i)) (pn(i)) ) \) =>

RET s(x.last)(pn.last) [\*] RAISE error FI >>

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

**MemNames0 = \{ pn | G(pn) IS V => G(pn) [\*] G(pn) IS D => isDir \}**

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.

The abstraction function from MemNames0 to ObjNames0 is left as an exercise.

**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 N -> Z functions in z may be very different from the Write and MakeDir operations of ObjNames0. 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 in context a 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 specification for a naming system that allows for both symbolic links and extensible implementation of directories.

A ObjNames0, D has get and set methods to allow for different implementations, though for now we don’t take any advantage of this, but use the fixed implementations 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 too
tricky to program with undefined in this general setting; this means that $Z$ needs Null as an extra case.  

Link is another case of $Z$, 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 $a$ 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**

**EXPORT** ...

**TYPE** D = Int $\quad$ % Just an internal name

\[ \begin{align*} 
\text{Link} & = [d: (D + \text{Null}), \ pn] \\
\text{Z} & = (V + D + \text{Link} + \text{Null}) \\
\text{DD} & = N -> Z \\
\text{root} & : D := 0 \\
\text{null} & : D := 0 \\
\text{s} & := (D -> DD){}{root -> DD{}} \\
\end{align*} \]

\[ \begin{align*} 
\text{APROC GetFromS}(d, n) & -> Z = \\
& \begin{cases} 
\text{d}.get(n) & \text{if } z = \text{null} \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[ \begin{align*} 
\text{APROC SetInS}(d, n, z) & = \\
\text{d}.set(n, z)
\end{align*} \]

**PROC GetDN(pn, followLastLink: Bool) -> (D, N) RAISES {error} =**

\[ \begin{align*} 
\text{PROC GetD(root, pn)} & -> Z = \\
\text{DO} & \text{IF } \text{pn} = {} \Rightarrow \text{RAISE error} \\
& \text{IF } \text{z IS D} \Rightarrow \text{RET z AS D} \\
& \text{IF } \text{z IS D} \Rightarrow \text{FOLLOW Link(z, n)} \\
& \text{IF } \text{z IS D} \Rightarrow \text{RET (d, n)}
\end{align*} \]

**PROC Read(pn) -> V RAISES {error} =**

\[ \begin{align*} 
\text{PROC Enum(pn) -> SET N RAISES {error} =}
\end{align*} \]

**PROC MakeDir(pn) RAISES {error} =**

**PROC Rename(from: PN, to: PN) RAISES {error} =**

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

\[ \begin{align*} 
\text{PROC GetD(root, pn.reml), pn.last} & \Rightarrow \\
\text{GetD(root, pn.reml)} & \Rightarrow \text{null}
\end{align*} \]

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} =**

**PROC SetInS (d, n, z) =**
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 implementation.

To do this, we change the type to:

\[
\text{TYPE D } = \{ \text{get: APROC (n) \rightarrow Z, set: PROC (n, z) RAISES \{error\}} \}
\]

\[
\text{DR } = \text{ Int} \quad \text{% what D used to be; R for reference}
\]

We also need to change the state to:

\[
\text{CONST root } := \text{NewD()} \quad \text{s } := \text{(DR \rightarrow DD)\{root \rightarrow DD\}} \quad \text{% initially empty root}
\]

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 \( \text{get:=s(dr)} \) 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.

\[
\text{APROC NewD()} \rightarrow D = \langle\langle \text{VAR dr } | \sim s!dr = \rangle
\]
\[
\quad s(dr) := DD{}; \quad \text{RET D\{get := (\n | s(dr)\(n)\},
\]
\[
\quad \text{set := (PROC (n, z) = IF } z \neq \text{nil } \Rightarrow s(dr)\(n) := z
\]
\[
\quad \text{[\*] s(dr) := s(dr)\{n \rightarrow \}} \text{ FI}\}
\]

\[
\text{PROC SetErr(n, z) RAISES \{error\} = RAISE error}
\]

\% 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 in a name space,
- ‘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 \( (d', pn') \), 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. 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 hard link is an inode number, which you can think of as a direct pointer to a file. 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. If we do this repeatedly, writing d1 + d2 + d3, we get the effect of a ‘search path’ that looks at d1 first, then d2, and finally d3 (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; a search path must be implemented 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}

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 an efficient implementation.

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}

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 implementation 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 is it 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 path’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, MakeDir, 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 ObjNames specs made all the update operations atomic. For an implementation 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:

1. In the Read, Get(root, a) = d1
2. Rename(/a, /b) makes /a undefined and d1 the value of /b
3. Write(/b/x, 3) makes 3 the value of x in d1
4. 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 level 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 $\text{Sync}$. In return, we can now have much more availability in the implementation, since both a read and a write can be done to a single replica. This means that if you do $\text{Write}(\text{/a, 3})$ and immediately read $a$, you may not get 3 because the $\text{Read}$ might use a different replica that hasn’t seen the $\text{Write}$ yet. Only $\text{Sync}$ requires communication among the replicas.

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

\[
\text{TYPE DD} = N \rightarrow \text{SEQ } Z
\]

\[
\text{APROC } \text{GetFromS}(d, n) \rightarrow Z = \langle  \text{d.get(n)} \rangle
\]

\[
\text{PROC } \text{SetToS}(d, n, z) = \langle \text{d.set(n, z)} \rangle
\]

\[
\text{PROC } \text{Sync}(pn) \text{ RAISES } \{\text{error}\} = \langle \text{true} \rangle
\]

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). 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 $\text{/bin}$ and $\text{/etc}$ 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.

\[\text{Gray and Reuter, Transaction Processing, Morgan Kaufmann, 1993, pp 768-786.}\]
Search some ad-hoc set of places in the hope of finding a replica. This search is often called a ‘broadcast’.

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 implemented 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**

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**

Not restricted to relational databases.

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

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

**Table (relational database)**

Table example:

<table>
<thead>
<tr>
<th>ID no (key)</th>
<th>Name</th>
<th>Salary</th>
<th>Married?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1432</td>
<td>Smith</td>
<td>21,000</td>
<td>Yes</td>
</tr>
<tr>
<td>44563</td>
<td>Jones</td>
<td>35,000</td>
<td>No</td>
</tr>
<tr>
<td>8456</td>
<td>Brown</td>
<td>17,000</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

- #44563/Name = Jones
- Name/#44563 = Jones
- Smith, cat name/* yields
- Jones

5 Pike et al., The use of name spaces in Plan 9, ACM Operating Systems Review 27, 2, Apr. 1993, pp 72-76.


**Network naming**

Example: `theory.lcs.mit.edu`

Distributed implementation. 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 distribution lists.

**SNMP**

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.

7 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.

Hierarchical network addresses


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


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
