Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 12 March 3, 1997

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, 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/lynch`.
>
> 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@crl.dec.com`	Internet mail address. The path name is `{"com", "dec", "crl", "lampson"}`
`16.23.5.193`	IP network address (fixed length)

In this handout we will 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 implement them by addresses, and paths at lower levels by routes. Examples:

```
src.dec.com -> 16.23.114.5 -> sequence of [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@crl.dec.com`, 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.

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 -> 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 is 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`.

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

FUNC Next(pn) -> PN    =
    << RET {pn' | d!pn' /\ pn.LexLE(pn', N.”<=”)}.min [*] RET {} >>
```

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.

```
APROC Rename(from: PN, to: PN) RAISES {error} = <<
    IF VAR d0 |
         (ALL x: PN, y: PN | (    x >= from                  => ~ d0!x
                             [*] 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.

```
MODULE MemNames EXPORT Read, Write, MakeDir, 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) = << DO VAR pn' :IN d.dom | (pn <= pn') => d := d{pn' -> } OD >>
% Erase everything that has pn as a prefix.

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 + Dir) RAISES {error} =
     << pn # {} /\ d(pn.reml) IS Dir => 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.

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.

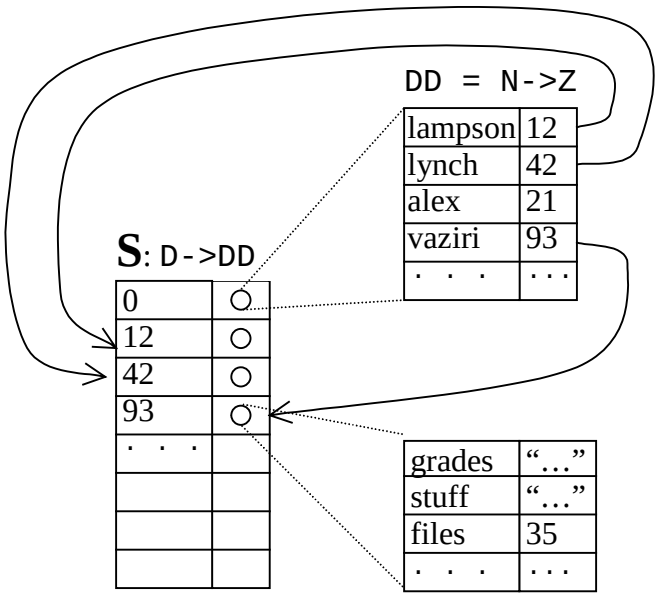
Name space as graph of directory objects

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 -> 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.

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.[1]

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

[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, *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.

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)
     DD         =  N -> Z

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

FUNC Read(pn) -> V RAISES {error} = VAR z | 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 >>
```

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) | s(d) := s(d){pn.last -> } >>

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

The remaining routines are internal. `Get(d, pn)` 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 /\ s(z)!n => z := s(z)(n); pn := pn.tail [*] 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)(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 `x` of directories from `d` 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
          /\ ( ALL i :IN x.reml.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

```
MemNames.d = (\ 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 MemNames to ObjNames0 is left as an exercise.

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 s 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 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 reflects all these considerations. It is rather complicated, but the complexity is the result of the many demands placed on it. A ObjNames.D has get and set methods to allow for different implementations, though for now we fix them; the section on coherence below explains why get is a procedure rather than a function. 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 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 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 means the containing D
     Z          =  (V + D + Link + Null)                % nil means undefined
     DD         =  N -> Z

VAR  root       :  D := 0                               % Constant
     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 | 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 because get isn't a function
     VAR d := GetD(root, pn), ns: SET N := {}, z |
          DO VAR n | << z := d.get(n); ~ n IN ns /\ n # nil => ns := ns + {n} >> OD;
          RET ns

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

PROC MakeDir(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

```
RET (GetD(root, pn.reml), pn.last)
```

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

FUNC 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 implementation.

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
```

We also need to change the state to:

```
VAR  root       := NewD()                               % Constant
    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 assigned to `get` and `set` have the same bodies as the `GetFromS` and `SetInS` routines. 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 the 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
```

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, give it the interface of a `D`. 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 write the value of a name just like 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 type of the second argument 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 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, 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[2]. 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; 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.

```
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. Any predicate could be used for this purpose, so we get:

```
FUNC Filter(d, p: (D, N) -> Bool) -> D =
     RET D{get := ( \ n | (p(d, n) => d.get(n)) ), 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 last section.

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.

[2] See section 9 of the Spec reference manual.

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`, `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 `ObjNames0` 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:

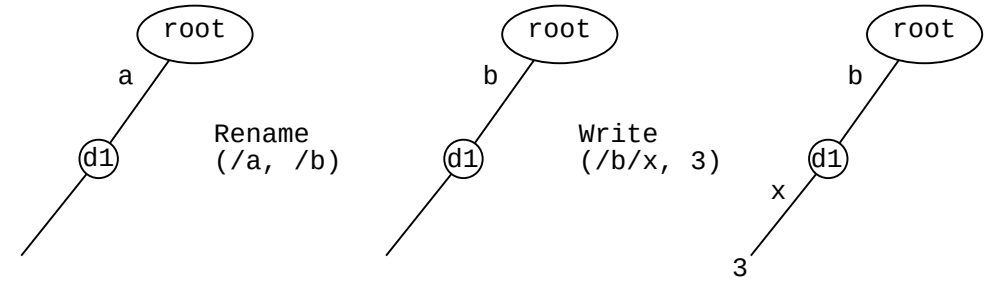
initially `/a` is the directory `d1` and `/b` is undefined;

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`:

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 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. It is possible to make tradeoffs by generalizing the notion of majority to 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 doing 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 implementation, 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. 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).[3] Network management calls this a 'management information

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

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.

Resource control. Writes often consume resources that are expensive or in fixed supply. 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

So far we have ducked the question of how the `root` is represented, or the `D` in a link which 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 is highly implementation-dependent. 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 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 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 replica 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 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[4]	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[5]	Not restricted to relational databases. Free-text indexing: `~lampson/Mail/inbox/(word="compiler")` Program cross-reference: `/project/sources/(calls="DeleteFile")`

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

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          Name/#44563 = Jones
```

The second way, `cat Name/*` yields

```
Smith                        Jones
```

Network naming[6]

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: `lynch@theory.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[7]

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. Its 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).

---

[5] Gifford et al., Semantic file systems, *Proc. 13th ACM Symposium on Operating System Principles*, Oct. 1991;, pp 16-25 (handout 13).
[6] 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.
[7] M. Rose, *The Simple Book*, Prentice-Hall, 1990.

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.	
Hierarchical network addresses[8]	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[9]	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') 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`	

[8] R. Perlman, *Connections*, Prentice-Hall, 1993.
[9] Andrew Birrell et al., Network objects, *Proc. 14th ACM Symporium on Operating Systems Principles*, Asheville, NC, Dec. 1993 (handout 24).