6.826 Principles of Computer Systems	Fall Semester, 1992

Lecture 10: October 19, 1992

Lecturer: Butler Lampson — *Scribe: Yasuhiro Endo*[1]

1 Handouts

Handouts 27 (*A Simple Approach to Specifying Concurrent Systems*), Notes on Lecture 5 and 7.

2 Overview

In this lecture, name related issues, atomicity, aliases, and links in the filesystem are discussed.

3 What do we mean by name?

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

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

```
TYPE
    N = STRING              % Name
    PN = SEQ[N]             % Pathname
```

Note that PN is an abstraction of hierarchical name. Though they may use different separator to separate two names in the pathname, many of the file naming schemes that we know are the implementations of this abstraction.(e.g. unix, VMS, DOS).

4 Looking up names

With this hierarchical name structure, we can consider how names can be looked up. Although not currently implemented, the following naming scheme can be used to identify any file in the whole world:

/edu/mit/lcs/photon/aaa/bbb/ccc

Here, /edu/mit/lcs identifies the machine, and /aaa/bbb/ccc identifies a file in the machine. Note that we already have something that is less sophisticated than this working today. Mailing address

[1]These notes borrow some lines from the notes prepared by Kah-Kay Sung last Fall.

is one of the example, and weihl@photon.lcs.mit.edu identifies a single person from all the people in the world.

What we have discussed so far in this section is how to identify an object using absolute address(path) of the object, and we all know that this is not the only way we specify an object. We often specify objects by its relative position based on current context. For example, if someone asks you where the Albany Street is in front of MIT Coop, you are pretty sure that she is not asking you to tell her where the Albany Street is in California.

The same thing applies to file system as well, the aforementioned file can be identified by name ccc if your current host is photon.lcs.mit.edu and current directory(context) is /aaa/bbb. One should realize that the type we defined for the name is not sufficient to accommodate the use of relative path. We will now add C to hold current context.

```
TYPE
    N = STRING            % Name
    PN = SEQ[N]           % Pathname
    C = PN->V             % Context(pathname to value mapping)
```

Given the new definition, following specifications can be written:

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

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

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

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

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

We do so by modifying the definition of C as follows:

```
TYPE
    C = N->(V+C)        % Context(name to value or context mapping)
```

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

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

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

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

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

5 Atomicity

Now, consider the following scenario:

```
Process 1                                    Process 2
_________                                    __________

read(/a/x)                                   rename(/a, /c)
                                             write(/a/x, V)
```

There is no doubt that if all operations are atomic, the read should fail. This is a somewhat unrealistic assumption, and strange things can happen in the real world where lookups are often non-atomic. Under certain conditions, a lookup like the one above may indeed succeed. For an example, look at the sequence of events in Figure 1.

We must introduce some kind of locking mechanism to ensure that the actions that implementation take are "sensible." One might consider locking all the node that GetDir has visited, but this scheme is not ideal, for it is likely to force many of other threads performing a lookup to wait while one thread locks up the whole tree by locking the highest node. This performance bottleneck can be made less serious by using read/write lock, but the performance penalty still outweighs the benefit.

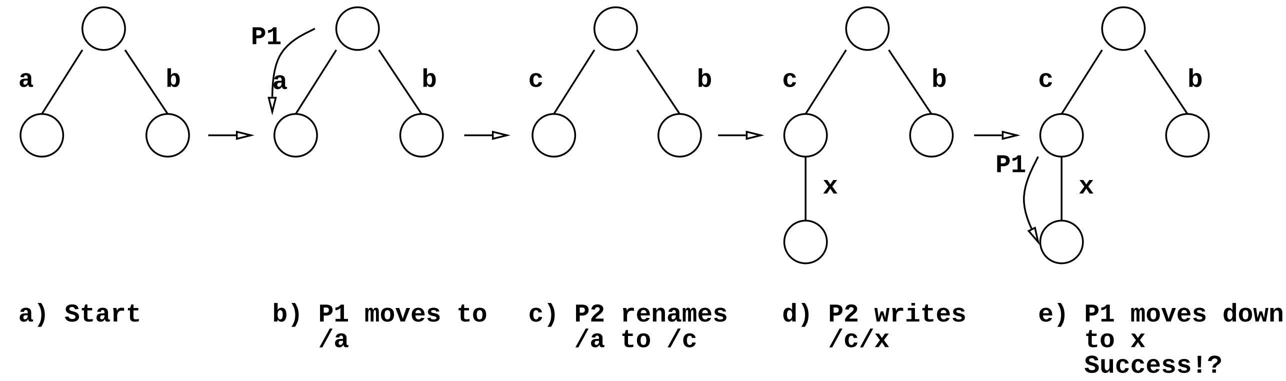

Figure 1: A case when read(/a/x) succeeds erroneously.

6 Lock Coupling

In the above example, process 1 ended up having an “inconsistent” result because process 2 could access file system objects below (i.e., descendants of) the current node of process 1. One way of overcoming this problem is for processes to use a technique called *lock coupling* that prevents other processes like process 2 from “slipping by”. A *lock* is a mutex associated with a node of a hierarchical directory structure. To perform *lock coupling*, a process traverses the directory path as follows: while holding the lock for the directory last read, it locks the child directory to be read, and only then relinquishes the lock for the parent. Locking starts from the root node and leapfrogs along the path as it is being traversed (see Figure 2).

7 Aliases and Links

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

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

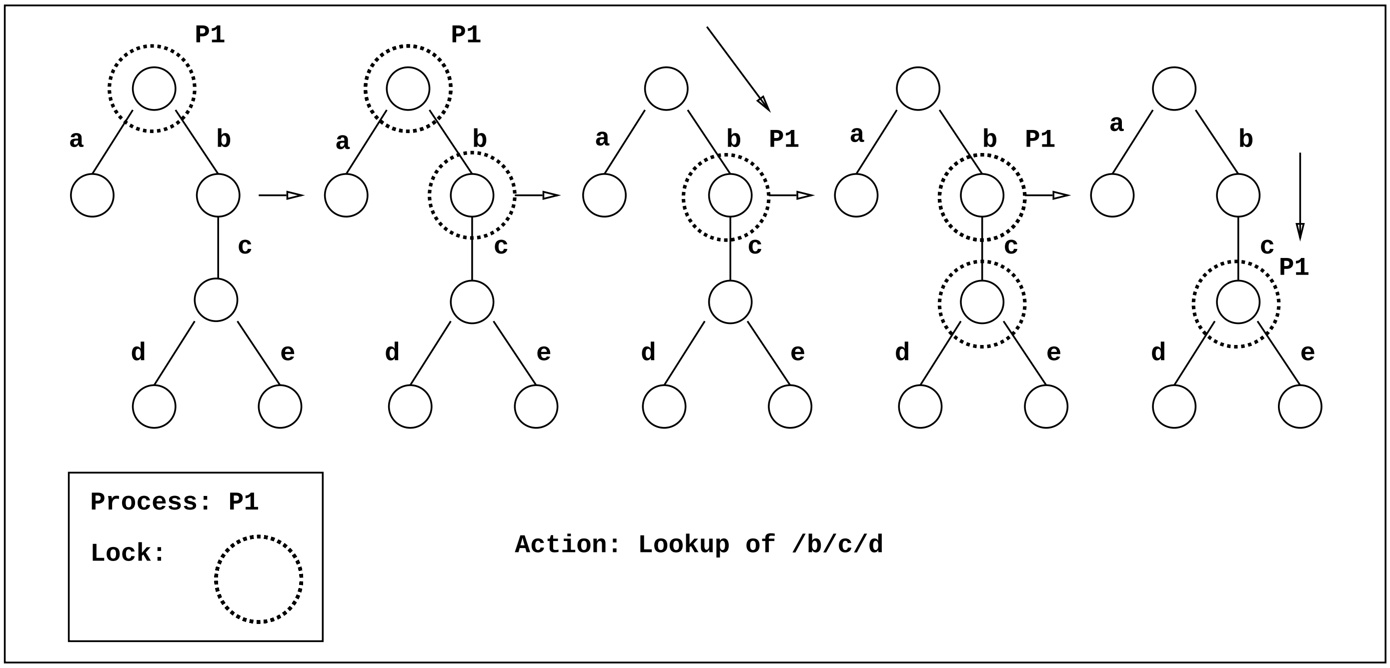

Figure 2: Snapshots of lock coupling along search path for `/b/c/d`.

8 Example Application of Hierarchical Filesystem - Relational DataBase

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

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

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

9 Enumeration Operation

We have not discussed how to enumerate files in a directory. One naive approach may be to define a function that returns a set of names, but it is often so that the set is too large to be returned in one shot. Instead of this naive approach, we define the following operation:

```
% given C and pathname, returns entry after pn in C, in depth-first order.
FUNCTION Next(C, pn) -> pn
```

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