6.826 Principles of Computer Systems	**Fall Semester, 1993**

Lecture 11: October 20, 1993

Lecturer: Butler Lampson *Scribes: Misha Davidson and Roy Seto*[1]

1 Handouts

Handout 28 "Semantic File Systems" by David Gifford *et al.*

Handout 29 "Concurrent Disk Reads"

2 Overview

This lecture discusses hierarchical name spaces and concurrency control, especially for hierarchical file systems.

3 Hierarchical Name Spaces

A hierarchical name space is one with the following recursive structure:

```
TYPE
    D = N -> D + V  % directory D, name N, value V.
```

Some examples of systems with hierarchical name spaces follow.

3.1 The Internet Domain Name System

The Internet Domain Name system is a large-scale example of a hierarchical name space. Names are of the form "`weihl@lcs.mit.edu`." The hierarchical arrangement makes this scheme similar to the standard filesystem directory structure. However, there are several differences:

- Different levels of the hierarchy are implemented by different name servers.
- Many nodes are replicated. Since updates of replicated nodes are not atomic, this can cause inconsistencies: for example, a read of an old value in the hierarchy can occur after the new value has been both written and read elsewhere. However, this is considered acceptable for this application because the name space does not change very quickly, and users don't depend on its correctness to the same extent as applications built on top of filesystems.

[1] These notes are based on those of Yasuhiro Endo last fall, who in turn acknowledges Kah-Kay Sung from Fall 1991, and on the notes taken by Ken Mackenzie, who acknowledges Henry Minsky

- Nodes (*i.e.*, directories) in filesystems typically hold links to their children and parents. Domain name server nodes also have to be able to reach their siblings too, to provide robustness in case the parent stops being available.

3.2 Network-wide pointers

Another possible hierarchical naming scheme could be constructed to allow pointers to any individual word of any process on the Internet.

One method would be to concatenate the (network-unique) process ID and the word address. However, this isn't very workable because it's too flat.

It would be easier if we added more levels to the hierarchy. Imagine the following levels:

network address *e.g.*, 16.143.2.12

process ID *e.g.*, 2416

process uniquifier or tag *e.g.*, 13642895311
(We need this because machines can fail and come back up again, recycling their process ID's. Adding this field makes it unlikely that this recycling of PID's will cause a false match. It plays no role in the search, serving only as a correctness check.)

internal address *e.g.*, 100
In many cases, this is not allowed to be an arbitrary address in the process. Instead, it is interpreted in a conservative way (for example, as in index into a table of externally visible vectors). This prevents other processes from doing nonsensical things. This kind of usage is often called an "RPC binding."

The "RPC binding" level of this hierarchy can be manipulated to increase efficiency. For example, suppose we had two communicating processes, and we wanted one to send some text to the other, to be printed in a Times Roman font. The thought of sending off the entire font in the message is certainly unappealing. Instead, we could use an "abbreviation" that would represent Times Roman with some short identifier. This approach would trade some robustness for efficiency. Note that the abbreviation approach is not stateless, because communicating processes need to remember the bindings for abbreviations. We could also put even more state into the scheme by adding operations such as "change to Times Roman font." The cost of this approach would be more synchronization problems.

Another example of this abbreviation approach is used to implement complicated protocols (with correspondingly complex network message headers) over slow lines. The nodes at the ends of the lines can agree to abbreviate often-seen long network headers with shorter ones to improve efficiency.

3.3 Network management

Another example of a hierarchical namespace is the "simple network management protocol" (SNMP) in the Internet protocol suite. The purpose of SNMP is to define a standardized way to access different

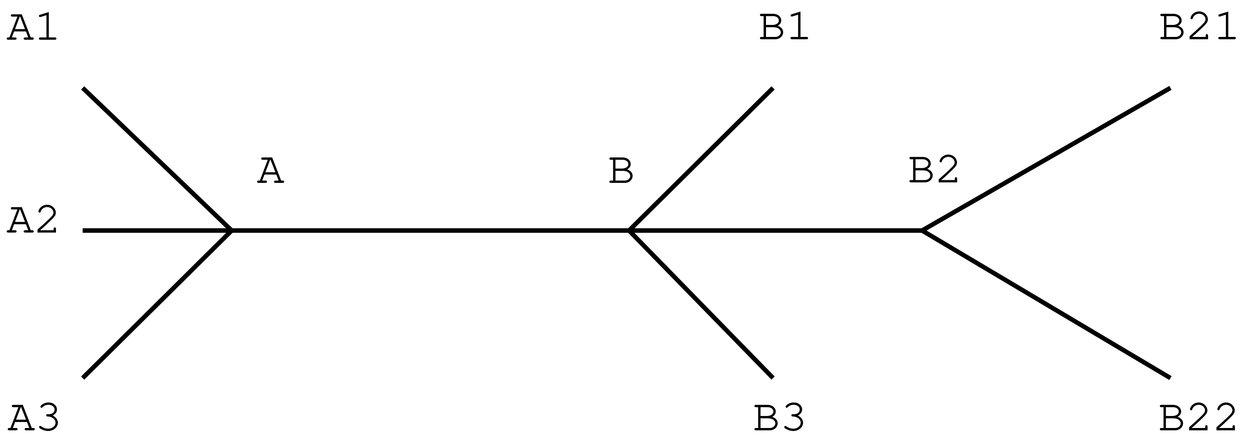

Figure 1: Routing through a network

boxes on the net. It is implemented using tables called "management information blocks" (MIB's). Each row of the MIB is associated with a different network virtual circuit. Each column describes a different attribute, such as the near end, the far end, or the bandwidth. The MIB is a hierarchical namespace because cells of the table can be MIB's too.

As a side note, SNMP is not very good for renaming and change operations because it would distribute too much authority across the network. However, it is very important for its read and enumerate operations.

3.4 Multiplexed channels

Multiplexed channels serve as another example of hierarchical naming. Looking at the network in Figure 1, we can consider the route through the nodes `A`, `B`, `B2`, and `B21` as one channel, specified by that sequence of nodenames; the sequence can be appended to packets to indicate a route from the source to the destination. This is similar to a hierarchical directory structure.

3.5 Input/output addressing

Typically, device addressing on the I/O bus is implemented with memory-mapped bus addresses, with a bit in the address to indicate that we're selecting the I/O space instead of the normal memory address space. In itself, this is a fairly flat structure; however, it's often desirable to retain backward compatibility with older I/O bus standards, since an investment may have been made in devices for the older busses. We can retain compatibility by plugging the older buses into device slots in the newer buses. As this bridging continues, I/O addressing will become increasingly hierarchical.

For example, suppose we want our machine's native I/O bus to support SCSI devices. We can simply plug the SCSI bus into one slot of the native bus. We could also plug an "XMI" bus into the native bus using a bridge; it's conceivable that we'd use second bridge to tie the XMI bus to the even older Unibus.

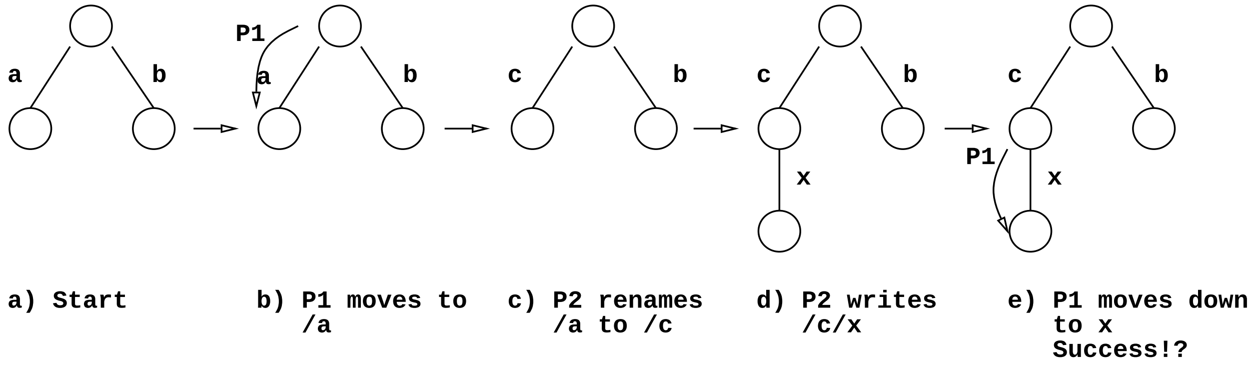

Figure 2: A case when `read(/a/x)` succeeds erroneously.

A complexity arises with this sort of ad-hoc hierarchical structure, though. Recall that the operations we typically want on a namespace include `Read`, `Write`, `Rename`, and `Enumerate`. How would we implement the enumerate operation with this type of hierarchical arrangement? One method would be to do "autoconfiguration," which talks to the devices connected to the bus and asks them to report their identities.

4 Concurrency Issues in Implementation

4.1 Atomicity

Let us consider an implementation of a hierarchical file system. Now, consider the following scenario:

P1	P2
`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 atomicity is a somewhat unrealistic assumption, however, 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 2.

We must introduce some kind of locking mechanism to ensure that the actions that implementations take are "sensible."

4.1.1 Lock Coupling

In the previous example, `P1` ended up having an "inconsistent" result because `P2` could access file system objects below (*i.e.*, descendants of) the current node of `P1`. One way of overcoming this problem is for processes to use a technique called *lock coupling* that prevents other processes like `P2` from "slipping by" `P1`. 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

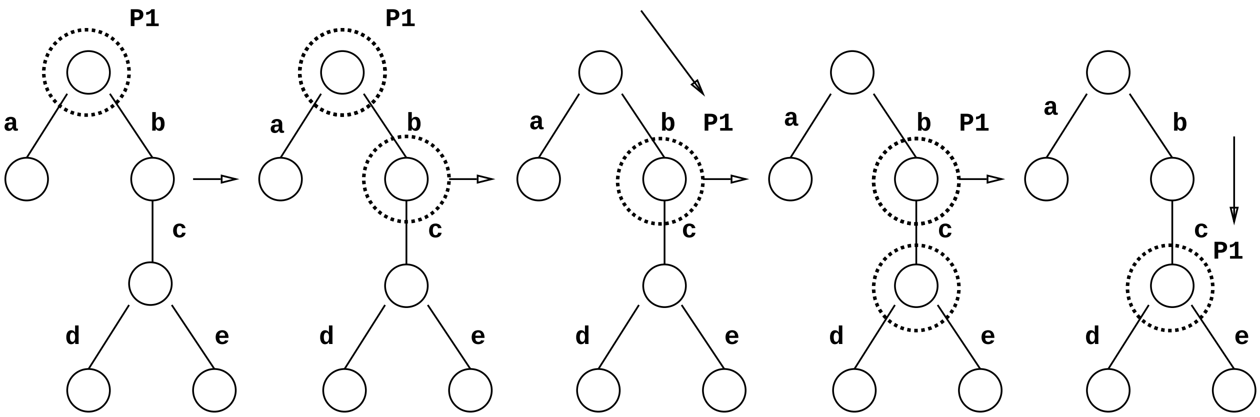

Figure 3: Snapshots of lock coupling along search path for `/b/c/d`. Dashed circles represent locks held by `P1`

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 3).

While lock coupling avoids the problem from Figure 2, it can cause a deadlock if there are cycles in the order that locks are acquired (e.g. Unix's "`..`" reverse links).[2]

4.1.2 Two-Phase Locking

Two Phase Locking is a technique commonly used in database systems. The ideas behind this method are:

- Acquire all the locks before releasing any; this guarantees atomicity.
- Acquire locks in order, according to some global partial order on the locks; this eliminates any possibility of deadlock.

This allows all nodes in a path to be locked safely before any operations take place, and avoids deadlock problems. It can, however, introduce long (but finite) delays as processes hold some locks while waiting for others.

Note also that, though a useful (and used) scheme, it's not always easy (or possible) in a directory system to find an ordering on the locks that both works and that provides any advantage over a single, global lock. (Consider what happens, for example, if your ordering requires you to always lock the root of the tree before each operation.)

4.1.3 Optimistic Locking

Optimistic locking is a variation on two-phase locking which can reduce the time that locks need to be held.

[2]BSD Unix apparently solves this by abandoning the lock coupling when traversing reverse links.

- Traverse the tree using some low level mechanism, and find all the directories you would have needed to lock.
- Acquire these locks in accordance with some ordering.
- Repeat the exploration. If it fails, release all the locks and retry.
- Otherwise perform the operation and release the locks.

Optimistic locking will never deadlock. However, in some circumstances it can produce a livelock (where, without good scheduling, every process might starve because no process will ever be able to acquire all the necessary locks.)

4.1.4 Exponential Back-Off

If we are using optimistic locking and do not want to build an elaborate scheduler, we can use *exponential back-off*. This technique is used in Ethernet to make sure that only one host is transmitting at every point in time. If two or more hosts try sending messages at the same time, there is a detectable collision, and the messages are discarded. (Note that the collisions must be detectable, or this scheme will be unimplementable.) The hosts then retry sending after some random delay. If there is a collision again, the hosts keep on retrying, increasing the maximum possible delay every time there is a collision. Note that if delay is increased by a constant factor each time, it grows exponentially. The idea is to space events far enough apart, so that they do not interfere with each other. In fact, if the sending period is much greater than the occupancy time, the chances of successfully transmitting a message are good. If every host could figure out the load on the system, then the optimal sending period would be $t = (1/load)^r$, where r is the number of retries.

5 Concurrent Disk Read Example

We want the implementation of the `ConcurrentDisk` module (Handout 29) to support multiple threads reading and writing simultaneously. This could be accomplished by serializing all of the operations by making all operations atomic, which would basically be the same as the `BufferedDisk` implementation. But this makes threads wishing to access independent cached disk blocks wait needlessly for other's slow disk operations or block-copies to complete.

What benefit would be gained here by using concurrency? Consider the cases of cache-lookup and block-copy, and a single CPU system. If we have two or more CPUs, or a CPU and disk drives, then it is possible to make real gains in performance, if several tasks can actually be executed simultaneously. In general, if we have several engines of computation which are capable of making progress simultaneously, then concurrency will be useful.

The natural thing to do is to make the system as concurrent as possible, for example by using fine grain locks. Why is it a bad idea? Too much concurrency is hard to manage. The compromise solution is to make the locks in the system as coarse as possible, but consistent with the above considerations.

Consider the cost of operations on a `Concurrent Disk`:

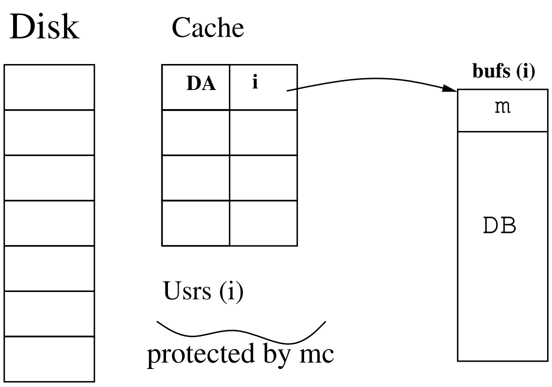

Figure 4: Locking for the ConcurrentDisk module

- *scalar* = cheap
- *block-copy* within main memory = medium
- *disk access* = expensive

Figure 4 shows the relation between locks and implementation. All operations working with the `map` will be scalar, so they can hold a common mutex `mc`. When reading or changing a content, lock individual locks `m` on buffers. For instance, to modify buffer `b` mutexes must be acquired in the following order: first `mc`, then `bs(b).m`.