6.826 Principles of Computer Systems	Fall Semester, 1993

Lecture 4: September 22, 1993

Lecturer: Butler Lampson — *Scribe: David Hau*[1]

1 Overview

Today's lecture finished the discussion of file representation begun last time. Then we discussed allocation and encoding/decoding issues. The lecture ended with a brief discussion of crash atomicity and its implementation using redo logs and version numbers.

2 File System Representations

Assume that there is one directory mapping path names to file structures:

```
Byte = INT
F = SEQ[Byte]
PN = STRING
Dir = PN -> F
```

The most straightforward way to represent the file would be to store the data contiguously in a sequence of data blocks:

```
F = [start: DA, blocks: INT, bytes: INT]
```

where `start` is the disk block address of the first block, `blocks` is the number of blocks occupied by the file, and `bytes` is the exact size of the file in bytes. This scheme above provides sequentiality, which is good for performance but makes storage allocation impractical. Such a method involves constant reorganization as files grow and shrink and leads to extensive fragmentation.

Another solution uses a hierarchical approach to represent the file system. A tree structured index is built whose internal nodes are index blocks containing pointers to either data blocks or other index blocks.

```
F = INDEX
INDEX = (DA + SEQ INDEX)
```

The problem with this method is that each data block has its own entry in the index, whether or not it is contiguous with other blocks. There is no obvious sequentiality. Even if the sequentiality is somehow extracted by noting that successive blocks are adjacent, this method wastes space by storing all block addresses.

[1] These notes are based on notes prepared by Anne Clark and Mitchell Charity in Fall 92

A third method for representing a file system is actually a generalization of the simple sequential one outlined above. This solution partitions the file into a number of contiguous series of disk blocks. On page 7 of Handout 9, the concept of *extents* is explained. Extents are references to contiguous sequences of disk blocks. `F` is a sequence of these extents that would be concatenated together to get the data.

```
EXTENT = [start: DA, blocks: INT]
F = [data: SEQ EXTENT, bytes: INT]
```

Problems with this implementation occur when F is big so that the sequence of extents does not fit into a block. This is usually solved by building a hierarchical structure like the one described earlier, except that the leaf nodes are variable sized extents of contiguous data blocks rather than individual blocks. The indexing nodes of the tree might themselves be variable sized sequence of blocks, but for the sake of simplicity, this is not employed in most file systems. Most implementations use B-trees to prevent the structure getting unbalanced and keep the index nodes fixed-size while the data is in variable-sized extents.

```
EXTENT = ([start: DA, blocks: INT] + SEQ EXTENT)
F = [extent, bytes: INT]
```

3 Allocation

Allocation involves the assigning of free space to extend a current file or form a new one. The free space available for allocation comprises all blocks that are not reachable from the directory, and is constantly changing as data is added and deleted on the disk. A file system needs some method of keeping track of what space is free to prevent valid data from being overwritten. The most basic way of allocating space would be to define reachable data (disk addresses already in the file structure) as

```
Reachable (da) = {pn | da IN blocks (dir (pn))}.
```

where `blocks` returns a set of the disk addresses of all blocks in the file.

When a free disk block is needed, adding da to a file involves searching for an unreachable block:

```
VAR da | ~ Reachable (da)
```

This scheme is very inefficient, because it explores the entire file system each time a new disk address is needed. Its form is useful though, because when a more optimized solution is found, we can ask if it is a restriction of above. If it is, the new method need not be proved correct separately. A better solution to the free space problem would be to define a variable `free` which tracks whether a block is free:

```
VAR free: DA -> Bool
```

The invariant to be maintained for correctness is:

```
free (da) ==>  ~ Reachable(da)
```

The standard way of representing free is through a SEQ[BOOL] (a bit table) with a bit per disk block. (If the table gets too big to fit in a block, the extent trick can be used.) Each update involves two writes: to the file itself and to the free table. The problem is that a crash could occur between these two atomic operations. If the system writes to a file and then crashes before it resets the free-bit in the table, the disk address could be allocated to another file on recovery, breaking our invariant.

On the other hand, the invariant permits updating the table before the file. If a crash occurs between the writes, the block is lost, but there is no loss of data. This can cause problems after a time if too many blocks are lost, so most systems now include a garbage collection system. Many systems also amortize the updates to bit table by taking a large batch of free block addresses in advance so that the system is not slowed down by having to do two writes at the time of each file update.

4 Encoding and Decoding

Programs are written using abstract types to define variables and relationships. This information must be preserved when the data is saved in a file. Encoding involves representing these types as sequences of bytes. Any system that is used to do this must (1) be total, that is, work for any type T, and (2) decoding the encoded form of a data structure must yield the same data structure:

```
(ALL t |  T.Decode (T.Encode (t)) = t)
```

The standard way to do this is to use *type-length-value* (TLV) encoding, in which the result of every `Encode` function starts with some sort of encoding of the type, followed by an encoding of its own length, followed by an arbitrary `D`.

5 Crash Recovery

A generic way is needed to deal with stable and volatile variables after a system has crashed. The stable variables are not affected by a crash, while the volatile ones are reset. We assume that writes to disk blocks are atomic and will either be completed or not done at all. Any disk write involving a number of blocks of data can be interrupted by a crash. This is not a bad assumption considering that a disk block write takes less than 1ms, which would not be affected by the system losing power, the most common form of crash. The basic idea is to build a system with both stable and volatile states:

```
S : state
ss : S := S$s0()                % initialization of the stable state
vs : S := S$s0()                % initialization of the volatile state
```

The system would then need to provide means of doing actions which would affect the volatile state, a procedure to commit the volatile state to the stable state, and a crash recovery procedure that copies the stable state into the volatile one.

```
A : Action = S -> (V,S)
```

```
Do (a : A) -> V = VAR v | (vs,v) := a (vs)
        RET v

Commit ( ) = ss := vs

Crash ( ) = vs := ss
```

The way this is implemented is by using *redo logs*. Each action is broken into a sequence of atomic actions which are written to the log atomically. In case of a system crash, the stable state is copied into the volatile state and the log is redone. This relies on the deterministic nature of the state machine (so that redoing the atomic actions always results in the same final state), and *log idempotence* of the atomic actions (so that redoing the log multiple times results in the same final state).

6 Redo Logs

Now let us go into the details of redo logs. This section is extracted from Fall 92 lecture 5 notes to provide some additional commentary. The ideas described in the following provide some useful insights, especially section 6.6, but some of the errors carried through from last year have not been fixed. Please refer to the revised Handout 31 whenever you have doubts.

6.1 Specification of Sequential Transactions

Abstract variables: vs volatile state - erased by crash ss stable state - unaffected by crash

Abstract operations:

```
Do(a)    = << vs := a(vs) >>
Commit   = << ss := vs >>
Crash    = << vs  := ss >>
```

Doing (applying) an action a changes the volatile stable vs. When you Commit, the stable state gets set, atomically, to the volatile state. Thus, one can do lots of individual things (actions), one at a time, and then commit, and ss will change atomically. Crash forgets the volatile state, and thus is, in some sense, the opposite of Commit. (For simplicity, this description leaves out both value returns and the "meaning" of actions.)

This is a completely general mechanism, given arbitrary actions a, of implementing crash atomic updates of arbitrary complexity.

6.2 Implementation Using Redo Log

The implementation idea is to keep the implementation's stable state in some stable place, and to have two logs - a volatile log and a stable log.

```
VAR ss    implementation's stable state
    vl    implementation's volatile log
    sl    implementation's stable log
```

A log is a sequence of updates. An update is such that it can be atomically applied to the stable state.

The abstraction function is:

```
AF  ss = ss + sl
    vs = ss + sl + vl        (this, not the handout, is correct)
```

The first just means the abstract stable state is implementation's stable state with the application of the implementation's stable log. (Here, the $+$ operator sequentially applies, to a state, each one of the updates in a log.)

How do we implement the log operations?

Do:

```
Do(a) =  vl := vl + Updates(a)
```

Do appends the new action a to the volatile log. Implementationally, it appends the updates of a, rather than a as one big chunk. As noted, the idea behind Updates is to break down the action into small enough things that each one can be done atomically. So if a is complex, it may have to be broken up quite a bit.

For example, if the stable state is implemented on the disk, each update must be small enough that it can implemented in a single disk write. If the action is to write a megabyte, and the disk blocks are a kilobyte, a will have to be broken down into a thousand updates. Of course, we can code the updates in some clever way, but logically there will have to be the thousand updates, because we can only write one disk block at a time atomically.

This implementation of Do correctly updates the volatile state, as:

```
vs = ss + sl + vl
```

Commit:

```
Commit =  sl := sl + vl
```

An interesting note is that this doesn't change the volatile log, so the volatile state is now the stable log followed by two copies of the volatile log.

```
vs := ss + (sl_original + vl) + vl
```

This doesn't matter given the *log idempotency* property (ie it is ok to repeat stuff at the end of the log).

There is more in the handout, but this is sufficient to do the job.

Crash:

```
Crash =  vl := {}
```

This corresponds to setting vs to ss:

```
vs_abstract = ss_impl + sl_impl + vl_impl
vs_abstract = ss_abstract       + vl_impl
vs_abstract = ss_abstract,   when vl_impl = {}
```

So we have (sortof) succeeded in implementing the redo log abstraction.

6.3 Redo Log Implementation - atomicity

But what is weird about this?

The implementations of Do, Commit, and Crash should be atomic, because the abstract versions are atomic, and we want everything to be straight forward.

So let's look at each again:

Do just changes volatile state, and there is no difficulty in making that atomic (given that there is no concurrency).

Commit is less obvious because it adds a potentially large thing to the stable log, which could involve a number of disk writes. So making this an atomic operation will not be completely trivial. But is much simpler than the original problem, which was to implement arbitrary actions. Now instead, we need to implement the process of appending an arbitrary large pile of update to the end of the stable log. There are many possible ways to do that.

One way is described in the problem set 2. The trick is, when writing something into the log, you arrange that the region of disk containing the log has the property that every disk block starts with a recognizable tag, set for example to 0. You then write blocks in in which the first byte is always set to 1. That allows you to tell how much writing has actually been done, when you come back to look at the log later. Thats the essential idea. It is part of the problem set to write down the implementation and abstraction function and so forth.

6.4 Redo Log Implementation - need explicit volatile state

Now we have to go back and correct a simplification we made earlier. We left out that Do might return something. An action not only modifies state, but may also return some value from the call.

The previous specification of Do:

```
Do(a) = << vs := a(vs) >>
```

is extended to have a return value v:

```
VAR v | << (vs,v) := a(vs); RET v >>
```

As we need the return value, Do actually needs to apply a. So we need a vs to apply it to. But we don't have one. We have not been keeping one.

This is flaky if you have a lot of actions that return values. Every time the client does a Do, you have to laboriously reconstruct enough of the volatile state, by applying the current volatile log, to determine the return value. That is not very attractive.

So, in most cases, we add another component, the vs, to the state variables in the implementation. With it we explicitly track the changes in the vs as opposed to just representing them implicitly, in the vl, as a sequence of updates. So our implementation state variables are:

```
VAR ss
    vl
    sl
    vs                    (added)
```

For a concrete example, think of a file system. The reads have to be affected by the writes which are done. If I do a write 6 to byte 1000 and then read byte 1000, I have to get 6 back, not the previous value. So if all I have done is remembered in the log that I have written 6 to byte 1000, then when I do the read I have to search the log to see whether there are any writes in the log which should affect the answer. An that, in general, is going to be somewhat painful and expensive.

So usually you don't do that. And we will see, in about 3 or 4 weeks, a considerably more complicated but also much more practical strategy for maintaining the volatile state efficiently. The difficulty with maintaining the volatile state is that it might get very big - you might run out of memory to store it. That gives rise to a number of complications that we will be discussing in a few weeks, but for now, we are not going to go into that.

6.5 Redo Log Implementation - Redo

There is another difficulty. Each Commit adds to the stable log, which keeps getting longer and longer. Further, nothing in our implementation every changes the stable state.

So, not only do we have this problem that we are not actually representing the volatile state, we are not really representing the stable state either. There is a a stable state variable, but having never changed it, we might as well not have written it. As more and more work gets done, there gets to be more and more log to search through in order to find out what is going on.

Of course, the only reason to find out what is going on is if you have to return an answer, but the system is not very interesting if it is write only. There are going to be reads occasionally. In fact, usually there are more reads than writes.

So clearly we have to do something about the stable state. But what are we going to do?

We add a new operation Redo. Redo essentially takes things, one at a time, out of the stable log, and applies them to the stable state.

Illustration: Given a stable log with several updates, sl = $\{$a, b, c$\}$, Redo does:

```
ss := a(ss)
ss := b(ss)
ss := c(ss)
sl := {}
```

Redo applies the updates in the stable log to the stable state. Of course, it is possible to fail and crash in between these. So in order to guarantee that Redo gets to run to completion after a Commit, Crash, as well as Commit, runs Redo. So either way, we are going to keep going through this whole sequence of applications until we succeed. Log idempotence permits us not to care if this happens repeatedly.

```
Commit = << sl := sl + vl >> ; Redo()

Crash  = << vl := {} >> ; Redo()
```

When we are all finished, we can set the sl to $\{\}$, because we don't need it once it has been applied.

The whole picture: To implement arbitrary updates, and make them crash atomic, you have this abstraction with Do and Commit as the client actions, along with an uncontrollable action, Crash. You keep a log in which you record the actions. It is volatile as long as you just keep doing Do's. When you do a Commit, the log is made stable atomically. Then you do this redo thing which goes through and actually executes these actions on the stable state. If you succeed in getting all the way to the end of that, then you reset the log. Then you are ready for another sequence of this whole process.

It should be pretty obvious that you don't have to arrange Redo in the a rigid way described here and in the handout. We do an exhaustive Redo as part of Commit. But we could do any prefix of the log, then delete that prefix, and then keep going. We actually have a lot of flexibility about the order in which we do the stable state update.

There are lots of practical reasons why you might want to take advantage of that flexibility, as the problem set suggests. You may wish to batch updates, as they require access to the disk, which is going to be slow. Further, as it is written, Commit requires the client to wait around while the relatively expensive stable update (Redo) operation takes place. There is no need for this. All that is required is to wait until the stable log has been changed. Having changed the stable log, the abstract stable state has been changed. Which is all you need to do.

Notice that this Redo operation has the property that it doesn't change the abstract state at all. All it is doing is applying the implementation's stable log to the implementation's stable state, but we already have it applied in the abstraction. The abstract stable state is defined as the concrete stable state with the concrete stable log applied ($ss_{abs} := ss_{impl} + sl_{impl}$). But that is all Redo does. And we have already "done" it in the abstraction function.

So as we execute redo, the concrete stable state is going to change, but because of the log idempotence property, redoing the log on the changed stable state is not going to have any abstract effect.

To make this more concrete: the simplest actions to understand which have the idempotence property are straightforward writes to fixed addresses. You can repeat these as many times as you want, and you will still get the same answer as long as you finish up by writing them in the right order.

This set of ideas is used repeatedly in the course and in the construction of real systems which have fault tolerance. It is important to understand them.

6.6 Idempotence and arbitrary updates

Here is a general recipe for taking a arbitrary update and making it have the log idempotence property. It involves attaching uids to the updates.

The whole idea of the redo is to break the actions into atomic stable state updates and do them one at a time. As we have been doing previously, we will consider disk page writes to be the primitive atomic operation. However this need not be the case, we might be using some higher level abstraction than the disk, but it is convenient and very concrete.

The idea of this general technique for turning an arbitrary disk page update into one that has log idempotence, is to put a version stamp (or time stamp, or unique id), on the disk page, and put the same uid on the update. That way you can tell if the update has been applied.

Let's look at a concrete example.

Suppose that the disk page contains just one major variable `x`. Suppose the update is add `3` to `x`.

Of course real disk pages don't contain just one anything, they contain all kinds of stuff. The variable `x` just stands for whatever arbitrary pile of garbage is on the disk block.

This update obviously does not have the log idempotence property—if we do it twice we don't get the same answer we would get if we only do it once. In fact, most interesting updates do not have the log idempotence property. That's why this is an interesting construction.

The notion is to replace the original update with one which is conditional on the version of the disk page. We go to the page and find out its current version number, and then create an update which only acts if the version number is the same as it was when the update was created. When it acts, it does the original update, with an incremented version number.

```
                  Original                 Idempotent

     The disk:       x: Int                   x: Int
                                              vn: Int

     The update:     x := x+1                 vn = 10 => x  := x+1;
                                                         vn := 11
                                              [*] SKIP
```

So now, if the update has already been applied, it won't be done again because the version number won't be 10. It will be at least 11. And if subsequent updates come along, they will always increment the version number.

On the other hand, if it hasn't been applied yet, then the version number will be 10, and we will apply it.

Why will the version number be 10? Why won't it be 9, 8, or 7? Because the only way it could have gotten it up to being 10 at the time we generated this update, is that previously, we had updates which incremented it up through 7, 8 and 9. Those updates preceded this one in the log, because we put updates in the log in the same order that we generate them. Therefore, we definitely must be up to 10, if we replay the log up to this point.

This is a completely general method, using version numbers, or time stamps or unique ids, or whatever you want to call them, for taking an arbitrary update, and making it have the log idempotence property.

Question: Does it have to be one disk page at a time?

Yes. In this example, yes. The reason it has to be one disk page at a time, is because redo relies on having each individual update be atomic, and if we are building right on top of the raw disk, the only atomic operation we have is to write one disk page. Now, of course, we could build on top of some other abstraction, one that gives you bigger atomic operations. But what is written in the handout is extremely generic. All it says is that you have some way of representing a stable state. It doesn't say anything about disk pages. I have been using the disk pages in the filesystem as a concrete example but the idea is extremely general.

For example, it is very common to use these methods on top of a standard Unix filesystem. If you want to build a database on top of a filesystem that can tolerate crashes, this is the standard way to do it. You represent the data in a file, you represent the updates in a log, which is another file, that you just append to. You then apply the atomic update operation, or whatever your filesystem gives you as atomic operations, depending what sort of filesystem you have. Though you have to read the manual pretty carefully in order to get even a clue as to what that might be half of the time. But it could be much more than single disk pages.

Question: The handout uses sets of uids rather than version numbers.

Our approach here is a bit cheaper to implement, as you don't need to maintain sets of uids. Another way to look at it is that this is an implementation of the slightly more abstract version with the sets of uids. This one is perhaps a little bit too detailed. If you look at real implementations, they don't necessarily increment the version number by 1, for example. What is very common to do in database systems is to actually use the position of the update in the log as the version number.

I gave this example to be very, very concrete, but this is the standard technique to make database systems work, and there are a number of different strategies for choosing the version numbers. I don't think anyone uses exactly this strategy. Maybe some do. Certainly most systems don't use exactly this strategy.

Question: I don't see how this gives you fault tolerance.

What this does is provide the property that, if there is a crash, then anything that got done since the last commit is going to get lost. This is because what happens when there is a crash in the abstraction is that the volatile state gets reset to the stable state. The stable state only gets changed when there is a commit.

Question (cont): So there is still data lost...

There is data lost, but it is not arbitrary what gets lost. The client knows that after a failure, the thing is always in the state it was in after the last commit. Which is much much better than having done a whole bunch of updates and having no idea which ones got applied. Thats pretty hopeless. This is called crash atomicity, because it allows you to take a bunch of updates, group them together and then guarantee that either all of the updates in the group get done, or none of them get done.

Question: If you committed your updates to the stable log, each time you changed the stable log, then you would have fault tolerance.

In general, you don't want to do that. Remember the example of maintaining the free list for the filesystem. We had two updates there, we had to add a block to the indexing structure at the bottom, and to turn off the bit in the bit table representing the free space. We didn't want to do only one of those, we wanted to guarantee that we got either both of them or neither. It would be a big loss if you said you _had_ to make each update stable.

It is a very important property of this that you get to choose when the thing becomes stable, because if you are in the middle of doing the updates, then the fact that you are still computing is encoding the fact that there is still more work to be done. But after a failure that information is all lost. So you have to have some other way to capture what constitutes a good state, as opposed to some intermediate state which still needs work. That is the function of commit. When I commit, that means now it is a good state. I am willing to live with this state if there is a failure.

In the example of filesystem, I would not be willing to live with a state in which I had allocated a page, but had not yet put it in the indexing structure, so it would be lost.

Question: What if we fail in the middle of a Redo operation?

Redo will get restarted because Crash will do Redo again. Where will it redo from? It will redo from whatever is left in the stable log. In this version of Redo which erases the stable log each time it is called, it would be from the most recent commit. But as I pointed out, it doesn't have to be like that. You could have a lot more flexibility about exactly when to take stuff out of the stable log.