Transactional Block Store

1 Introduction

The goal of this project is to construct a transactional block store The block store must be transactional. You must use the write-anywhere file layout (WAFL) scheme to implement such a block store. All operations including updates to data and meta-data must must be atomic. If the program dies at any time, it must always be possible to *quickly* recover a consistent state of the store. You can assume that there will be no concurrent operations on the same blockstore

For the purposes of this project, the backing store will be an ordinary unix file. At the time of blockstore creation, create a new file, fill it with zeros up to the required size, and think of this as the disk drive. All read/write operations that you perform on this "drive" should be in units of BLK_SZ (the block size). The blocks on this drive will provide the necessary storage both payload data and meta-data.

You may not use or make references to other transactional stores like Aries, Berkley DB, etc. This project will be due as posted on the course website. The interface file `blockstore.h` and a test file `bl-test.c` can be found at `/home/blstore` on `cs418.cs.jhu.edu`. The test program can be invoked from the command line as:

```
bl-store -s <size> <blockstore-name>
```

Much of the text in the rest of this document is borrowed from Prof. Shapiro's original description of the project.

2 Virtualization of Disk Blocks

The block layer has two name spaces: the DBA of a block is its ``disk block address." This describes the actual location of the block relative to the start of the volume. The LBA is the ``logical block address" (Note that this is *not* the same as the logical block address that we discussed for disk drives). All references made from the object layer into the block layer should be made in terms of logical block addresses. The logical block address of a block does not change during the lifetime of a block. The disk block address may (probably will) change as the block is moved or updated. The idea here is that the LBA gives an externally referenceable *name* to each block, while the DBA described a *location* for the block. The mapping from name to location must be maintained by the block store.

Note that the LBA space is deliberately bigger than the DBA space. It is possible that you will attempt to write a new block and learn that there is no space to hold it. Logical block addresses (LBAs) are not reused – they start at zero and go upward. Disk block addresses *are* reused – the block layer must maintain a free list for them. We recommend an allocation bitmap. The part of this that is tricky is that *no disk block that is allocated at the start of a transaction can be overwritten during that transaction*. The block layer must always allocate new (free) blocks the first time you tell it to write a given LBA in a given transaction. It must also arrange that the lba->dba directory is updated at the end of the transaction to point to the new location of that LBA. Finally, it must ensure that after the transaction is completed the *old* DBAs (the ones that have now been replaced by new data at new locations) are free.

Implementation Note: Note that you should *never* reference the `value` field of a DBA or an LBA directly in your code (see blockstore.h). Always use the DBA/LBA macros, as in:

```
DBA(someDBA)=newValue;
read(fd,DBA(someDBA)*blockSize,blockSize);
```

This will ensure correct functionality when we interpret LBAs ans DBAs as integers (by enabling the `PRODUCTION` flag). Your code must work correctly with this switch enabled or disabled.

3 The Block Store

There are several parts to the implementation of the block store:

1. The handling of transactions.
2. Keeping track of which LBAs have been assigned to new locations during the current transaction in progress. This is an in-memory data structure.
3. Allocating new blocks and arranging (when appropriate) for old blocks to be freed at the end of the current transaction. This implies managing the allocation bitmap inside the store and also in memory.
4. Keeping the lba->dba directory updated.

We have discussed the header management part of the transaction handling in class: you need two header blocks and you need an ever increasing transaction number so you always know which one is current.

3.1 Managing the Allocation Bitmap Updates

It is tempting to imagine that we should just use LBAs for the allocation bitmap and the lba->dba directory. Actually, we *could* do this for the allocation bitmap, but it's dicey for the lba->dba directory. If traversing the lba->dba directory requires lba lookups, then the lba->dba directory becomes a *very* delicate data structure.

Still, it would be nice to have a renaming layer here so that we can rewrite the allocation bitmap and the lba->dba directory conveniently. We want something *like* the lba->dba directory, but we don't want exactly that solution. Here is one way to handle this.

First, notice that the allocation bitmap is a fixed size data structure (except when the block store is resized, but this can be handled as a special case). We only have so many blocks, so there are only so many possible bits in the allocation bitmap. Second, notice that in contrast to the situation in the lba->dba directory, our numbering scheme for these blocks is dense. The first block of the allocation bitmap is block zero, the second block is block one, and so forth.

This suggests that we could create a separate mapping scheme for the allocation bitmap blocks by simply creating a hierarchical arrangement of indirection blocks. This data structure works exactly like the indirection blocks in a conventional file system, except that we are only going to have one block pointer at the top of the tree. Let us call this a “hierarchical block map” and refer to addresses in this map as HMAs. A hierarchical block map translates from HMAs to DBAs. What we are trying to do here is set things up so that the first block of the allocation bitmap can live at any DBA but always lives at HMA=0.

Constructing the initial hierarchical block map is not hard – you are simply building a tree. Each entry in the indirection blocks is a dba_t for the next block down. The entries in the bottom level of the hierarchical block map are the DBAs of the actual bitmap blocks.

The tricky part about managing the hierarchical block map is what to do when you update it – that is, when a block in the allocation bitmap moves. Obviously we are going to rewrite the block of the allocation bitmap, but this means that we need to rewrite the hierarchical block map table entry that points to it, which means that we need to rewrite (and relocate) that block – and so forth. It turns out that there is a simple strategy for dealing with this.

When you are getting ready to relocate an allocation bitmap block, you are going to make some sort of procedure call that updates the location of the allocation block within the hierarchical block map. As you walk your way down the map, check each block that you traverse to see if that block has been copied yet. If not, allocate a new block, copy it, and traverse the copy instead. You can use the allocation bitmap to determine if the block you are looking at has been copied – if it has been copied, it will show up as “not previously allocated” and also “allocated at the end of this transaction.”

All of this will involve recursive calls into the block allocation system, which will recursively call the allocation bitmap code, which will recursively call the hierarchical block management code. As long as you get the order of allocation and replacement correct, this will all work out fine. Just be sure to replace the blocks in the hierarchical map from the top down, and stick the new block in place before you try to deal with the next layer.

Of course, you do not rewrite the hierarchical block map when you are doing a lookup – that is a read only operation, and no updates are required anywhere. The current top block of this hierarchical block map is named by a field in the header.

3.2 Managing LBA->DBA Directory Updates

If you wrote your hierarchical block map in reusable form, you can use a second hierarchical block map to identify the blocks in the lba->dba directory. Let us call the namespace that linearly identifies the blocks of the lba->dba directory as GMA (similar to HMAs). So, this second hierarchical map translates GMAs to DBAs. In practice, this is probably the easiest way to deal with the locations of the allocation bitmap and the lba->dba directory. Since we are now using a separate name space for these overhead structures, they should not appear in the LBA space at all.

3.3 Initializing the Volume

When you initialize the block layer, you need to create the headers, the initial lba->dba directory, the initial allocation bitmap, and the initial indirect block hierarchy for the overhead structures.

3.4 Writing a Newly Allocated Block

Suppose we do a transaction that writes exactly one new block. This block is written with *bl_write*, so we need to allocate a new DBA for the new content. Here is how matters proceed:

The way to think about this is that before you can write a block you have to allocate it from the allocation bitmap. So the first thing you need to do is to allocate a new DBA for this new content. If you can't find a free block, return a “no space” error. From this point on, let us ignore the “no space” case.

Problem 1: So now we have allocated a block, but we have a problem – we need to mark this allocation in the allocation bitmap. In order to do that, we need to rewrite the block in the allocation bitmap that contains this bit. We cannot do that without allocating a new DBA *for the bitmap block*, so we have to find a free block again, and *that* free block needs to come out of the allocation bitmap.

We might get lucky, and the second block allocation will end up in the same allocation bitmap block as the first, but we might not. In the limit, we might end up (recursively) needing to reallocate *every block in the bitmap*. The rules are:

1. You can't change a block in the bitmap without allocating a new location for that bitmap block.
2. Each bitmap block only needs to get reallocated once – once you have allocated new storage for a given block, you don't need to allocate storage for it again during the same transaction. You can know that a block has already been reallocated by seeing if it is newly allocated in the allocation bitmap.

Problem 2: But what about the blocks that you are *releasing*? Once the transaction is completed, the *old* locations of the allocation bitmap – the ones that you are in the process of rewriting – are going to become free. As they become free, *their* bitmaps in the allocation bitmap need to change. In order for those entries to change, the containing blocks of the bitmap need to be rewritten, and so forth.

Problem 3: This creates a third problem: *you must not mark the deallocated blocks free too eagerly*. If you do, there is a possibility that you could reuse a block too soon (i.e. before the transaction commits). The easiest way to avoid this is to build an in-memory allocation bitmap. This is initialized at startup from the on-disk bitmap, and it is updated at the end of each transaction. In this bitmap there are *two* bits for every block. The first describes the 1=allocated/0=free state at the *beginning* of the transaction, and the second describes the 1=allocated/0=free state at

the *end* of the transaction. A block is a candidate for allocation if (1) it was free at the beginning of the transaction and (2) it will not (yet) be allocated at the end of the current transaction. That is, if the bitmap state is "00". At the end of the transaction, after all blocks have been written, the "new state" bits get copied to the "old state" bits. Conceptually two bits is the right idea, but it is the wrong implementation. The implementation will be much easier if you simply maintain *two* in-memory allocation bitmaps: the before map and the after map for the current transaction.

3.5 Updating the Directory

So now we have written the block and we have dealt with the allocation bitmap, but we need to solve a new problem: we need to assign an lba to this block and introduce an lba->dba mapping into the block layer lba->dba directory. Assigning an LBA is easy. Take the last allocated LBA value and add one. We already know the DBA because we know what block we allocated at the disk level.

Updating the lba->dba directory is conceptually just like updating the allocation bitmap: we are probably going to be revising an existing block in the directory, and we need to allocate a new DBA for the relevant block and deallocate the old one. You must use an extensible data structure like a balanced tree or an extensible hash table to manage the LBA->DBA map.

3.6 Rewriting an Already Allocated Block

When you *rewrite* a block, it is very much like writing a newly allocated block. You still need to allocate a new DBA. The difference is that you already know the LBA and you need to arrange for the *old* DBA to be freed at the end of the transaction.

3.7 Dropping a Block

This is the inverse of inserting a new block – you are freeing a DBA at the end of the transaction and *deleting* a directory entry. Note that since the drop must be done in a transactional manner (with copy on write behavior), dropping a block might fail with a "no space" error!

3.8 Resizing the blockstore

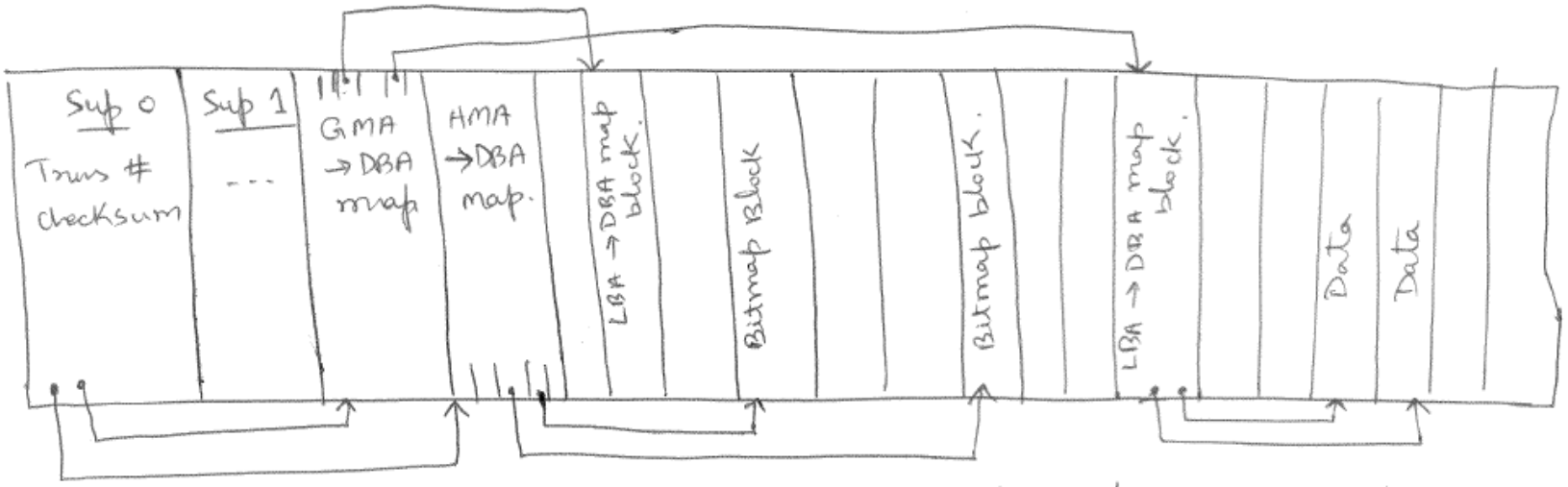
A blockstore resize (growing / shrinking) will result in a change in the size of bitmap blocks as well as its indirection structure – the hierarchical block map. It might also result in changes to the LBA->DBA map (and its indirection structure) . These operations must be done in a way that does not compromise the integrity of blockstore , and only if there are enough blocks to perform meta-data updates. That is, resizing a blockstore is best thought of as an independent transaction in itself, and must never perform in-place updates.

3.9 The SUPER block(s)

As discussed in class, you must maintain *two* super-blocks. The super-blocks maintain the latest transaction number along with a simple checksum of the *entire* contents of the super-block. During startup/recovery, you should use the super-block which completed the latest transaction with a valid checksum. At a commit stage, you should ensure that all other blocks are written to disk, and write a super-block that was not a part of the previous commit.

3.10 A full Picture of the block store

A full picture of the blockstore depicting all the data-structures explained above is shown in the figure below:



only one Layer of Indirection is shown for HMA → DBA and GMA → DBA hierarchical maps. In practice, they will have multiple levels, depending on the size of the Physical volume (# DBAs). The two Super blocks are the ONLY blocks that live at fixed Locations