FCFS: On-Disk Design

Revision: 1.8

Stewart Smith
sesmith@csse.monash.edu.au

Date: 2003/07/06 12:26:43

1 Introduction

This document describes the on disk format of the FCFSobject store.

2 Design Constraints

2.1 Constraints from attributes of physical disks

The way disks in the real world work place a number of design constraints on us. These aren't so much strict constraints as constraints we need to follow in order to acheive good performance.

- Disk seeking is expensive and should be avoided
- Blocks with similar numbers are less expensive to seek to than blocks with very differnt numbers.
- Accessing N blocks at once is quicker than accessing N blocks one at a time.
- Blocks go bad after time, that is they can no longer be reliably used to store data.

Even in upcoming and emerging storage technologies (e.g. holographic storage) these constraints are still relevant.

2.2 Walnut Constraints

The system the object store is primarily designed for is the Walnut Kernel[?]. The design of Walnut places several design constraints on the system.

- Objects must be easily memory mapped
- Object lookup must be fast
- The storage system must be robust
- The storage system must ensure both data and metadata consistency.
- The storage system must support a persistent system
- The storage system must not leak information (one of Walnut's goals is security)
- The storage system must be suitable to implement in a microkernel environment.
- The storage system must support caching of objects from other devices (Walnut is meant to be distributed)
- The storage system must be able to support Walnut's password-capability.

```
struct a_data_structure {
       u64 MAGIC1;
       u32 value;
       u32 value2;
       u32 value3;
       u32 value4[10];
       u64 MAGIC2;
       };
```

Figure 1: Seperated Magic Numbers

3 Magic Numbers

Magic numbers are often used in data structures to aid in debugging. When dealing with on-disk structures, they can also greatly help in disaster recovery. By searching a disk for blocks containing magic numbers in specific locations it can aid us in reconstructing a partially destroyed volume.

By seperating magic numbers withing data structures (e.g. Figure 1) it allows better detection of corruption within a specific area of a data structure. For example (from Figure 1), if a buffer overflow was to write 11 words into value4[], hence overwriting MAGIC2, we'd still be able to see the MAGIC1 number and examine the possibility of this being a `data_structure`.

4 Addressing Blocks

There are a number of ways of addressing blocks in use today. The way we (usually) have to reference them to the block device layer of the Operating System is by a logical block number from the start of the block device. Most systems also have the ability to specify an amount of read-ahead that will proceed asyncronously after reading the block we requested.

Traditional UNIX filesystems track what parts of the disk are being used by a file in its inode structure. There is an attempt to optimize for the (traditionally) prevalent smal file on UNIX systems. Thus, on very large files it is quicker to access the first few blocks of a file than it is to access the last one. Traditional UNIX systems (e.g. FFS, ext2, ext3) the inode has space for several block numbers (direct blocks), several indirect blocks (that is, a pointer to a structure that lists block numbers), less doubly indirect blocks (a pointer to a structure of pointers to structures with block numbers) and sometimes even an entry for triply indirect blocks (see Figure 2).

This system has several disadvantages when dealing with larger files. Although a block at the start of a file will not require a seek to find, a block at the end of a file may require up to four extra seeks to find. Also, this method provides little advantage in the optimal case of all blocks in a file being sequential on disk.

4.1 Block Runs

BeFS[?] uses a structure called a Block Run (Figure 3) instead of block numbers to reference locations on disk. This has the distict advantage of optimizing for the best-case scenario where each file is a contiguous run of blocks on disk (a single structure can address up to 64MB with a 1kb block size). The `allocation_group` member refers to the number of the BeFS allocation group.

We use a variation of the BeFS `block_run` structure, removing several of its limitations. The original structure (Figure 3) was limited to 2^{48} blocks on a volume (with 1K block size, a maximum volume size of 2^{58} bytes). The FCFSblock run (Figure 4) has an addressing limit of 1.84×10^{19} blocks. Considering that even with the (relatively small) block size of 1K, this would address over 16.7 million terabytes of data, I do not perceive this will be a problem in the near future.

4.2 Allocation Groups

In BeFS[?] an allocation group is not a structure in itself. It is a collection of parts of several data structures within its filesystem. Logically,

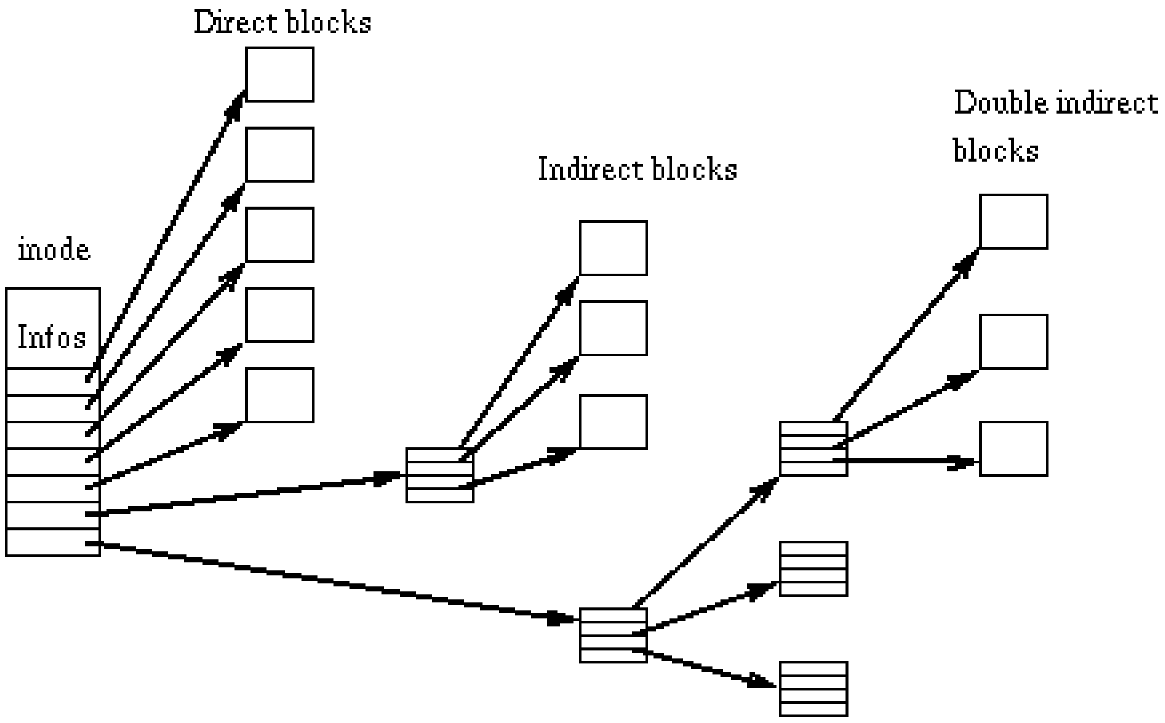

Figure 2: Direct, Indirect, Doubly Indirect and Triply indirect blocks (ext2 inode from [?])

```
typedef struct block_run
{
        int32 allocation_group;
        uint16 start;
        uint16 len;
} block_run;
```

Figure 3: BeFS Block Run ([?] P47-48)

```
/* fcfs_block_run
   --------------
   We have a max of ~4billion alloction groups,
   within these, we can have up to 4billion blocks.
   and we can address these with *one* block_run structure.
   This means we don't have a lot of block address lookup on large files
   or contiguous files. Should lead to fast IO...
 */
struct fcfs_block_run {
  u32 allocation_group; /* Allocation group */
  u32 start; /* Start Block */
  u32 len; /* Length (blocks) */
};
```

Figure 4: FCFSBlock Run (BLAH.h)

4.3 Sparse Objects

Many file systems have found it advantageous to support "sparse files". A sparse file is one where as-yet unwritten areas of the file are not written to disk. A good example is a sparsly populated array on disk where large parts of the file remain unoccupied for large periods of time (or even forever).

A sparse object is indicated by the `FCFS_OFLAG_IsSparse` flag being set in the onode. A `block_run` for a sparse area of a file will simply start at offset 0 of allocation group 0. This location could never **really** be allocated to an object as this is where the super block5 is located.

As a result of this, sparse areas in an object are at a block granularity. That is, only blocks left blank won't be written to disk - a block that only has one byte written will be written in full, and occupy a full block. In my opinion, the added complexity and implementation time for finer granularity is unwarranted at this level - having a higher level filter or compression mechanism will be supported and work if any user requires such an arrangement.

5 Superblock

A super block is used both to easily identify the start of a valid volume and hold some critical file system information such as the location of certain data structures.

5.1 Constraints

The design of the super block must follow a few extra constraints:

- Must be easily identifiable from other system's superblocks
- Must be reconstructable (in case of a true disaster)
- Must be duplicatable (multiple copies = easier recovery when one is lost)
- Must be rarely needed to be modified as this could be a bottleneck when multiple processes are updating the Object Store.

5.2 Required Data

The Super Block will need to

5.3 Placement on Disk

The superblock will be the first block on the volume. In the case of existing in an environment where the first block on the disk has special meaning to the bootloader (e.g. the PC's boot block), then the FCFSsuper block can be the second block on the disk. Backup superblocks (for redundancy) will be kept at the start of each allocation group and in the final block of the filesystem.

To differentiate between the locations of the backup superblocks (to aid in volume recovery)

6 Object Node (o-node)

Most conventional UNIX file systems have the concept of an inode. That is, a data structure (stored in a block) that represents a file. Directories are actually files containing filenames and inode numbers. This implemenation is hidden from the user but internally, the filesystem takes advantage of this to aid in block allocation.

Keeping the indexing structure seperate from the data allows for great flexibility within the object store. The main advantage is that it becomes possible for the indexing structures to **be** an object on disk. This would allow indexes to take advantage of features like sparse objects4.3 and versioning.

6.1 onode fields

6.1.1 FCFS_ONODE_MAGIC1

A magic number that allows us to identify that this block is an onode. We're also human readable string to aid in debugging and human based recovery. We use the string "ONoDeV1" followed by a revision (unsigned char), This is currently `0x01`.

```
#define FCFS_ONODE_MAGIC1  0x4f4e6f4465563101ULL /* ONoDeV1 0x01 */

enum fcfs_onode_flags {
  FCFS_OFLAG_NoVersion,/* Don't version track this onode */
  FCFS_OFLAG_IsSparse /* Contains unwritten areas */
};

struct fcfs_onode1 {
  u64 magic1; /* Identify as O-Node */
  u64 onode_num; /* FS Specific Unique ID */
  u64 onode_revision; /* Revision of onode */
  u64 flags; /* fcfs_onode_flags */
  u64 use_count; /* Reference Counter (for indicies) */
  u32 onode_size; /* Length of o-node structure. */
  u32 forknr; /* Number of Forks */
};
```

Figure 5: Base O-Node Structure

6.1.2 onode_num

Each onode gets an identification number which is unique for **this volume only**. In Walnut, an object ID should be globally unique, we do not natively provide this and it is expected to be provided by the Walnut index to the object store.

The decision to not use globally unique identifiers was a consious one. Ensuring global uniqueness at this level seemed excessive (especially for systems which don't require such things). There is also the advantage of being able to allocate onode numbers in a way that helps optimize the onode index tree and keep it from getting too unbalanced.

6.1.3 onode_revision

Every time the onode is changed, the onode_revision attribute should be incremented by one. This number could be used in a distributed or networked environment to ensure that client nodes are using up to date copies of the onode structure.

7 Block Allocation

We're currently using the cop-out approach of a block bitmap. Starting from block 1 of an allocation group (the block after the backup super block), one bit represents a disk block in the allocation group. This starts from block zero (i.e. the backup superblock).

References

[1] Maurice David Castro. *The Walnut Kernel: A Password-Capability Based Operating System.* PhD thesis, Monash University, 1996.

[2] Dominic Giampaolo. *Practical FileSystem Design with the Be File System*, chapter 1. Morgan Kaufmann Publishers, Inc., 1999.