ETFS Design and Implementation Notes#

One of the first questions that comes to mind when learning a new file system, is "What is the on disk file system structure?" ETFS does not have one. Well, not a traditional one, anyways. It does not have a freespace bitmap, free inode bitmap, FAT table, directory structures, or any of that jazz. Then how can it work? It has an on disk structure, but no on disk file system structure. As I wrote earlier, it does not have what one would consider a traditional on disk file system structure.

At its core, ETFS (which stands for the Embedded Transactional File System) is composed of transactions. The transactions are written to the media, but the file system structure is entirely in the host's memory. When the file system starts up, all the transactions are read from the media, and then file system structure is reconstructed from the transactions.

ETFS can work with flash (NAND and NOR), RAM and Disk media. These notes will focus on ETFS as it relates to NAND flash media. When it comes to NAND flash, ETFS has drivers (available from BSPs and Drivers project) for NAND512 and NAND2048. Although flash is accessed a page at a time (except for erasing, at which point it is a block at a time), the ETFS library accesses flash a cluster at a time. The ETFS cluster is not the same as a cluster used for file systems such as fs-dos; it is different. With NAND512, an ETFS cluster is 2 pages in size (1 kB + 32 spare bytes); with NAND2048, an ETFS cluster is 1 page (2 kB + 64 spare bytes). NAND4096 (not yet supported) is likely to have 1 page per cluster and 128 spare bytes. You may be wondering why the NAND512 has 2 pages per cluster when all the others have 1. It's simple. The transaction information is stored in the flash's spare area. Unfortunately, there is not enough room in the spare area of 1 page to store both the transaction information and the ECC values.

Let's build on this. Although flash blocks are composed of pages, for our purposes, a flash block is composed of clusters. An ETFS transaction is also made of contiguous clusters, but it does not cross a block boundary. From this, one could say that a used block is composed of transactions. This file system is used in embedded systems where the typical number of files is expected to be less than 4096. However, it can not exceed 32K files. This page describes some aspects of ETFS and its transaction headers.

When the file system is loaded, all the transactions in the each block are read. As each transaction has a file ID, sequence number and associated extent (starting cluster + number of clusters), the file system structure can be reconstructed from this.

What follows is a brief summary of some of the ETFS source code modules and their routines.

slab.c#

This module manipulates slabs. What is a slab? A slab is simply a page's worth of slab entries that are linked together. You have a slab header that is always at the start of the page. It links to other slabs, and contains a link to a linked list of available entries in that slab. Each slab entry will have a pointer to either the next free entry in the slab, or a back pointer to its slab header. But this still doesn't really explain a slab. A slab is a convenient way to allocate memory for a bunch of objects. With one memory allocation, you can create space for manipulating a number of objects of the same type. If you need more or fewer objects, you don't have to go through malloc()/free() or mmap()/munmap() code every time. All you have to do is grab one from the slab's free list, or add it back to its free list. If the slab's free list is empty, allocate another slab.

Other modules can call the following slab routines ...

void slab_init(slab_ctrl *ctrl, size_t bytes)
This routine initializes a slab's control structure. The control structure resides in the device control structure and describes the slab and its properties.

void *slab_calloc(struct slab_ctrl *ctrl)

This routine allocates an unused slab entry and returns a pointer to its data section. If the slabs governed by <ctrl> did not have an unused entry, a new slab of entries is allocated and the unused entry is allocated from that slab.

slab_free(void *ptr)
This routine frees the slab entry associated with the supplied data pointer. If this was the only used entry in the slab, then the entire slab will be freed provided that there are at least a few other unused slab entries on other slabs.

reclaim.c

The reclaim module is responsible for reclaiming/reusing ETFS blocks. The reclaim structure resides in the device control structure. When you look at the source, you will see that it has two clean lists. One of them tracks the overworked clean blocks (clean2). An overworked block is simply one that has been used significantly more than average. A clean block is one that has no active clusters AND has been erased. The other clean list tracks the lazy, or under-used clean blocks. It also has a filthy list for tracking unused blocks that have not yet been erased. It also has a spare list that is used to track clean blocks for emergency use. Finally, it also has a the *blks* list for tracking all the flash blocks. Each block tracks the number of its inactive clusters. There are other fields to this structure, but these are the most important for understanding the reclamation sub-system.

int relcaim_init(struct devctl *dcp)
This routine initializes the reclaim sub-system. On the first time this routine is called, it allocates and initializes the necessary memory to track all the blocks belonging to the ETFS device. This routine may be called upon a restart condition after the ETFS driver received a STOP command. In that case, the memory has already been allocated, and it just needs to be re-initialized.

int reclaim_scavange(struct devctrl *dcp)
This routine finds the least active block that is in use. It then attempts to move its active clusters to another block. This would result in a filthy block that can be erased to become clean.

void reclaim_inactive(struct devctrl *dcp, unsigned devcluster, unsigned numclusters, int which)
This routine marks a group of clusters (within a block) as unused. This has the potential to free up an entire block.

unsigned reclaim_getblk(struct devctrl *dcp, int usespare)
This routine is responsible for getting a new block for writes. Before it tries to get a block, it will erase a filthy block if it exists and then add it to the appropriate clean (or spare) list. First it will try the average/underworked clean list. If no block was found there, it will try the overworked clean list. If that still didn't work it will try the spare list, only if <usespare> was non-zero. It it found a block, it will return the block number, or else it will return ~0 if it failed to do so.

void reclaim_eraseblk(struct devctrl *dcp, unsigned blk)
This routine, as its name suggests, erases a block. Depending both upon how heavily worked the block has been, and the number of spare blocks, it will either be added to the "spare" list, the overworked clean list, or the average/underworked clean list.

int reclaim_copyblk(struct devctrl *dcp, unsigned srcblk, unsigned dstblk) This routine copies the contents of one block to another. It is used to help balance the load between lazy and overworked blocks.

int reclaim_rewriteblk(struct devctrl *dcp, unsigned srcblk, unsigned *dstblk, int *offset)
This routine rewrites the current/active block. It is only called after a media error. It will try until it succeeds, or until it can not try another block. If it runs out of blocks, it will return ENOSPC.

int reclaim_mineblk(struct devctrl *dcp, unsigned blk)
This routine "mines" the specified block for the active clusters and transfers them to a new block. The block being "mined" will become filthy so that it can be reused.

void reclaim_write_badblks(struct devctrl *dcp)
This routine writes out the .badblks file. It is always exactly one (1) cluster in size. Unused block number entries are set to ~0. This routine does NOT tell the underlying driver that the blocks are bad, or that they should be marked as bad.

void reclaim_write_counts(struct devctrl *dcp)
This routine writes out the .counts file. It contains the erase and read counts for every block in the file system.

void reclaim_link(struct devctrl *dcp, struct linklist *list, unsigned blk)
This routine links an item (a block number) to the tail of the given list. The given list may be either the overworked clean list, lazy clean list, spare list, or filthy list.

unsigned reclaim_unlink1st(struct devctrl *dcp, struct linklist *list)
This routine unlinks the first item from the given list and returns its block number.

cache.c

This module is responsible for handling the cluster caching. The default number of of clusters it will cache is 64, although this can be overridden using the -c option (minimum=32). Why 64? There are conveniently 64 pages per block (when using 2k flash).

int cache_init(struct devctrl *dcp)
On the first time this routine is called, it allocates and initializes the memory to for the cluster caching. This routine may be called upon a restart condition after the ETFS driver received a STOP command. In that case, the memory has already been allocated, and it just needs to be re-initialized.

static struct cachenode *cache_get(struct devctrl *dcp, struct filenode *fnp, unsigned cluster, int flags)
This routine searches the cluster cache for a the desired cluster. If it does not find it, it attempts to return a pointer to an unused cache node. If it does not find an unused cache node, it performs a flush operation.

int cache_readv(struct devctrl *dcp, struct filenode *fnp, unsigned cluster, struct cachenode *vec, int nvec, int flags)
This routine performs a scatter/gather read operation.

int cache_writev(struct devctrl *dcp, struct filenode *fnp, unsigned cluster, struct cachenode *vec, int nvec, int flags)
This routine performs a scatter/gather write operation.

void cache_relv(struct devctrl *dcp, struct cachenode *vec, int nvec)
This routine releases a set of cache buffers. When the cache node's link field reaches zero (0), then it is deemed available.

struct cachenode *cache_read(struct devctrl *dcp, struct filenode *fnp, unsigned cluster, int flags)
This routine is a wrapper for a 1-part cache_readv().

struct cachenode *cache_write(struct devctrl *dcp, struct filenode *fnp, unsigned cluster, int flags)
This routine is a wrapper for a 1-part cache_writev().

void cache_rel(struct devctrl *dcp, struct cachenode *cap)
This routine is a wrapper for a 1-part cache_relv().

void cache_purge(struct devctrl *dcp, struct filenode *fnp, unsigned cluster)
This routine traverses the list of cache nodes to find stale entries. Once located, the stale entry is discarded; it does not get sent to disk. Stale entries can potentially arise after truncating (or deleting) a file. They are identified by having a cluster number greater than or equal to the starting cluster parameter.

node.c

This module implements the node library that is used to map a file ID to a node (directory node or file node). The file ID is used as an index into the fid2nodes field as well as a bit index into the bitmap field. This provides both a fast way to convert the file ID into either a file node or a directory node as well as a fast way to determine whether the file ID is in use. A 1-bit in the bitmap indicates that the corresponding file ID is available; similarly a 0-bit indicates that the corresponding file ID is in use.

To go backwards (from the file/directory node to the file ID) is also very easy. The file/directory nodes contain the file ID. File nodes and directory nodes are allocated using slabs. The fid2nodes field points to the slab entries for the appropriate file/directory nodes.

int node_init(struct devctrl *dcp)
This routine allocates any necessary memory for the (node) structure and initializes its fields. The memory is only allocated the first time this routine is called. It may be called again upon a restart condition, which will re-initialize the data.

void *node_alloc(struct devctrl *dcp, unsigned fid)
This routine allocates an in-memory directory node or file node. When FID_ALLOC is used as the file ID, the bitmap will be searched for the first available file ID. Once a file ID has been chosen (whether it was passed in or it was found), a new file node or a new directory node is allocated as appropriated.

void node_free(struct devctrl *dcp, void *ptr, int skipbitmap)
This routine frees an in-memory file/directory node. Typically when freed, its file ID is also marked as available. This however, can be overridden by setting the <skipbitmap> argument to a non-zero value. This particular behaviour is used with extended file IDs.

void node_reclaim(struct devctrl *dcp, unsigned efid)
This routine marks the bit associated with <efid> as free in the bitmap.

void *node_lookup(struct devctrl *dcp, unsigned fid)
This routine returns the pointer to the node associated with the file ID. This may be either a file node or a directory node.

background.c

This module is responsible for background processing such as saving wear information, erasing filthy blocks, block reclamation and defragmentation. As mentionned earlier, ETFS resource manager is mostly single threaded. That is, it only has one thread handling the ETFS server requests. To prevent the background thread and the main ETFS thread from clobbering the ETFS memory and disk data structures, a mutex is used.

It is undesirable to have the main ETFS thread wait for the background thread to finish all of its work before the main thread can can do its own. That would have too much of an impact on performance. To work-around this, a compromise has been struck--the background thread's code has been strategically peppered with code for cooperative thread preemption. When the main thread has work to do, it sets the flag dev.preempt in the device's control structure. The background thread will check this flag and unlock the mutex if set.

Let's take a closer look at each of the routines.

void *backthread(void *arg)
This routine is the entry point to the background thread.

void background(struct devctrl *dcp)
This routine is the main body of the background processing thread. It calls the routines for saving the wear information, erasing filthy blocks, reclaiming blocks and defragmenting the file system.

void defragment(struct devctrl *dcp, int duration) This routine defragments the files. Each operation has a sort of "cost". The "cost" of getting a valid file ID is 10. The "cost" of getting a valid file ID that needs to be defragmented is 110. The "cost" of getting an invalid file ID is 1. This limits how much time one spends in the defragmentation portion of the background thread. All the file IDs can not be processed in a single pass, even if the cost of each is 1. Therefore, the background thread keeps track of the last file ID it processed so that it can resume on the next pass.

void processblks('struct devctrl *dcp, int bg)
This routine is responsible for mining blocks for reclamation in the background. It will do this to blocks that have enough inactive clusters and for NAND flash blocks that may be getting weak (too many reads, or the block has been encountering ECC issues).

static void erasefilthy(struct devctrl *dcp)
This routine is responsible for erasing all the filthy blocks. Once erased, they are added to either the reclaim's lazy clean, over-worked-clean or spare lists.

static void savewearinfo(struct devctrl *dcp)
This routine saves the wear information. This includes flushing out the ".badblks" file, as well as the ".counts" file.

static void preempt(struct devctrl *dcp, int allow)
This routine performs the cooperative preemption with the main ETFS thread. If the dev.preempt field is set, it indicates that the main ETFS thread has work to do, and that the background thread should relinquish its lock on the mutex. It will regain the lock once the main ETFS thread is done with it.