JFFS3 plan extension

Ferenc Havasi, Zoltán Sógor, Mátyás Majzik

University of Szeged, Hungary

http://www.inf.u-szeged.hu/jffs2

28th September, 2006

Introduction

This document is an extension of the plan of JFFS3 that Artem B. Bityutskiy described at http://www.linux-mtd.infradead.org/doc/jffs3.html.

It includes a garbage collecting method which is relatively simple and easy to implement and can work without serious locking problems in the background, too.

Region

First we introduce the concept of *regions*. A region contains fixed number of erase blocks, similar to the old concept of virtual erase blocks in JFFS2.

In the following table you can see some examples:

Flash size	***Eraseblock size***	***Eraseblock Number***	***Eraseblocks in Region***	***Number of regions***
4GB	128KB	32 768	8	4096
4GB	128KB	32 768	16	2048
4GB	256KB	16 384	8	2048
4GB	256Kb	16 384	16	1024

There are three types of regular regions: *closed, unclosed* and *empty*. Closed regions are full of data, in unclosed regions there are some data and some free space too, empty ones are totally free.

There are five type of special regions: *erase, erasing, gc, sb, journal*. Erase regions contain no valid data and can be erased. Erasing regions are already under erasing. Gc region is under garbage collecting. Sb region stores the superblock (in a way described by Artem). Journal regions store the journaling information for the tree and the region-map.

To speed up accessing nodes we store local indexing information at the end of every closed region. It is similar to the *EBS* (erase block summary) introduced in JFFS2. This indexing information called *region summary* is an array and will be described later. The region summary of unclosed regions is collected in the RAM until it is closed. It is an endeavor to keep the number of unclosed region as small as possible.

Every region has a logical number which identifies it and there is also a place (physical number) where it is actually stored on the physical memory. The connection between these is stored in a map.

Region-map

This map is an array of records. The i^{th} record (raw) stores information about the region logical number i. It stores its physical number (where it is actually stored) and some other accounting information: erase counter, the size of the dirty space and status information.

The map is stored on the flash (position is stored in the super block) and read into the memory during the boot process. In most cases the map is changed only in the memory and its changes are stored in the journal. If the journal is committed, the new map will be written out, too.

Let us see a simple example:

Map in the flash:

Physical place	***Erase count***	***Dirty***	***State***
0	21	12342	closed
1	23	0	free
3	15	15404	unclosed
2	29	3433	closed

Map in the memory:

Physical place	***Erase count***	***Dirty***	***State***
0	21	12342	closed
1	23	0	unclosed
3	15	15544	closed
2	29	3433	closed

Corresponding journal entries:

1. [MAP_CHANGE, 2, MAP_CHANGE_DIRTY, 15456]
2. [MAP_CHANGE, 2, MAP_CHANGE_DIRTY, 15544]
3. [MAP_CHANGE, 2, MAP_CHANGE_STATE, MAP_STATE_CLOSED]
4. [MAP_CHANGE, 1, MAP_CHANGE_STATE, MAP_STATE_UNCLOSED]

Reading

As you can see, the plan of Artem JFFS3 will use B^+-tree to store global indexing information about nodes. In that plan a non-leave node stores concrete pointers to the nodes.

We introduce now to avoid storing direct pointers. Instead of a pointer, the address of the nodes will be divided into two parts:

- the logical region number where this node is stored [32 bit][1]
- the ordinal number (in the region) of this node [32 bit]

When we would like to read that node, first we have to identify its exact place in the region using the region summary. It is an additional indirection level, but it makes possible a simple garbage collection algorithm due to the map.

Let's see an example how logical address 0x00000001000000a3 is translated to physical:

The high 32 bit shows the logical region number of the node, in this case the value of it is 1. It can be translated to a physical region number using the first raw of the region-map. In the first field of this raw there are the physical region number of it. If its status is CLOSED there is region summary

1 The highest bit is always 0 on the flash. In the memory this bit is used by the tree to mark if the address is not calculated yet.

on the flash (may be cached in the memory, too), otherwise it is already in memory.

The second 32 bit shows the ordinal number of the node, in this case the 0xa3-th element of the region summary will show the offset of the not.

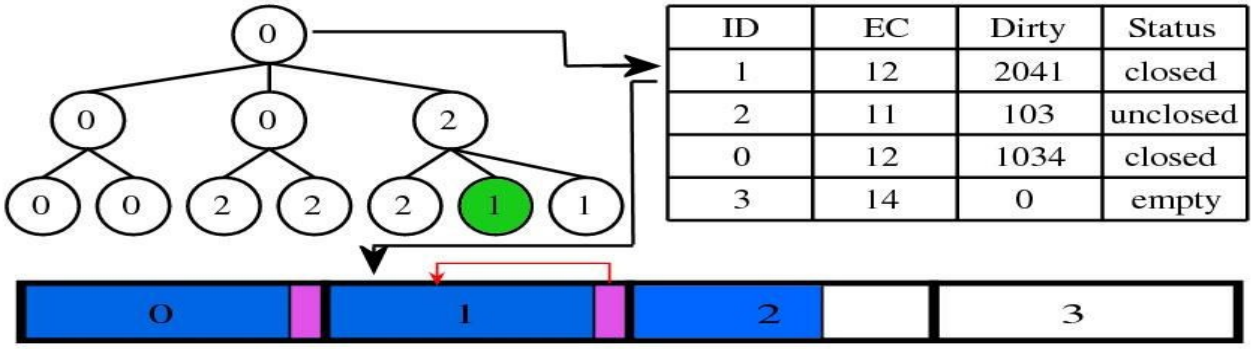


Figure 1.

Let's see how the reading happens when we would like to read the green node in Figure 1.

1. Super block has to be read to determinate the location of the root node. It is in region 0 in this case.
2. 0 is only the logical ID of the region in which the root node is. Using the region-map we can locate the physical location of this region, in this case it is the 1st physical region.
3. As this region is closed, the region summary tells the real position of the node in the region (it is shown by the red arrow).
4. The next node is reached by the information stored in this node. The read of the next node is similar to the process described above.

Region summary

The region summary is an array. The i^{th} element of this array is the offset of the node ordinal number *i*. There are a few special notes:

- If an ordinal number is not used (it is possible due to the MIRROR mode of garbage collecting), the offset of it is marked with 0xffffffff.
- When a region summary is loaded into the memory, not only the offset of the node is stored but the size of the node as well. The size of each node can be calculated except the last one. It is stored in the last element of the array on the flash.
- At the case of collecting region summary (at unclosed regions) *default_size*[2] of memory are reserved for it. It is expanded dynamically if necessary. Onto the flash it is written out only the used summary entries.

2 The default size of the region summary is: region size divided by the size of jffs3_raw_inode and jffs3_raw_dirent.

Journal

There are four kinds of journal entry:

- Journal management entries: journal start, stop entries and next region entry which resides at the end of a journal region and contains a direct link to the next journal region which is needed by the recovery process.
- B+ tree manipulation
- B+ tree commit
- Region-map manipulation
- Region-map commit

In most cases the manipulation of the tree (using wandering algorithm) and the change of the region-map is realized in the memory and every step is recorded in the journal. Periodically both of them are committed, and a commit entry is written out after the successful process. Typical reasons of commit can be:

- the changes use too much space in memory (in the case of tree)
- sync request
- journal cycle (described below)

During the file system mount process the system loads the last region-map then checks the journal if the filesystem is cleanly unmounted last time. If not, the recovery process triggers in and first it replays all region-map manipulation journal entries after the last region-map commit entry. Then scans all unclosed regions and when it is done the recovery process executes all tree operation in journal since the last tree commit entry.

The size of the journal will be 2+1 regions (can be extended easily because the implemented algorithm can handle more journal regions). 2+1 means that there can be two journal regions and when they are full then a new one can be reserved before the previous two dismissed. The logical number of the first journal region is stored in the super block and the regions are chained by the next region entry at the end of all journal regions. When the second region is also full, the following process will be applied:

- Reserve an empty region as new journal region
- Check if there are both B+ tree and region-map commit entries in the second active journal region.
    - If not force both the tree and the region map to commit and these commit processes use the new empty journal region to write their journal entries during commit process. After these operation completed the tree and region-map commit entries are written into the journal.
    - If yes then no commit action is needed.
- Now the previous two journal regions can be freed and now has one active journal region.
- Free the first active journal region. It is not needed because there are commit entries in the second active journal region. Now the second journal region will be the first one and the new region will be the second one.

W-buffers

JFFS2 uses only one write buffer. JFFS3 uses three write buffers:

- one for the normal data (data nodes and tree nodes),
- one for the journal,

- one for the mirror mode of the garbage collector.

Every write buffer allocates sector_size sized buffer.

Tree implementation

Tree-node on flash

A tree-node has two sections: tree-node header and tree-node data. The first two 16-bit wide field of the tree-node header is the magic and the node-type field. The value of the node-type field is JFFS3_NODETYPE_INDEX. The next two 32 bit wide field is the total length of this node and next the summary position[3]. In this implementation the size of the index nodes are always 4096. The last two 32-bit wide fields are the header_crc and the data_crc. Data_crc contains the CRC of node data.

```
struct jffs3_raw_index {
      uint16_t magic;     // JFFS3_MAGIC_BITMASK
      uint16_t nodetype;  // JFFS3_NODETYPE_INDEX
      uint32_t totlen;    // length of the node
      uint32_t sum_pos;   // position in region summary
      uint32_t hdr_crc;   // header crc checksum
      uint32_t data_crc;  // index node data crc
      uint8_t data[0];    // index data
}


struct node_data {
      uint16_t type;
      uint16_t keys;
      uint64_t data[0];
};
```

The node data section begins with a 16-bit wide type field identifying this node either is an indexing node or a leaf node. The next field contains the number of used keys in this node. After this there are as many 64-bit wide key fields as the previous number of keys field indicates then as many 64-bit wide link fields as the number of keys field + 1 because non-leaf nodes require one additional link in a tree-node.

Tree-node in memory

Tree-nodes in memory are similar to the data section of tree-node in flash but a tree-node has an additional 64 bit wide parent link information, which is needed by tree-node-cache wandering algorithms during flush and this information is not saved to flash.

3 It is used only by the recovery and the scan processes.

Writing and garbage collection

When writing new nodes we choose a region using the following algorithm:

1. If there are some unclosed regions: the region with the less empty space will be used for writing. It can reduce the number of unclosed regions.
2. If there are no unclosed regions, we choose the empty region with the least erase counts.

When a new node is written out, it gets a new ordinal. This ordinal will be the smallest number which is not yet used in that region (the first 0xffffffff entry in the region summary or if there is not any, the highest used ordinal plus 1).

Free space can be increased only by the Garbage Collector. Similar to JFFS2, the GC runs on a separated thread but its execution can also be forced.

The method of choosing a region for garbage collection is very similar to the algorithm used in JFFS2: if there is a big difference among erase counts (the difference between the minimum and the maximum value is greater than 1024), always the region which has the smallest erase count will be chosen, otherwise the one which has the highest dirty space will be chosen.

Regions protected against GC selection store the following elements:

- the super block
- the journal
- the region-map

Because all of them are moving periodically do not cause problem in wear leveling.

The garbage collector has two working modes: MIRROR mode and MOVE mode.

MOVE mode

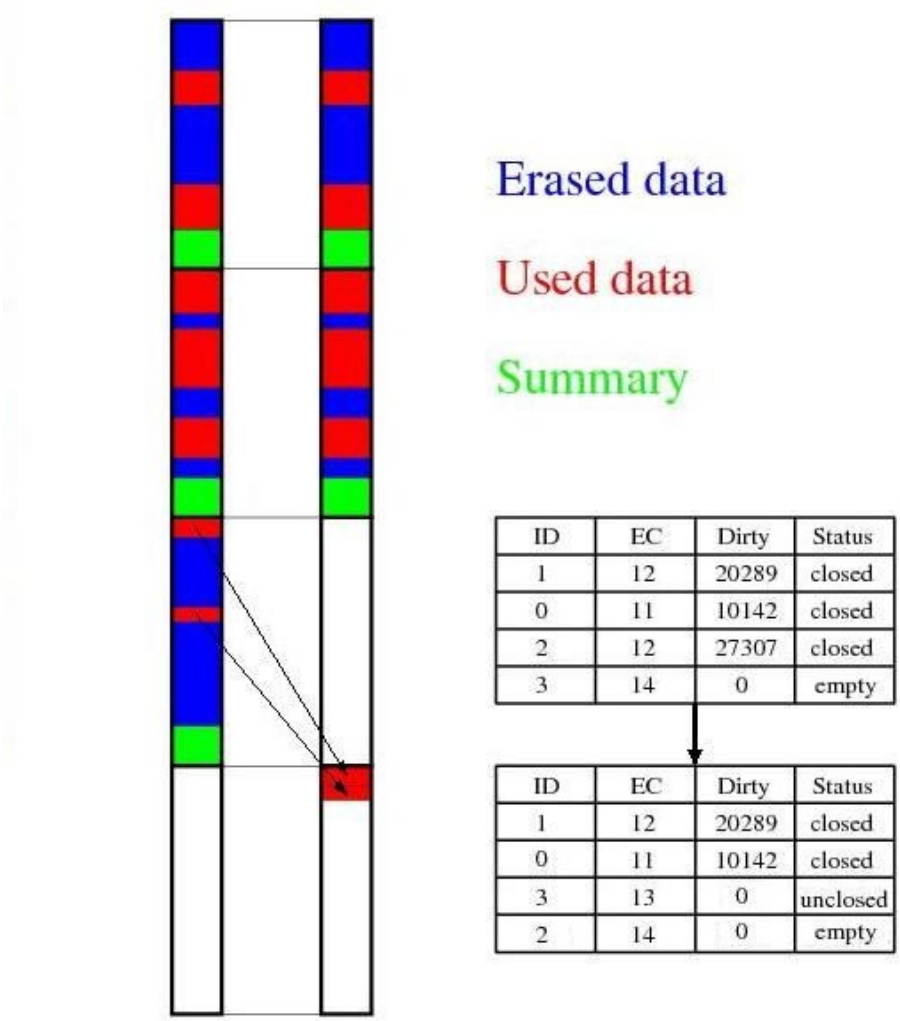
The MOVE mode scans all nodes in the region. It checks if it is in the B+ tree. If it is not in the B+ tree, it is a useless (obsoleted) node so it can be skipped. If it is in the tree, it calls tree delete and tree insert (to the actual writing region) to move the node.

Due to the wandering algorithm it produces not only free but dirty space, too. Therefore this mode is prohibited in the case of low space.

MIRROR mode

MIRROR mode is similar to MOVE mode, but it does not apply to any tree function. It is a special mode that can be processed in the background and guarantees that all the dirty space in the selected region will be freed. Let's see an example how region 2 is garbage collected.

1. Reserve an empty region, in this case it is region 3.
2. Check all nodes if they are in the B+ tree. The ones that are not, will be obsolete ones and can be skipped.
3. If the node is not obsolete (red), it has to be copied to the free region. It is important to keep its ordinal number so in the region-summary every deleted order will be 0xffffffff.
4. When all used node are copied, we have to exchange the two regions in the region-map (see figure). Due to this no wandering algorithms are necessary. The nodes will be found at their new place because its logical region number and its ordinal is the same as before the copying.
5. Old region (in this case region 2) can be freed.



ID	EC	Dirty	Status
1	12	20289	closed
0	11	10142	closed
2	12	27307	closed
3	14	0	empty

ID	EC	Dirty	Status
1	12	20289	closed
0	11	10142	closed
3	13	0	unclosed
2	14	0	empty

Selecting GC mode

The default GC mode is MIRROR mode. The disadvantage of it is that it can cause a fragmentation of the free space. To avoid it, sometimes the MOVE mode has to be applied.

Algorithms to select the mode: there is a *counter*. This counter can never be lower then 0. This counter is changed if the selection is done on the score of dirty space:

- If the highest dirty level is very low (there are too many nodes that are only copied and only a few of them can be deleted), the counter is increased by 4.
- If the highest dirty level is big enough, the counter will be reduced by 1.
- If the counter reaches 40, the next GC mode will be MOVE mode and the counter will be reduced by 30.

In the case of low space

There are 4 free regions reserved for MIRROR mode and journaling. If there are less than 5 free regions, only these two methods can reserve free region.

If the case of very low space only tree methods are allowed in order to make deletion possible.

Boot process

1. Determine the place of the super block using Artem's algorithm
2. Load region-map, its place is stored in the super block
3. Check the journal, replay the region-map and tree changes after the last commit
4. Scan all unclosed regions to collect summary

Ram consumption

The following data structures are held in the memory. Calculated memory consumption for 4G flash with 512K region size is represented in [].

- Logical superblock

```
struct jffs3_sb_info;

Used memory: 188 bytes
```

- region-map:

```
struct jffs3_region {
	uint32_t phy_num;		// physical region number
	uint32_t erase_cnt;		// erase count
	uint32_t dirty_size;		// size of the dirty data in the region
	uint32_t flags;			// unclosed, closed, empty, erasing
};

struct jffs3_region_map {
	struct jffs3_region *regions;			// regions array
	uint32_t reg_nr;				// number of the regions in map
	 struct jffs3_region_summary *next_region;	//current region
};

Used memory: 12 bytes + reg_nr * 16 bytes [12 bytes + 128K]
```

- Garbage Collector:

```
struct jffs3_gc {
	struct jffs3_region_summary *gc_reg;		// the selected region to gc
	struct jffs3_region_summary *gc_target_reg;	// where to gc (mirror)
	uint8_t gc_mode;
	uint32_t gc_pos;			// the gc position in region summary (move)
	uint32_t poorly_dirty;		// counter helps to select the gc mode
};

Used memory: 20 bytes
```

- Tree descriptor

```
struct tree_descriptor {
        uint32_t version;
        uint32_t keys_per_node;
        uint32_t links_per_node;
        uint16_t split_pos;
        uint32_t fillfactor;
        uint64_t root;
        int8_t depth;
        int8_t tree_depth;
        struct node *root_node;
};

Used memory: 36 bytes
```

- Treenode cache:

```
struct node {
        uint16_t type;
        uint16_t keys;
        uint64_t parent;
        uint64_t *key;
        uint64_t *link;
};

struct jffs3_treenodecache {
        struct node **cache;
        uint64_t *addr;
        uint64_t *index;
        int16_t *index_iid;
        int16_t *free_stack;
        int16_t stack_index;
        uint16_t cached_nodes;
        uint16_t max_cache_level;
        uint64_t next_vaddr;
        uint64_t start_vaddr;
        int16_t *parent;
        uint16_t *posinparent;
        uint8_t *flag;
        uint16_t *depends;
        int16_t *depth;
        int16_t depthbottom;
};

Used memory: 68 bytes + predefined treenode cache size
```

- Three write buffers:

```
struct jffs3_write_buffer {
        uint8_t *w_data;
        uint32_t w_len;
        uint64_t w_ofs;
        uint32_t w_pagesize;
};

Used memory: 3 *  (20 bytes + sector size)
```

- Journal:

```
struct jffs3_journal {
        uint32_t max_regions;
        uint8_t need_flush;
        uint8_t active_region;
        uint32_t *region;
        uint8_t *status;
        struct jffs3_region_summary **jrs;
        struct read_buffer *rbuf;
};

Used memory: 24 bytes
```

- Cached and unclosed regions

```
struct jffs3_summary_entry {
       uint32_t ofs;  // offset of the entry from the beginning of the region
       uint32_t len;  // length of the entry
};

struct jffs3_region_summary {
       uint32_t reg_nr;              // logical region number
       struct jffs3_summary_entry *sum_data;       // summary array
       uint32_t min_idx;             // GC
       uint32_t max_idx;             // for reserve space, the maximal index in use
       uint32_t free_space;          // free space in the region
       struct list_head list;        // list header for unclosed or cached list
};

Used memory: (28 bytes + regsum_default_size * 8 bytes) * num_of_unclosed
              [ (28 bytes + 64K) * num_of_unclosed ]
```

  Regsum_default_size is: region size divided by the size of jffs3_raw_inode and jffs3_raw_dirent, and it is gowned dynamically if necessary (there is a lot of very small node in the region)

First and last: the most relevant memory structures (in point of memory consumption) are the region-map, treenode cache, cached and unclosed regions.

Test implementation

We have realized a user-space implementation of this plan. This implementation is executed in a way which is very similar to the kernel environment, so most of the codes will be usable in the kernel variant, too. However, the debugging is much easier than in kernel-space.

This implementation has the following modules:

- **Super block management**: very simple, always stored in the first region (Artem has already implemented the final solution)
- **B+ tree**: full function
- **Journal**: every event is recorded, recovery implemented but needs more testing
- **Garbage collection**: full function (both MIRROR and MOVE mode)
- **Node management**: full function
- **Region-map**: full function
- **Flash emulation**: full function, flash delays are also emulated
- **Tester**: can be controlled by its own script language

The implemented functionalities:

- Create files / directories (jffs3_create / jffs3_mkdir)
- Delete files / directories (jffs3_unlink / jffs3_rmdir)
- Read / write files (jffs3_read_page / jffs3_commit_write)
- Mount / unmount filesystem (jffs3_get_sb / jffs3_kill_sb)
- Garbage collection (only when low space detected)
- Recovery from power loss

Test results

Emulated flash: Samsung K9F1G08R0A, using the following delays to emulate flash chip operations:

- ACCESS_DELAY = 50 μsec
- PROGRAM_DELAY = 200 μsec
- ERASE_DELAY = 2000 μsec

Flash Memory Size	Region Size (kiB)	Tree Node Cache Memory (kiB)	Region Summary Cache Size	Read Rate (kB/sec)	Fill Rate (kB/sec)	*GC Fill Rate (kB/sec)*	Maximal Memory Usage (kB)
256	128	128	5	19459	9390	1016	392
256	128	128	10	16480	7830	836	474
256	128	128	20	16519	7819	836	638
256	128	128	30	16107	7986	854	802
256	128	512	5	20469	8544	863	792
256	128	512	10	20412	8547	861	874
256	128	512	20	20815	8605	869	1039
256	128	512	30	20458	8543	862	1203
256	128	1024	5	20897	8528	863	1320
256	128	1024	10	21009	8519	864	1402
256	128	1024	20	20976	8541	863	1567
256	128	1024	30	19340	8612	872	1731
256	256	128	5	12113	7187	823	537
256	256	128	10	12675	7186	823	701
256	256	128	20	12758	7184	824	1029
256	256	128	30	13415	7186	824	1357
256	256	512	5	19490	8486	893	933
256	256	512	10	19553	8471	892	1097
256	256	512	20	19578	8490	892	1425
256	256	512	30	19517	8497	893	1753
256	256	1024	5	20386	8729	901	1461
256	256	1024	10	20308	8739	902	1625
256	256	1024	20	20795	8829	910	1953
256	256	1024	30	18977	8820	911	2281
256	512	128	5	9809	6088	701	907
256	512	128	10	9925	6081	701	1173
256	512	128	20	9407	6125	706	1820
256	512	128	30	9534	6084	702	2476
256	512	512	5	17447	8193	835	1303
256	512	512	10	17644	8188	834	1631
256	512	512	20	18142	8215	835	2286
256	512	512	30	17758	8172	834	2942
256	512	1024	5	18640	8512	851	1831
256	512	1024	10	18544	8495	851	2159
256	512	1024	20	18404	8511	850	2814
256	512	1024	30	19353	8496	850	3470
512	512	1024	30	17420	7255	857	3478
1024	512	1024	30	18438	7751	853	3495
2048	512	1024	30	18730	8506	957	3538

Flash raw read speed: 40M/sec

Flash raw write speed: 10M/sec

Columns

1. Read rate (after filling and GC-ing)
2. Fill rate without GC
3. Fill rate with GC continuously in action
4. The maximal used memory by the JFFS3 (without the flash emulator and the test scrip runner program)

The benchmark consists the following steps (for 256M flash):

- write[4] 17 * 10MB data
- write 20 * 2MB data
- write 40 * 512KB data
- write 80 * 128KB data
- write 160 * 10KB data
- write 320 * 1KB data
- delete every second file (make dirty space for GC and fragment the filesystem)
- write the deleted files again (test for write through garbage collection)
- copy all the files to outer filesystem (read benchmark)

4 "write" means creating the file and fill it up with data