JFFS : The Journalling Flash File System

David Woodhouse
Red Hat, Inc.
dwmw2@cambridge.redhat.com

Abstract

Until recently, the common approach to using Flash memory technology in embedded devices has been to use a pseudo-filesystem on the flash chips to emulate a standard block device and provide wear levelling, and to use a normal file system on top of that emulated block device.

JFFS is a log-structured file system designed by Axis Communications AB in Sweden specifically for use on flash devices in embedded systems, which is aware of the restrictions imposed by flash technology and which operates directly on the flash chips, thereby avoiding the inefficiency of having two journalling file systems on top of each other.

This paper will give an overview of the restrictions imposed by flash technology and hence the design aims of JFFS, and the implementation of both JFFS and the improvements made in version 2, including compression and more efficient garbage collection.

1 Introduction

1.1 Flash

Flash memory is an increasingly common storage medium in embedded devices, because it provides solid state storage with high reliability and high density, at a relatively low cost.

Flash is a form of Electrically Erasable Read Only Memory (`EEPROM`), available in two major types — the traditional NOR flash which is directly accessible, and the newer, cheaper NAND flash which is addressable only through a single 8-bit bus used for both data and addresses, with separate control lines.

These types of flash share their most important characteristics — each bit in a clean flash chip will be set to a logical one, and can be set to zero by a write operation.

Flash chips are arranged into blocks which are typically 128KiB on NOR flash and 8KiB on NAND flash. Resetting bits from zero to one cannot be done individually, but only by resetting (or "erasing") a complete block. The lifetime of a flash chip is measured in such erase cycles, with the typical lifetime being 100,000 erases per block. To ensure that no one erase block reaches this limit before the rest of the chip, most users of flash chips attempt to ensure that erase cycles are evenly distributed around the flash; a process known as "wear levelling".

Aside from the difference in erase block sizes, NAND flash chips also have other differences from NOR chips. They are further divided into "pages" which are typically 512 bytes in size, each of which has an extra 16 bytes of "out of band" storage space, intended to be used for metadata or error correction codes. NAND flash is written by loading the required data into an internal buffer one byte at a time, then issuing a write command. While NOR flash allows bits to be cleared individually until there are none left to be cleared, NAND flash allows only ten such write cycles to each page before leakage causes the contents to become undefined until the next erase of the block in which the page resides.

1.2 Flash Translation Layers

Until recently, the majority of applications of flash for file storage have involved using the flash to emulate a block device with standard 512-byte sectors, and then using standard file systems on that emulated device.

The simplest method of achieving this is to use a

simple 1:1 mapping from the emulated block device to the flash chip, and to simulate the smaller sector size for write requests by reading the whole erase block, modifying the appropriate part of the buffer, erasing and rewriting the entire block. This approach provides no wear levelling, and is extremely unsafe because of the potential for power loss between the erase and subsequent rewrite of the data. However, it is acceptable for use during development of a file system which is intended for read-only operation in production models. The `mtdblock` Linux driver provides this functionality, slightly optimised to prevent excessive erase cycles by gathering writes to a single erase block and only performing the erase/modify/writeback procedure when a write to a different erase block is requested.

To emulate a block device in a fashion suitable for use with a writable file system, a more sophisticated approach is required.

To provide wear levelling and reliable operation, sectors of the emulated block device are stored in varying locations on the physical medium, and a "Translation Layer" is used to keep track of the current location of each sector in the emulated block device. This translation layer is effectively a form of journalling file system.

The most common such translation layer is a component of the PCMCIA specification, the "Flash Translation Layer" [FTL]. More recently, a variant designed for use with NAND flash chips has been in widespread use in the popular DiskOnChip devices produced by M-Systems.

Unfortunately, both FTL and the newer NFTL are encumbered by patents — not only in the United States but also, unusually, in much of Europe and Australia. M-Systems have granted a licence for FTL to be used on all PCMCIA devices, and allow NFTL to be used only on DiskOnChip devices.

Linux supports both of these translation layers, but their use is deprecated and intended for backwards compatibility only. Not only are there patent issues, but the practice of using a form of journalling file system to emulate a block device, on which a "standard" journalling file system is then used, is unnecessarily inefficient.

A far more efficient use of flash technology would be permitted by the use of a file system designed specifically for use on such devices, with no extra layers of translation in between. It is precisely such a filesystem which Axis Communications AB released in late 1999 under the GNU General Public License.

2 JFFS Version 1

The design goals of JFFS are largely determined by the characteristics of flash technology and of the devices in which it is expected to be used — as embedded and battery-powered devices are often treated by users as simple appliances, we must ensure reliable operation when the system is uncleanly shut down.

2.1 Storage Format

The original JFFS is a purely log-structured file system [LFS]. Nodes containing data and metadata are stored on the flash chips sequentially, progressing strictly linearly through the storage space available.

In JFFS v1, there is only one type of node in the log; a structure known as `struct jffs_raw_inode`. Each such node is associated with a single inode. It starts with a common header containing the inode number of the inode to which it belongs and all the current file system metadata for that inode, and may also carry a variable amount of data.

There is a total ordering between the all the nodes belonging to any individual inode, which is maintained by storing a `version` number in each node. Each node is written with a `version` higher than all previous nodes belonging to the same inode. The `version` is an unsigned 32-bit field, allowing for 4 milliard nodes to be written for each inode during the life span of the file system. Because the limited lifetime of flash chips means this number is extremely unlikely to be reached, this limitation is deemed to be acceptable.

Similarly, the inode number is stored in a 32-bit field, and inode numbers are never reused. The same logic applies to the acceptability of this limitation, especially as it is possible to remove this restriction without breaking backwards compatibility of JFFS file systems, if it becomes necessary.

In addition to the normal inode metadata such as

uid, gid, mtime, atime, mtime etc., each JFFS v1 raw node also contains the name of the inode to which it belongs and the inode number of the parent inode.[1]

Each node may also contain an amount of data, and if data are present the node will also record the offset in the file at which these data should appear. For reasons which are discussed later, there is a restriction on the maximum size of physical nodes, so large files will have many nodes associated with them, each node carrying data for a different range within the file.

Nodes containing data for a range in the inode which is also covered by a later node are said to be obsoleted, as are nodes which contain no data, where the metadata they contain has been outdated by a later node. Space taken by obsoleted nodes is referred to as *"dirty space"*.

Special inodes such as character or block devices and symbolic links which have extra information associated with them represent this information — the device numbers or symlink target string — in the data part of the JFFS node, in the same manner as regular files represent their data, with the exception that there may be only one non-obsolete node for each such special inode at any time. Because symbolic links and especially device nodes have small amounts of such data, and because the data in these inodes are always required all at once rather than by reading separate ranges, it is simpler to ensure that the data are not fragmented into many different nodes on the flash.

Inode deletion is performed by setting a deleted flag in the inode metadata. All later nodes associated with the deleted inode are marked with the same flag, and when the last file handle referring to the deleted inode is closed, all its nodes become obsolete.

2.2 Operation

The entire medium is scanned at mount time, each node being read and interpreted. The data stored in the raw nodes provide sufficient information to rebuild the entire directory hierarchy and a complete map for each inode of the physical location on the medium of each range of data.

[1] The lack of distinction between directory entries and inodes means that the original JFFS cannot support hard links.

JFFS v1 stores all this information at all times while the file system is mounted. Each directory lookup can be satisfied immediately from data structures held in-core, and file reads can be performed by reading immediately from the appropriate locations on the medium into the supplied buffer.

Metadata changes such as ownership or permissions changes are performed by simply writing a new node to the end of the log recording the appropriate new metadata. File writes are similar; differing only in that the node written will have data associated with it.

2.3 Garbage Collection

The principles of operation so far are extremely simple. The JFFS code happily writes out new jffs_raw_inode structures to the medium to mark each change made to the filesystem... until, that is, it runs out of space.

At that point, the system needs to begin to reclaim the *dirty space* which contains old nodes which have been obsoleted by subsequent writes.

The oldest node in the log is known as the head, and new nodes are added to the tail of the log. In a clean filesystem which on which garbage collection has never been triggered, the head of the log will be at the very beginning of the flash. As the tail approaches the end of the flash, garbage collection will be triggered to make space.

Garbage collection will happen either in the context of a kernel thread which attempts to make space before it is actually required, or in the context of a user process which finds insufficient free space on the medium to perform a requested write. In either case, garbage collection will only continue if there is dirty space which can be reclaimed. If there is not enough dirty space to ensure that garbage collection will improve the situation, the kernel thread will sleep, and writes will fail with −ENOSPC errors.

The goal of the garbage collection code is to erase the first flash block in the log. At each pass, the node at the head of the log is examined. If the node is obsolete, it is skipped and the head moves on to

the next node.[2] If the node is still valid, it must be rendered obsolete. The garbage collection code does so by writing out a new data or metadata node to the tail of the log.

The new node written will contain the currently valid data for at least the range covered by the original node. If there is sufficient free space, the garbage collection code may write a larger node than the one being obsoleted, in order to improve storage efficiency by merging many small nodes into fewer, larger nodes.

If the node being obsoleted is already partially obsoleted by later nodes which cover only part of the same range of data, some of the data written to the new node will obviously differ from the data contained in the original.

In this way, the garbage collection code progresses the `head` of the log through the flash until a complete erase block is rendered obsolete, at which point it is erased and becomes available for reuse by the `tail` of the log.

2.4 Housekeeping

The JFFS file system requires a certain amount of space to be available between the `head` and the `tail` of the log at all times, in order to ensure that it is always possible to proceed with garbage collection by writing out new nodes as described above.

A simplified analysis of this situation is as follows:

In order to be able to erase the next block from the `head` of the log, there must be sufficient space to write out new nodes to obsolete all the nodes in that block. The worst case is that all nodes in the block are valid, the first node starts at the very beginning of the block, and the final node starts just before the end of the block and extends into the subsequent block.

By restricting the maximum size of a data node to half the size of the flash erase sector, we limit the amount of free space required in this worst case scenario to one and a half times the size of the flash sectors in use.

In fact, the above is only an approximation — it ignores the fact that a name is stored with each node on the flash, and that renaming a file to a longer name will cause all nodes belonging to that file to grow when they are garbage collected.[3]

The precise amount of space which is required in order to ensure that garbage collection can continue is not formally proven and may not even be bounded with the current algorithms.

Empirical results show that a value of four flash sectors seems to be sufficient, while the previous default of two flash sectors would occasionally lead to the `tail` of the log reaching the `head` and complete deadlock of the file system.

2.5 Evolution

The original version of JFFS was used by Axis in their embedded devices in a relatively limited fashion, on 2.0 version of the Linux kernel.

After the code was released, it was ported to the 2.3 development kernels by a developer in Sweden. Subsequently, Red Hat, Inc. were asked to port it to the 2.2 series and provide commercial support for a contract customer.

Although the design of the file system was impressive, certain parts of the implementation appeared not to have been tested by its use in Axis' products. Writing data anywhere other than at the end of a file did not work, and deleting a file while a process had a valid file descriptor for it would cause a kernel oops.

After some weeks of reliability and compliance testing, JFFS reached stability. It is a credit to the clarity and quality of the original code that it was possible to become familiar with it and bring it to the current state in a relatively short space of time.

[2] Actually, if the node was obsoleted the reference to it would already have been removed from the linked list of nodes which JFFS stores. The `head` pointer only ever points to a valid node. This is an implementation detail, though. The point is that valid nodes are obsoleted, and obsoleted nodes are ignored — either explicitly or implicitly.

[3] An attempt was made to limit this growth by counting the number of valid nodes containing the current name of each file, and writing out a name with a new node only if there were fewer than two such nodes. This attempt was abandoned because the initial implementation was buggy and could lead to a situation with no valid copies of a file name, and because it would not have solved the problem properly even if the hard-to-find bugs were located and fixed.

However, during this time it became apparent that there were a few serious flaws in the original implementation of the filesystem:

Garbage collection would proceed linearly through the medium, writing out new nodes to allow it to erase the oldest block in the log, even if the block being garbage collected contained only clean nodes.

In the relatively common situation where a 16 MiB file system contained 12 MiB of static data — libraries and program executables, 2 MiB of slack space and 2 MiB of dynamic data, the garbage collection would move the 12 MiB of static data from one place on the flash to another on every pass through the medium. JFFS provided perfect wear *levelling* — each block was erased exactly the same number of times — but this meant that the blocks were also erased more often than was necessary.

Wear levelling must be provided, by occasionally picking on a clean block and moving its contents. But that should be an *occasional* event, not the normal behaviour.

Compression was not supported by JFFS. Because of the cost of flash chips and the constant desire to squeeze more functionality into embedded devices, compression was a very important requirement for a large proportion of potential users of JFFS.

Hard links were also not supported by the original version of the filesystem. While this lack was not particularly limiting, it was annoying, as was the fact that file names were stored with each `jffs_raw_inode`, potentially leading to unbounded space expansion upon renames.

3 JFFS2

In January of 2001, another customer required compression support in JFFS to be provided as part of a contract undertaken. After a period of discussion on the mailing list, it was concluded that the most appropriate course of action would be a complete reimplementation, allowing all of the above-mentioned deficiencies in the original implementation to be addressed.

The JFFS2 code was intended to be portable, in particular to eCos, Red Hat's embedded operating system targetted at consumer devices[eCos]. For this reason, JFFS2 is released under a dual licence — both under GPL and the MPL-style "Red Hat eCos Public License", to be compatible with the licence of the remainder of the eCos source.

Although portability was intended, no ports have yet been completed, and the current code is only usable with the 2.4 series of Linux kernels.

3.1 Node Format and Compatibility

While the original JFFS had only one type of node on the medium, JFFS2 is more flexible, allowing new types of node to be defined while retaining backward compatibility through use of a scheme inspired by the compatibility bitmasks of the ext2 file system.

Every type of node starts with a common header containing the full node length, node type and a cyclic redundancy checksum (CRC). The common node structure is shown in Figure 1.

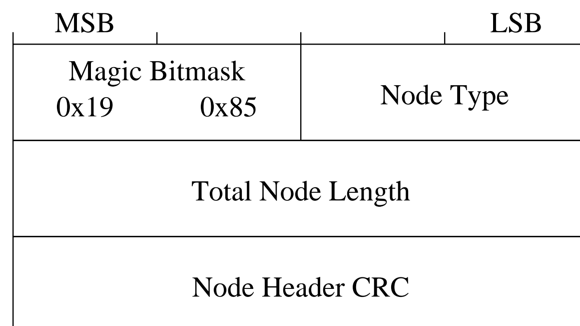

Figure 1: JFFS2 Common Node Header.

In addition to a numeric value uniquely identifying the node structure and meaning, the node type field also contains a bitmask in the most significant two bits which indicates the behaviour required by a kernel which does not support the node type used:

`JFFS2_FEATURE_INCOMPAT` — on finding a node with this feature mask which is not explicitly supported, a JFFS2 implementation must refuse to mount the file system.

`JFFS2_FEATURE_ROCOMPAT` — a node with this feature mask may be safely ignored by an implemen-

tation which does not support it, but the file system must not be written to.

`JFFS2_FEATURE_RWCOMPAT_DELETE` — an unsupported node with this mask may be safely ignored and the file system may be written to. Upon garbage collecting the sector in which it is found, the node should be deleted.

`JFFS2_FEATURE_RWCOMPAT_COPY` — an unsupported node with this mask may be safely ignored and the file system may be written to. Upon garbage collecting the sector in which it is found, the node should be copied intact to a new location.

It is an unfortunate matter of record that this compatibility bitmask was in fact the reason why it was necessary to *break* compatibility with older JFFS2 file systems. Originally, the CRC was omitted from the common node header, and it was discovered that because the `INCOMPAT` feature mask has more bits set than the other bitmasks, it is relatively easy, by interrupting erases, to accidentally generate a structure on the medium which looks like an unknown node with the `INCOMPAT` feature bit set. For this reason, a CRC on the node header was added, in addition to the existing CRCs on the node contents and on the data or name field if present.

3.2 Log Layout and Block Lists

Aside from the differences in the individual nodes, the high-level layout of JFFS2 also changed from a single circular log format, because of the problem caused by strictly garbage collecting in order. In JFFS2, each erase block is treated individually, and nodes may not overlap erase block boundaries as they did in the original JFFS.

This means that the garbage collection code can work with increased efficiency by collecting from one block at a time and making intelligent decisions about which block to garbage collect from next.

Each erase block may be in one of many states, depending primarily on its contents. The JFFS2 code keeps a number of linked lists of structures representing individual erase blocks. During the normal operation of a JFFS2 file system, the majority of erase blocks will be on the `clean_list` or the `dirty_list`, which represent blocks full of valid nodes and blocks which contain at least one obsoleted node, respectively. In a new filesystem, many erase blocks may be on the `free_list`, and will contain only one valid node — a marker which is present to show that the block was properly and completely erased.

As mentioned previously, the garbage collection code uses the lists to choose a sector for garbage collection. A very simple probabilistic method is used to determine which block should be chosen — based on the `jiffies` counter. If `jiffies % 100` is nonzero, a block is taken from the `dirty_list`. Otherwise, on the one-in-one-hundred occasions that the formula is zero, a block is taken from the `clean_list`. In this way, we optimise the garbage collection to re-use blocks which are already partially obsoleted, but over time, we still move data around on the medium sufficiently well to ensure that no one erase block will be worn out before the others.

3.3 Node Types

The third major change in JFFS2 is the separation between directory entries and inodes, which allows JFFS2 to support hard links and also removes the problem of repeating name information which was referred to in the footnote on page 4.

At the time of writing there are three types of nodes defined and implemented by JFFS2. These are as follows:

`JFFS2_NODETYPE_INODE` — this node is most similar to the `struct jffs_raw_inode` from JFFS v1. It contains all the inode metadata, as well as potentially a range of data belonging to the inode. However, it no longer contains a file name or the number of the parent inode. As with traditional UNIX-like file systems, inodes are now entirely distinct entities from directory entries. An inode is removed when the last directory entry referring to it has been unlinked, and there it has no open file descriptors.

Data attached to these nodes may be compressed using one of many compression algorithms which can be plugged into the JFFS2 code. The simplest types are "none" and "zero", which mean that the data are uncompressed, or that the data are all zero, respec-

tively. Two compression algorithms were developed specifically for use in JFFS2, and also the JFFS2 code can contain yet another copy of the zlib compression library which is already present in at least three other places in the Linux kernel source.[4]

In order to facilitate rapid decompression of data upon `readpage()` requests, nodes contain no more than a single page of data, according to the hardware page size on the target platform. This means that in some cases JFFS2 filesystem images are not portable between hosts, but this is not a serious problem because the nature of the flash storage medium makes transportation between devices unlikely. JFFS2 is also entirely host-endian in its storage of numbers larger than a single byte.

`JFFS2_NODETYPE_DIRENT` — this node represents a directory entry, or a *link* to an inode. It contains the inode number of the directory in which the link is found, the name of the link and the inode number of the inode to which the link refers. The `version` number in a dirent node is in the sequence of the parent inode. A link is removed by writing a dirent node with the same name but with target inode number zero — and obviously a higher `version`.

POSIX requires that upon renaming, for example "`passwd.new`" to "`passwd`", the replacement of the `passwd` link should be atomic — there should not be any time at which a lookup of that name shall fail to return either the old or the new target. JFFS2 meets that requirement, although as with many other file systems, the *entire* rename operation is not atomic.

Renaming is performed in two stages. First a new dirent node is written, with the new name and the inode number of the inode being renamed. This atomically replaces the link to the original inode with the new one, and is identical to the way in which a hard link is created. Next, the original name is unlinked, by writing a dirent node with the original name and target inode number zero.

This two-stage process means that at some point during the rename operation, the inode being renamed into place is accessible through *both* the old and the new names. This behaviour is permitted by POSIX — the atomicity guarantee required is for the behaviour of the *target* link only.

`JFFS2_NODETYPE_CLEANMARKER` — this node is written to a newly erased block to show that the erase operation has completed successfully and the block may safely be used for storage.

The original JFFS simply assumed that any block which appeared at first scan to contain `0xFF` in every byte was free, and would select the longest run of apparently free space at mount time to be the space between the `head` and `tail` of the log. Unfortunately, extensive power fail testing on JFFS proved this to be unwise. For many types of flash chips, if power is lost during an erase operation, some bits may be left in an unstable state, while most are reset to a logical one. If the initial scan happens to read all ones and treat a block containing such unstable bits as usable, then data may be lost — and such data loss may not even be avoidable by the naïve method of verification by reading back data immediately by writing, because the bit may just happen to return the correct value when read back for verification.

Empirical results showed that even rereading the entire contents of the block multiple times in an attempt to detect unstable bits was not sufficiently reliable to avoid data loss, so an alternative approach was required. The accepted solution was to write the marker node to the flash block immediately after successful completion of an erase operation. Upon encountering flash blocks which do not appear to contain any valid nodes, JFFS2 will trigger an erase operation and subsequently write the appropriate marker node to the erased block.

This node type was introduced after JFFS2 had started to be used in real applications, and uses the `RWCOMPAT_DELETE` feature bitmask to signify that an older JFFS2 implementation may safely ignore the node.

3.4 Operation

The operation of JFFS2 is at a fundamental level very similar to that of the original JFFS — nodes, albeit now of various types, are written out sequentially until a block is filled, at which point a new block is taken from the `free_list` and writing continues from the beginning of the new block.

[4] This duplication is scheduled to be fixed fairly early during the 2.5 development series.

When the size of the free_list reaches a heuristic threshold, garbage collection starts, moving nodes from an older block into the new block until space can be reclaimed by erasing the older one.

However, JFFS2 does not keep all inode information in core memory at all times. During mount, the full map is built as before — but the structures kept in memory are strictly limited to the information which cannot be recreated quickly on-demand. For each inode on the medium, there is a struct jffs2_inode_cache which stores its inode number, the number of current links to the inode, and a pointer to the start of a linked list of the physical nodes which belong to that inode. These structures are stored in a hash table, with each hash bucket containing a linked list. The hash function is a very primitive one - merely the inode number modulo the size of the hash table. The distribution of inode numbers means this should be well-distributed.[5]

Each physical node on the medium is represented by a smaller struct jffs2_raw_node_ref, also shown in Figure 2, which contains two pointers to other raw node references — the next one in the physical erase block and the next one in the per-inode list — and also the physical offset and total length of the node. Because of the number of such structures and the limited amount of RAM available on many embedded systems, this structure is extremely limited in size.

Because all nodes on the medium are aligned to a granularity of four bytes, the least significant two bits of the flash_offset field are redundant. They are therefore available for use as extra flags. The least significant bit is set to indicate that the node represented is an obsolete node, and the other is not yet used.

For garbage collection, it is necessary to find, given a raw node reference, the inode to which it belongs. It is preferable not to add four bytes containing this information to every such structure, so instead we play even more evil games with the pointers. Rather than having a NULL-terminated linked list for the next_in_ino list, the last raw node reference actually contains a pointer to the struct jffs2_inode_cache for the relevant inode. Because

[5] The size of the hash table is variable at compile time, and in all cases is currently only one entry - which effectively means that all inode cache structures are stored in a single linked list. If and when this becomes noticeably suboptimal, it will be simple to correct.

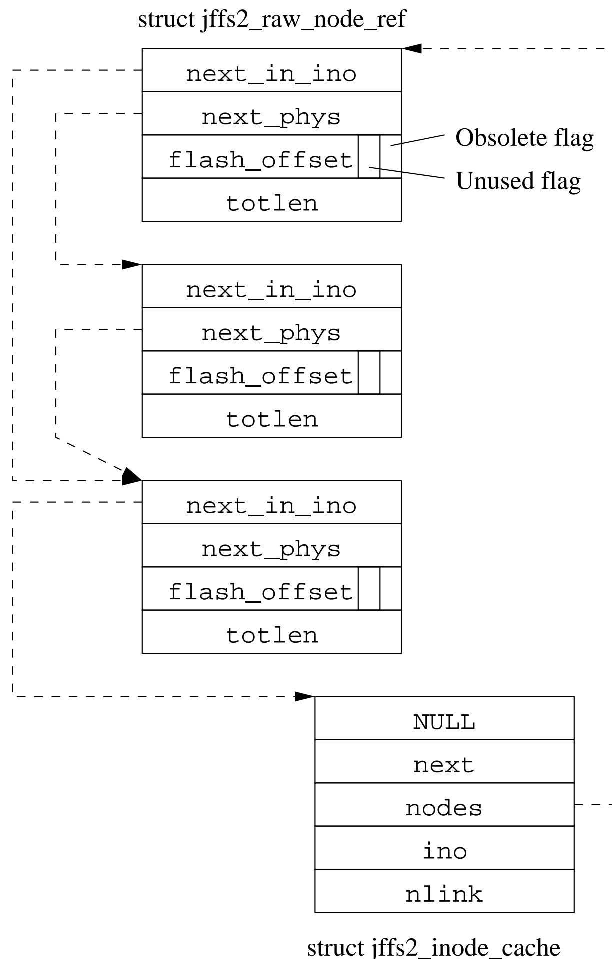

Figure 2: Raw Node Reference Lists

that structure has a NULL at the offset where the struct jffs2_raw_node_ref would the pointer to the next node in the inode, the code traversing the list knows it has reached the end, at which point the pointer can be cast to the appropriate type and the inode number and other information can be read from the structure.

The NULL field shown in the inode cache structure is used only during the initial scan of the filesystem for temporary storage, and hence can be guaranteed to be NULL during normal operation.

During normal operation, the file system's read_inode() method is passed an inode number and is expected to populate a struct inode with appropriate information. JFFS2 uses the inode number to look up the appropriate struct jffs2_inode_cache in a hash table, then uses the list of nodes to directly read each node which

belongs to the required inode, thereby building up a complete map of the physical locations of each range of the inode's data, similar to the information which JFFS would have kept in memory even while it was unused.

Once the full inode structure has been populated in this manner, it remains in memory until the kernel later tries to prune its inode cache under memory pressure, at which point the extra information is freed, leaving only the raw node references and the minimal JFFS2 inode cache structure which were originally present.

3.5 Mounting

Mounting a JFFS2 file system involves a four-stage operation. First, the physical medium is scanned, the CRCs on all the nodes are checked for validity, and the raw node references are allocated. During this stage, the inode cache structures are also allocated and inserted into the hash table for each inode for which valid nodes are found.

Extra information from the nodes on the flash is cached, such as the `version` and the range of data covered by the node, to prevent the subsequent stages of the mount process from having to read anything again from the physical medium.

After the physical scan is complete, a first pass through all the physical nodes is made, building a full map of the map of data for each inode so that obsoleted nodes can be detected as such, and increasing the `nlink` field in the inode cache of the linked inode for each valid directory entry node.

A second pass is then made to find inodes which have no remaining links on the file system and delete them. Each time a directory inode is deleted, the pass is restarted, as other inodes may have been orphaned. In future, this behaviour may be modified to store orphaned inodes in a `lost+found` directory instead of just removing them.

Finally, a third pass is made to free the temporary information which was cached for each inode; leaving only the information which is normally kept in the `struct jffs2_inode_cache` during operation. In doing so, the field in the inode cache which corresponds to the `next_in_ino` field of the raw node reference is set to `NULL`, thereby enabling the slightly evil hack referred to earlier, for detecting that the end of the `next_in_ino` list has been reached.

3.6 Garbage Collection

In JFFS2, garbage collection moves data nodes by determining the inode to which the node to be garbage collected belongs, and calling the Linux kernel's `iget()` function for the inode in question. Often, the inode will be in the kernel's inode cache — but sometimes, this will cause a call to the JFFS2 `read_inode()` function as described above.

Once the full inode structure is obtained, a replacement node can be written to obsolete the original node. If it was a data node, the garbage collection routine calls the standard `readpage()` function for the page for which the node contains data — again using the existing file system caching mechanism because the required page may already be in the page cache. Then, as much of the page as possible is recompressed and written out in a new node. A partial page may be written if the node to be garbage collected is small and there is not sufficient slack space to allow a full page to be written, or if the page being garbage collected is at the end of the inode.

One of the features which is strongly desired for JFFS2 is a formal proof of correctness of the garbage collection algorithm. The current empirical method is not sufficient. The compression, however, gives rise to a serious potential problem with this proof. If a full page is written which compresses extremely well, and later a single byte is written in the middle of the page which reduces the compressibility of the page, then when garbage collecting the original page we may find that the new node written out is *larger* than the original. Thus, there would be no way to place an upper bound on the amount of space required to garbage collect an erase block full of data.

The proposed solution to this is to allow the total ordering of the `version` field to be relaxed to a partial ordering. We allow two nodes to have the *same* `version` field as long as they have identical data. Thus, when garbage collection finds a node which would expand, yet insufficient slack space to allow it to do so, it may copy the original node intact, preserving the original version so that the nodes which overlay the data contained therein will still continue

to do so.

3.7 Truncation and File Holes

A problem which arose during the design stage for JFFS2 which had not already been addressed for the original version involved truncation of files. The sequence of events which could be problematic was a truncation followed by a write to an offset larger than the truncation point — leaving a "hole" in the file which should return all zeroes upon being read.

On truncation, the original JFFS merely wrote out a new node giving the new length, and marked (in memory) the older nodes containing data beyond the truncation point as obsolete. Later writes would occur as normal.

During a scan of the file system on remounting, the sequential nature of the garbage collection ensured that all the old nodes containing actual data for the ranges which should be "holes" were garbage collected before the truncation node. As nodes were interpreted in `version` order after the physical scan, correct behaviour could be guaranteed, because the evidence of the truncation was still present at all times until the old data were erased.

For JFFS2, where blocks can be garbage collected out of order, it was necessary to ensure that old data could never "show through" the holes caused by truncation and subsequent extension of a file.

For this reason, it was decided that there should be no holes in the proper sense — a complete absence of information for the range of bytes in question. Instead, upon receiving a request to write to an offset greater than the current size of a file, or a request to truncate to a larger size, JFFS2 inserts a data node with the previously-mentioned compression type `JFFS2_COMPR_ZERO`, meaning that no actual data are contained with the node, and the entire range represented by the node should be set to zero upon being read.

In the case where a file contains a very large hole, it is preferable to represent that hole by only a single physical node on the medium, rather than a "hole" node for each page in the range affected. Therefore, such hole nodes are a special case of data node; the only type of data node which may cover a range of more than one page.

This special case is itself the reason for further complication, because of concerns about expansion during garbage collection. If a single byte is written to a page which was previously part of a hole, it is necessary to ensure that garbage collection of either the original hole node or the node containing the new byte of data should not require more space than is taken by the original.

The solution to this problem is the same as for compressed nodes which may expand when merged — if garbage collection would cause an expansion, and there is insufficient slack space to accommodate such growth, then the original node is copied exactly, retaining the original `version` number.

4 Future Development

One oft-requested feature which is currently *not* planned for development in JFFS2 is eXecute In Place (XIP) functionality. When programs are run from JFFS2, the executable code is copied from the flash into RAM before the CPU can execute it. Likewise, even when the `mmap()` system call is used, data are not accessed directly from the flash but are copied into RAM when required.

XIP functionality in JFFS2 is not currently planned because it is fairly difficult to implement and because the potential benefits of XIP are not clearly sufficient to justify the effort required to do so.

For obvious reasons, XIP and compression are mutually exclusive - if data are compressed, they cannot be used directly in place. Given a prototype platform with sufficient quantities of both RAM and flash that neither XIP or compression are required, and the desire to save money on the hardware, a choice can be made between halving the amount of RAM and using XIP, or halving the amount of flash and using compression.

By choosing the latter option, the cost saving will generally be greater than the former option, because flash is more expensive than RAM. The operating system is able to be more flexible in its use of the available RAM, discarding file buffers during periods of high memory pressure. Furthermore, because write operations to flash chips are so slow, compressing the data may actually be faster for many workloads.

The main problem with XIP, however, is the interaction with memory management hardware. Firstly, for all known memory management units, each page of data must be exactly page-aligned on the flash chip in order for it to be mapped into processes address space – which makes such a file system even more wasteful of space than the mere absence of compression already implies. Secondly, while giving write or erase commands to a flash chip, it may return status words on all read cycles, therefore all currently valid mappings of the pages of the chip would have to be found and invalidated for the duration of the operation. These two limitations make a writable filesystem with XIP functionality extremely difficult to implement, and it is unlikely that JFFS2 could support XIP without fundamental changes to its design.

An read-only XIP filesystem would be a more reasonable request, and an entirely separate file system providing this functionality, based on the existing ROMFS file system, is likely to be developed at some time in the near future.

4.1 Improved Fault Tolerance

The main area where JFFS2 still requires development is in fault tolerance. There are still areas where, although designed to be resilient, JFFS2 may exhibit a more serious failure mode than is absolutely necessary given a physical error.

In particular, JFFS2 will need more sophisticated methods of dealing with single-bit errors in flash chips. Currently, the node contains a 32-bit CRC, but this only gives error detection; it does not allow the file system to correct errors. Error correction is an absolute requirement for operation on NAND flash chips, which have lower tolerances. It is desirable even on NOR flash.

JFFS2 already has a primitive method of dealing with blocks for which errors are returned by the hardware driver — it files them on a separate `bad_list` and refuses to use them again until the next time the file system is remounted. This should be developed.

4.2 Garbage Collection Space Requirements

A major annoyance for users is the amount of space currently required to be left spare for garbage collection. It is hoped that a formal proof of the amount of required space can be produced, but in the meantime a very conservative approach is taken — five full erase blocks must be available for writing before new writes will be permitted from user space.

It should be possible to reduce this figure significantly — hopefully to a single block for NOR flash and to two or three blocks in the case of NAND flash, where extra space should always be available to copy away data from bad blocks.

The approach to this problem in JFFS1 was to evaluate and attempt to prove an upper bound on the amount of space required. This appeared to fail because there appeared to be no such upper bound. For JFFS2, it is suspected that a more useful approach may be to define a reasonable upper bound, such as a single erase block, and to modify the code to make it true.

4.3 Transaction Support

For storing database information in JFFS2 file systems, it may be desirable to expose transactions to user space. It has been argued that user space can implement transactions itself, using only the file system functionality required by POSIX. This is true — but implementing a transaction-based system on top of JFFS2 would be far less efficient than using the existing journalling capability of the file system; for the same reason that emulating a block device and then using a standard journalling file system on top of that was considered inadequate.

Little work — and relatively little thought — has gone into this subject with respect to JFFS2, yet at first consideration it seems that to implement this in JFFS2 would not be particularly difficult or obtrusive. It is an interesting avenue for future research.

5 Conclusion

Although JFFS2 is extremely young, it is relatively mature, because it is developed from the excellent start given by the design of JFFS v1.

The frequency of bugs being reported has reached a fairly stable low level, and the majority of recent problems reported with JFFS2 have actually turned out to be errors in the physical flash drivers or with other parts of kernel code — although sometimes this has highlighted an area where JFFS2 should be more fault-tolerant.

Both versions of JFFS are now in active use in a reasonable number of embedded systems, and JFFS2 has been included as a fundamental part of the "Familiar" distribution of Linux for the Compaq iPAQ handheld computer; replacing the read-only CRAMFS filesystem which was previously used on those devices.

The existence of a fully-functional writable file system for this class of device is an exciting development, and was absolutely essential to the progress of the Familiar distribution, allowing files to be overwritten individually without having to reset the device and use the bootloader to program a complete replacement CRAMFS.

Commercial support for JFFS2 is available from Red Hat, Inc., for customers wishing to use it in production systems with full backup from the developers.

6 Acknowledgements

The author would like to thank Björn Wesen and the staff of Axis Communications AB for designing the original JFFS and releasing it under the GNU General Public License — and in particular for then answering a stream of silly questions about it.

The author is also grateful to Red Hat, Inc., who for some reason took it upon themselves to actually pay him for playing with this stuff.

Also deserving of a special mention is Vipin Malik, who has done a wonderful job of testing JFFS and JFFS2, often managing to break the latter when it finally seemed to have reached stability.

7 Availability

JFFS was merged into the Linux kernel prior to the 2.4.0 release. The current JFFS2 code is also, at the time of writing, in Alan Cox's 2.4-ac kernels. The latest code for the 2.4 version of each, and for the 2.2 version of JFFS v1, is available from the Linux-MTD CVS repository. Instructions for accessing this, along with links to snapshot tarballs for the firewall-challenged, are available from:

`http://www.linux-mtd.infradead.org/`

The original web site for JFFS and the current code for the 2.0 kernels, along with a link to the `jffs-dev` mailing list which is used for discussion of both JFFS and JFFS2, is at:

`http://developer.axis.com/software/jffs/`

At the time of writing, a web site specific to JFFS2 is intended to appear "shortly" at:

`http://sources.redhat.com/jffs2/`

References

[FTL] Intel Corporation, *Understanding the Flash Translation Layer (FTL) Specification*, (1998). http://developer.intel.com/design/flcomp/applnots/297816.htm

[LFS] Mendel Rosenblum and John K. Ousterhout, *The Design and Implementation of a Log-Structured File System*, ACM Transactions on Computer Systems 10(1) (1992) pp. 26–52. ftp://ftp.cag.lfs.mit.edu/dm/papers/rosenblum:lfs.ps.gz

[eCos] Red Hat, Inc., *eCos — Embedded Configurable Operating System.* http://sources.redhat.com/ecos/