A Brief Introduction to the Design of UBIFS

Document version 0.1
by Adrian Hunter 27.3.2008

A file system developed for flash memory requires **out-of-place updates**. This is because flash memory must be erased before it can be written to, and it can typically only be written once before needing to be erased again. If eraseblocks were small and could be erased quickly, then they could be treated the same as disk sectors, however that is not the case. To read an entire eraseblock, erase it, and write back updated data typically takes 100 times longer than simply writing the updated data to a different eraseblock that has already been erased. In other words, for small updates, in-place updates can take 100 times longer than out-of-place updates.

Out-of-place updating requires garbage collection. As data is updated out-of-place, eraseblocks begin to contain a mixture of valid data and data which has become obsolete because it has been updated some place else. Eventually, the file system will run out of empty eraseblocks, so that every single eraseblock contains a mixture of valid data and obsolete data. In order to write new data somewhere, one of the eraseblocks must be emptied so that it can be erased and reused. The process of identifying an eraseblock with a lot of obsolete data, and moving the valid data to another eraseblock, is called **garbage collection**.

Garbage collection suggests the benefits of **node-structure**. In order to garbage collect an eraseblock, a file system must be able to identify the data that is stored there. This is the opposite of the usual indexing problem facing file systems. A file system usually starts with a file name and has to find the data that belongs to that file. Garbage collection starts with data that may belong to any file (or no file if it is obsolete) and must find which file, if any, it belongs to. One way to solve that problem is to store metadata in line with the file data. This combination of data and metadata is called a **node**. Each node records which file (more specifically inode number) that the node belongs to and what data (for example file offset and data length) is contained in the node. Both JFFS2 and UBIFS follow a node-structured design, that enables their garbage collectors to read eraseblocks directly and determine what data needs to be moved and what can be discarded, and to update their indexes accordingly.

The big difference between JFFS2 and UBIFS is that UBIFS stores the index on flash whereas JFFS2 stores the index only in main memory, rebuilding it when the file system is mounted. Potentially that places a limit on the maximum size of a JFFS2 file system, because the mount time and memory usage grow linearly with the size of the flash. UBIFS was designed specifically to overcome that limitation.

Unfortunately, storing the index on flash is very complex because the index itself must be updated out-of-place. When one part of the index is updated out-of-place, then any other parts of the index that reference the updated part, must also be updated. And then, in turn, the parts that reference those parts must be updated. The solution to this seemingly never-ending cascade of updates is to use a **wandering tree**.

In the case of the UBIFS **wandering tree** (which is technically a B+tree), only the leaves of the tree contain file information. They are valid nodes of the file system. The internal elements of the tree are index nodes and contain only references to their children. That is, an **index node** records the on-flash position of its child nodes. So the UBIFS wandering tree can be viewed as having two parts. A top part consisting of index nodes that create the structure of the tree, and a bottom part consisting of leaf nodes that hold the actual file data. The top part is referred to simply as the **index**. An update to the file system consists of creating a new leaf node and adding it, or replacing it into the wandering tree. In order to do that, the parent index node must also be replaced, and its parent, and so on up to the root of the tree. The number of index nodes that have to be replaced is equal to the height of the tree. There just remains the question of how to know where the root of the tree is. In UBIFS, the position of the root index node is stored in the master node.

The **master node** stores the position of all on-flash structures that are not at fixed logical positions. The master node itself is written repeatedly to **logical eraseblocks** (LEBs) one and two. LEBs are an abstraction created by UBI. UBI maps **physical eraseblocks** (PEBs) to LEBs, so LEB one and two can be anywhere on the flash media (strictly speaking, the UBI device), however UBI always records where they are. Two eraseblocks are used in order to keep two copies of the master node. This is done for the purpose of **recovery**, because there are two situations that can cause a corrupt or missing master node. The first is that there could be a loss of power at the same instant that the master node is being written. The second is that there could be degradation or corruption of the flash media itself. In the first case, recovery is possible because the previous version of the master node can be used. In the second case, recovery is not possible because it cannot be determined reliably what is a valid master node version. In that latter case, a userspace utility program would be needed to analyze all the nodes on the media and attempt to fix or recreate corrupt or missing nodes. Having two copies of the master node makes it possible to determine which situation has arisen, and respond accordingly.

The first LEB is not LEB one, it is LEB zero. LEB zero stores the superblock node. The **superblock node** contains file system parameters that change rarely if at all. For example, the flash geometry (eraseblock size, number of eraseblocks etc) is stored in the superblock node. At present, there is only one situation where the superblock node gets rewritten, which is when an automatic resize occurs. UBIFS presently has a very limited ability to resize to grow bigger, but only to a maximum size specified when the file system is created. This mechanism is needed because the actual size of a flash partition varies due to the presence of a variable number of bad eraseblocks. So when a file system image is created by **mkfs.ubifs**, the maximum number of eraseblocks is specified and the image records this, and the actual number of eraseblocks used, in the superblock node. When UBIFS is mounted on a partition (actually a UBI volume), if the number of eraseblocks is greater than that recorded in the superblock node and less than the maximum number of eraseblocks (also recorded in the superblock node), then the UBIFS file system is automatically resized to fit the partition (UBI volume).

There are in fact six **areas** in UBIFS whose position is fixed at the time the file system is created.

The first two areas have already been described. The superblock area is LEB zero. The superblock node is always at offset zero, and the superblock LEB is written using UBI's atomic LEB change facility which guarantees that the LEB is updated successfully or not at all. The next area is the master node area. It occupies LEB one and LEB two. In general, those two LEBs contain identical data. The master node is written to successive positions in each LEB until there is no more space, at which point the LEBs are unmapped and the master node written at offset zero (which automatically causes UBI to map an erased LEB). Note that the master node LEBs are not both unmapped at the same time because that would leave the file system temporarily with no valid master node. The other UBIFS areas are: the log area (or simply the log), the LEB properties tree (LPT) area, the orphan area and the main area.

The **log** is a part of UBIFS's **journal**. The purpose of the UBIFS journal is to reduce the frequency of updates to the on-flash index. Recall, that the index consists of the top part of the wandering tree that is made up of only index nodes, and that to update the file system a leaf node must be added or replaced in the wandering tree and all the ancestral index nodes updated accordingly. It would be very inefficient if the on-flash index were updated every time a leaf node was written, because many of the same index nodes would be written repeatedly, particularly towards the top of the tree. Instead, UBIFS defines a journal where leaf nodes are written but not immediately added to the on-flash index. Note that the index in memory (see TNC) is updated. Periodically, when the journal is considered reasonably full, it is committed. The **commit process** consists of writing the new version of the index and the corresponding master node.

The existence of the journal means that when UBIFS is mounted, the on-flash index is out-of-date. In order to bring it up-to-date, the leaf nodes in the journal must be read and reindexed. This process is called the **replay** because it replays the journal. Note, that the bigger the journal, the longer it will take to replay, and the longer UBIFS will take to mount. On the other hand, a bigger the journal needs to be committed less often, which make the file system more efficient. The size of the journal is a parameter to the **mkfs.ubifs** program, so it can be chosen to meet the requirements of the system. However, by default UBIFS does not employ its **fast unmount** option, and instead runs a commit before unmounting. That causes the journal to be almost empty when the file system is mounted again, which makes the mount very quick indeed. It is a good trade-off because the commit process itself is generally quite quick, taking a fraction of a second.

Note that the commit process does not move the leaf nodes from the journal. Instead, the journal moves. It is the purpose of the log to record where the journal is. The log contains two types of nodes. A **commit start node**, that records that a commit has begun, and **reference nodes** that record the LEB numbers of main area LEBs that make up the rest of the journal. Those LEBs are called **buds**, so the journal consists of the log and the buds. The log is a finite size and may be considered to be a circular buffer. After a commit, the reference nodes that recorded the previous position of the journal are no longer needed so the tail of the log is erased at the same rate that the head of the log is extended. While the commit-start node records the start of commit, the end of commit is defined to be when the master node is written, because the master node points to the new position of the log tail. If the commit is not completed because the file system is unmounted uncleanly (e.g. power loss), then the replay process replays both the old and new journal.

The **replay process** is complicated by several issues. The first complication is that the leaf nodes must be replayed in order. Because UBIFS employs a **multiheaded journal**, the order that the leaf nodes were written is not simply the order that the corresponding bud eraseblocks were referenced in the log. In order to sequence leaf nodes, every node contains a 64-bit sequence number that increments for the lifetime of the file system. The replay first reads all the leaf nodes in the journal and places them in a RB-tree sorted by sequence number. The RB-tree is then processed in order and the (in-memory) index updated accordingly.

The next complication is that the replay must handle deletions and truncations. There are two kinds of deletion. Inode deletion which corresponds to the deletion of files and directories, and directory entry deletion which corresponds to unlinking and renaming. In UBIFS, inodes have a corresponding **inode node** which records the number of directory entry links, more simply known as the link count. When an inode is deleted, an inode node with a link count of zero is written to the journal. In that case, instead of adding that leaf node to the index, it is removed from the index along with all index entries for nodes with that inode number. In the case of deleted directory entries, a **directory entry node** is written to the journal but the inode number that that directory entry previously referred to, is set to zero. Note that there are two inode numbers associated with a directory entry. The inode number of the parent directory, and the inode number of the file or sub-directory that the directory entry refers to. It is the latter inode number that is set to zero in the deletion directory entry node. When the replay processes a directory entry with an inode number of zero, it removes that entry from the index instead of adding it.

Truncations, of course, change the length of a file. In fact, a truncation can extend the length of a file as well as reduce it. For UBIFS, extending the length of a file requires no special handling. In file system parlance, extending the length of a file via truncation creates a "hole" which is an unwritten part of a file that is assumed to contain all zero bytes. UBIFS does not index holes and does not store any nodes corresponding to holes. Instead a hole is an index entry that is not there. When UBIFS looks up the index and finds no index entry, it is defined to be a hole and the zeroed data created accordingly. On the other hand, truncations that reduce the length of a file require that any data nodes that fall outside the new file length be removed from the index. In order to have that happen, truncation nodes are written to the journal that record the old and new file length. The replay processes those nodes by removing the corresponding index entries.

The next complication is that the replay must bring the **LEB properties tree** (LPT) area up-to-date. **LEB properties** are three values that need to be known for all LEBs in the main area. Those values are: free space, dirty space and whether the eraseblock is an index eraseblock or not. Note that index nodes and non-index nodes are never mixed within the same eraseblock, hence an index eraseblock is an eraseblock that contains (only) index nodes, and a non-index eraseblock is an eraseblock that contains (only) non-index nodes. Free space is the number of bytes at the end of an eraseblock that have not been written to yet, and so can be filled with more nodes. Dirty space is the number of bytes taken up by obsolete nodes and padding, that can potentially be reclaimed by garbage collection. The LEB properties are essential to find space to add to the journal, or the index, and to find the dirtiest eraseblocks to garbage collect. Every time a node is written, the free space must be reduced for that eraseblock. Every time a node is obsoleted or a padding node is written, or a truncation or deletion node is written, dirty space must be increased for that eraseblock. When an eraseblock is allocated to the index, it must be recorded so that, for example, an index eraseblock with free space is not allocated to the journal - which would cause index and non-index nodes to be mixed. Note, the reason that index and non-index nodes may not be mixed has to do with budgeting which is described further on.

Generally speaking, the index subsystem itself takes care of informing the LEB properties subsystem of LEB properties changes. The complexity that LEB properties raises in the replay happens when a garbage collected eraseblock has been added to the journal. Like the index, the LPT area is only updated by the commit. Like the index, the on-flash LPT is out-of-date at mount time and must be brought up to date by the replay process. So the on-flash LEB properties of the garbage collected LEB reflect the state as at the last commit. The replay will begin to update the LEB properties, however some of those changes occurred before the garbage collection, and some of them afterwards. Depending at which point the garbage collection occurred, the final LEB property values will be different. In order to handle that, the replay inserts a reference into its RB-tree to represent the point at which the LEB was added to the journal (using the sequence number of the log reference node). That enables the replay to correctly adjust the LEB property values when the replay RB-tree is applied to the index.

Another complication for the replay is the effect of the recovery on the replay. UBIFS records on the master node whether the file system was unmounted cleanly. If it was not, then certain error conditions trigger the recovery to make fixes to the file system. The replay is affected in two main ways. First, a bud eraseblock may be corrupted because it was being written at the time of a unclean unmount. Secondly, a log eraseblock may be corrupted for precisely the same reason. The replay handles those situations by passing the eraseblock to the recovery to try to fix the nodes in those eraseblocks. If the file system is mounted read-write, then the recovery will do the necessary fixes on the flash. In that way, the integrity of a recovered UBIFS file system is as perfect as one that has not suffered an unclean unmount. If the file system is mounted read-only, the recovery is deferred until it is mounted read-write.

The final complication is that some of the leaf nodes referenced by the on-flash index may not exist anymore. That happens when the nodes have been deleted and the eraseblock that contained them has subsequently been garbage collected. Generally, deleted leaf nodes do not affect the replay because they are not part of the index. However, one aspect of the structure of the index does require that leaf nodes are sometimes read in order to update the index. That happens because of directory entry nodes (and also extended attribute entry nodes). In UBIFS, a directory consists of an inode node and one directory entry node for each of the directory entries. Access to the index is done using a node key, which is a 64-bit value that identifies the node. In most cases, the node key uniquely identifies the node and so the index can be updated using just the key. Unfortunately, in the case of directory entries (and extended attribute entries) the uniquely identifying information is the name, which may be quite long (up to 255 characters in the case of UBIFS). To squeeze that information into 64-bits, the name is hashed into a 29-bit value which is not unique to the name. When two names give the same hash value, it is called a **hash collision** (or simply collision). In that case, the leaf nodes must be read to resolve the collision by comparing the names stored in the leaf nodes. So what happens if the leaf node is gone because of the reasons given above. It turns out that it does not matter. Directory entry nodes (and extended attribute entry nodes) are only ever added or removed - they are never replaced because the information they contain never changes. So the outcome of the name comparison is known even though the node contained one of the names is gone. When adding a hashed-key node, there will be no match. When removing a hashed-key node, there will always be a match, either to an existing node, or to a missing node that has the correct key. In order to provide this special index updating for the replay, a separate set of functions is used (denoted in the code by the prefix "fallible").

After the log area, comes the **LPT area**. The size of the log area is defined when the file system is created and consequently so is the start of the LPT area. At present, the size of the LPT area is automatically calculated based on the LEB size and maximum LEB count specified when the file system is created. Like the log area, the LPT area must never run out of space. Unlike the log area, updates to the LPT area are not sequential in nature - they are random. In addition, the amount of LEB properties data is potentially quite large and access to it must be scalable. The solution is to store LEB properties in a wandering tree. In fact the LPT area is much like a miniature file system in its own right. It has its own LEB properties - that is, the LEB properties of the LEB properties area (called ltab). It has its own form of garbage collection. It has its own node structure that packs the nodes as tightly as possible into bit-fields. However, like the index, the LPT area is updated only during commit. Thus the on-flash index and the on-flash LPT represent what the file system looked like as at the last commit. The difference between that and the actual state of the file system, is represented by the nodes in the journal.

The LPT actually has two slightly different forms called the **small model** and the **big model**. The small model is used when the entire LEB properties table can be written into a single eraseblock. In that case, LPT garbage collection consists of just writing the whole table, which therefore makes all other LPT area eraseblocks reusable. In the case of the big model, dirty LPT eraseblocks are selected for LPT garbage collection, which consists of marking the nodes in that LEB as dirty, and then writing out only the dirty nodes (as part of the commit). Also, in the case of the big model, a table of LEB numbers is saved so that the entire LPT does not to be scanned looking for empty eraseblocks when UBIFS is first mounted. In the small model, it is assumed that scanning the entire table is not slow because it is small.

One of the main tasks of UBIFS is to access the index which is a wandering tree. To make that efficient, index nodes are cached in memory in a structure called the **tree node cache** (TNC). The TNC is B+tree that is node-for-node the same as the on-flash index, with the addition of all changes made since the last commit. The nodes of the TNC are called **znodes**. Another way to look at that, is that a znode when it is on-flash is called an index node, and an index node when it is in memory is called a znode. Initially there are no znodes. When a lookup is done on the index, just the index nodes that are needed are read and added to the TNC as znodes. When a znode needs to be changed, it is marked as dirty which pins it in memory until the next commit at which time it becomes clean again. At any time UBIFS memory shrinker may decide to free clean znodes in the TNC, so that the amount of memory needed is proportional to the size of the parts of the index that are in use, not the total size of the index. In addition, hanging off the bottom of the TNC is a **leaf-node cache** (LNC) which is used only for directory entries (and extended attribute entries). The LNC is needed to cache nodes read as a result of collision resolution or readdir operations. Because the LNC is attached to the TNC it effectively gets shrunk when the TNC does.

The TNC is further complicated by the desire to make the commit have as little impact as possible on other UBIFS operations. To do that, the commit is split into two main parts. The first part is called **commit start**. During commit start the commit semaphore is down for writing, which prevents all further updates to the journal. At that time, the TNC subsystem makes a list of the dirty znodes and lays out the positions on flash where they will be written. Then the commit semaphore is released, and a new journal begins to be used, while the commit is still running. The second part of the commit is called **commit end**. During commit end, TNC writes the new index nodes but it does so without any lock on TNC. That is TNC can be updated simultaneously as the new index is being written to flash. That is accomplished by marking the znodes which are being committed as **copy-on-write**. If a znode that is being committed needs to be changed, it is copied so that the commit still sees the unchanged znode. In addition, the commit is mostly run by UBIFS background thread so that user processes wait as little as possible for the commit to run.

Note that LPT follows the same commit strategy as TNC and they are both wandering trees implemented as B+trees, resulting in a number of similarities in the code of LPT and TNC.

There are three important differences between UBIFS and JFFS2. The first has already been mentioned: UBIFS has an on-flash index, JFFS2 does not - thus UBIFS is potentially scalable. The second difference is implied: UBIFS runs on top of the UBI layer which runs on top of the MTD subsystem, whereas JFFS2 runs directly over MTD. UBIFS benefits from the wear-leveling and error handling of UBI at the cost of the flash space, memory and other resources taken by UBI. The third important difference is that UBIFS allows **writeback**.

Writeback is a VFS facility that allows written data to be cached and not written immediately to the media. That makes the system more responsive and potentially more efficient because updates to the same file can be grouped together. The difficulty with supporting writeback is that it requires that the file system know how much free space is available so that the cache is never bigger than the space on the media. That is very difficult for UBIFS to determine, so an entire subsystem called **budgeting** is dedicated to it. The difficulties are for several reasons.

The first reason is that UBIFS supports **transparent compression**. Because the amount of compression is not known in advance, the amount of space needed is not known in advance. Budgeting must assume the worst case, and assume that there is no compression. However, in many cases that is a poor assumption. In order to overcome that, budgeting will begin to force writeback when it detects that there is insufficient space.

The second reason that budgeting is difficult is that garbage collection is not guaranteed to be able to reclaim all dirty space. UBIFS garbage collection processes one eraseblock at a time. In the case of NAND flash, only complete NAND pages can be written at a time. A NAND eraseblock is made up of some fixed whole number of NAND pages. UBIFS calls the NAND page size the minimal I/O unit. Because UBIFS garbage collection processes one eraseblock at a time, if the dirty space is less than the minimal I/O size, it cannot be reclaimed - it will end up as padding in the last NAND page. When the dirty space in an eraseblock is less than the minimal I/O size, that space is called **dead space**. Dead space is not reclaimable.

Similar to dead space is **dark space**. Dark space is when the dirty space in an eraseblock is less that the maximum node size. In the worst case, the file system may be full of nodes of maximum size and garbage collection will not result in pieces of free space that are large enough for another maximum size node. So, in the worst case, dark space is not reclaimable, and in the best case it is reclaimable. UBIFS budgeting must assume the worst case and so both dead space and dark space and assumed to be unavailable. However, if there is insufficient space but a lot of dark space, budgeting will itself run garbage collection to see if it reclaims more free space.

The third reason that budgeting is difficult is that cached data may be obsoleting data stored on flash. Whether or not that is the case is not always known, and what the difference in compression may be is certainly not known. This is another reason that budgeting forces writeback when it calculates insufficient space. Only after trying writeback, garbage collection and committing the journal, will budgeting give up and return ENOSPC (the no space error code).

Of course that means that UBIFS becomes less efficient as the file system becomes closer to being full. In fact, all flash file systems become less efficient as the flash fills up. That is because it is less likely that there is an empty eraseblock that has been erased in the background, and more likely that garbage collection will have to run.

The fourth reason that budgeting is difficult is that deletions and truncations need to write new nodes. Thus if the file system is really out of space, it becomes impossible to delete anything because there is no room to write a deletion inode node, or a truncation node. To prevent that situation, UBIFS always keeps back a some space and always allows deletions and truncations.

The next UBIFS area to describe is the **orphan area**. An orphan is an inode number whose inode node has been committed to the index with a link count of zero. That happens when an open file is deleted (unlinked) and then a commit is run. In the normal course of events the inode would be deleted when the file is closed. However in the case of an unclean unmount, orphans need to be accounted for. After an unclean unmount, the orphans' inodes must be deleted which means either scanning the entire index looking for them, or keeping a list on flash somewhere. UBIFS implements the latter approach.

The orphan area is a fixed number of LEBs situated between the LPT area and the main area. The number of orphan area LEBs is specified when the file system is created. The minimum number is 1. The size of the orphan area should be so that it can hold the maximum number of orphans that are expected to ever exist at one time. The number of orphans that can fit in a LEB is:

(leb_size - 32) / 8

For example: a 15872 byte LEB can fit 1980 orphans so 1 LEB may be enough.

Orphans are accumulated in a RB-tree. When an inode's link count drops to zero, the inode number is added to the RB-tree. It is removed from the tree when the inode is deleted. Any new orphans that are in the orphan tree when the commit is run, are written to the orphan area in 1 or more orphan nodes. If the orphan area is full, it is consolidated to make space. There is always enough space because validation prevents the user from creating more than the maximum number of orphans allowed.

The final UBIFS area is the **main area**. The main area contains the nodes that make up the file system data and the index. A main area LEB may be an index eraseblock or a non-index eraseblock. A non-index eraseblock may be a bud (part of the journal) or have been committed. A bud may be currently one of the journal heads. A LEB that contains committed nodes can still become a bud if it has free space. Thus a bud LEB has an offset from which journal nodes begin, although that offset is usually zero.

UBIFS Source File List

budget.c	budgeting
build.c	module initialization, mounting and unmounting
commit.c	commit management
compress.c	compression management
debug.c	debugging self-checks
debug.h	debugging self-checks header
dir.c	directory operations
file.c	file operations
find.c	find eraseblocks for various purposes e.g. G.C.
gc.c	garbage collection
io.c	node I/O
ioctl.c	EXT2 compatible extended attribute ioctls
journal.c	journal updates
key.h	index key related functions
log.c	log
lprops.c	LEB properties
lpt.c	LEB properties tree (LPT)
lpt_commit.c	LPT commit
master.c	master node
misc.h	shared inline functions
orphan.c	orphans
recovery.c	recovery
replay.c	replay
sb.c	default file system creation and super block node
scan.c	general purpose eraseblock scanner (e.g. used by G.C.)
shrinker.c	memory shrinker for TNC
super.c	super operations
tnc.c	tree node cache (TNC)
tnc_commit.c	TNC commit
ubifs.h	internal definitions
ubifs-media.h	on-flash structures and definitions
xattr.c	extended attributes

UBIFS Glossary

B+tree Kind of tree used for UBIFS wandering tree.

base head Journal head used for non-data nodes. The base head provides the integrity and recoverability of UBIFS by keeping all inode nodes and directory entry nodes in sequence.

budgeting Estimation of free space.

bud An eraseblock used by the journal.

cnode Either a pnode or an nnode.

commit Process by which the index is updated.

commit start node Node written to the log when a commit starts.

common header Common header of all node types except nodes in the LPT area.

data head Journal head used for data nodes. Potentially UBIFS could have more than one data head which would assist in keeping data nodes for each file grouped together rather than interspersed. However presently there is only one data head.

directory entry node A node that contains a directory entry.

dirty space Space taken up on flash by obsolete nodes, padding nodes and padding bytes.

eraseblock The smallest unit of flash memory that can be individually erased.

extended attribute entry node A node that contains the name and inode number of an extended attribute (same structure as a directory entry).

freeable LEB A LEB in the main area that contains only free space and dirty space.

free space Space on flash that can be written to.

garbage collection The process of reclaiming dirty space to make free space.

garbage collection head Journal head used by the garbage collector. In order to move and reindex nodes, the garbage collector just feeds them back through the journal.

garbage collection LEB number LEB reserved for garbage collection (must always be one LEB reserved), also known as gc_lnum.

index The top part of the UBIFS wandering tree consisting of index nodes.

index LEB A LEB that contains (only) index nodes.

index node A node that forms the structure of the index. It contains the keys and on-flash positions of its child nodes.

inode A file system object. In UBIFS, an inode may be a regular file, a directory, a symbolic link, a special file, or an extended attribute (xattr) value holder.

inode node A node that holds the metadata for an inode. Every inode has exactly one (non-obsolete) inode node.

in-the-gaps method When the file system becomes full, the index cannot grow in size and consequently must be updated in-place. This is done by writing index nodes in-the-gaps created by obsolete index nodes.

JFFS2 Journaling Flash File System 2 which is the quintessential linux flash file system, however it is not scalable and the present limit of its usage is 1 GiB.

journal The journal is like a miniature JFFS2 file system, storing nodes without an on-flash index, so that index updates can be grouped altogether. Like JFFS2 the journal must be scanned at mount time.

journal head The position on-flash where the next node will be written. UBIFS adopts a multiheaded journal with two main heads: the base head and the data head.

key The key to the index. The index stores the position of all nodes, and their keys. A node may be found by looking up its key in the index. The key is a 64-bit value whose first 32 bits are the inode number. The next 3 bits are node type, and the remaining 29 bits vary depending on node type. UBIFS also supports the possibility of changing the key format and allows for up to 128-bits for the key.

LEB Logical eraseblock that is mapped to a physical eraseblock (PEB) by UBI.

LEB properties Main area LEBs have three important values that are recorded about them. These LEB properties are: amount of free space, amount of dirty space, and whether the LEB contains index nodes.

LNC Leaf node cache used to store directory entries and extended attribute entries

log The part of the journal that records where the buds are.

LPT LEB properties tree, a wandering tree used to store LEB properties.

main area LEBs that are used for data and the index.

master node The node that contains the positions of on-flash structures.

minimal I/O unit The smallest unit of flash memory that can be individually written.

mkfs.ubifs The userspace program that creates UBIFS images.

MTD Memory Technology Device. A linux device layer for flash memories.

node The logical component of node-structured flash file systems like JFFS2 and UBIFS.

nnode Internal node of the LPT.

orphan area Area for storing the inode numbers of deleted by still open inodes, needed for recovery from unclean unmounts.

orphan node The node used to store orphan information.

padding bytes Bytes written when the space to pad is too small for a padding node.

padding node A node written to fill space, usually to the edge of a minimal I/O unit.

PEB Physical eraseblock that is mapped to a logical eraseblock (LEB) by UBI.

pnode Leaf node of the LPT which contains the actual LEB property values.

recovery The functions by which UBIFS repairs itself after an unclean unmount.

replay The process by which UBIFS reindexes the nodes in the journal when the file system is mounted.

reference node Log nodes that store the LEB numbers of buds and their offset.

shrinker A linux facility the requests that some memory be freed if possible. UBIFS has a shrinker that frees clean znodes from the TNC.

superblock node The node that records file system parameters that almost never change.

TNC Tree Node Cache, UBIFS index in memory.

truncation node A node that records when a file is truncated and becomes smaller.

UBI Unsorted Block Images module that provides wear-leveling, bad-block handling, and the creation of logical volumes mapping logical eraseblocks to physical eraseblocks.

UBIFS UBI File System.

unclean unmount When the file system does not have a chance to write all cached data and metadata, for example because of a loss of power.

wandering tree A tree that moves in a way to allow out-of-place updates.

znode An index node in memory. Znodes form the TNC.