Edith Cowan University

CSG1102
Operating Systems

Assignment 2

Implementation and operation of the ext3 filesystem

Martin Ponce
Student 10371381

Tutor: Peter Hannay

September 30, 2015

Contents

1 Executive summary

This report provides a description of the ext3 filesystem.

2 Introduction to journaled filesystems

One of the most important parts of an operating system is the filesystem. It contains and manages critical data on disk drives, such as user configuration, user data and applications and of course, the operating system itself. The filesystem ensures that what is read from storage is identical to what was originally written. In other words, it ensures data integrity. A filesystem achieves data integrity by managing and storing not only the actual data, but also information about the data, or files in storage. It also stores and manages information about the filesystem itself. This information is known as metadata. User's trust the storage of data to the reliability and performance of the filesystem (Best, 2002).

But what happens to data when a system is improperly shutdown during file operations due to a crash or abrupt power loss? When Linux detects an improper shutdown, it signals a non-journaled filesystem (such as ext2) to perform a consistency check with `fsck` before booting into the operating system. This function scans the entire filesystem and attempts to fix any detected consistency issues that can be safely resolved. The `fsck` utility may take a considerable amount of time, depending on size of the filesystem, which can equate to serious downtime for the user (Jones, 2008). According to the developer of the ext3 filesystem, Tweedie (2000), Linux filesystems are growing in size, causing longer `fsck` process times.

Journaled filesystems negate the need for `fsck` (or equivalent in non *nix operating systems) to verify filesystem integrity, therefore greatly reducing downtime and increasing system availability in the event of an improper shutdowns (Best, 2002; Bost, 2012; Jones, 2008; Prabhakaran, Arpaci-Dusseau, & Arpaci-Dusseau, 2005; Tweedie, 1998, 2000). This is achieved by implementing a *log* or *journal* that records changes destined for the filesystem. The changes recorded in the journal are committed to the filesystem periodically, and when an improper shutdown occurs, the journal is referenced as a checkpoint to recover unsaved information to avoid corrupted filesystem metadata (Jones, 2008). Therefore, a `fsck` scan of the entire filesystem is no longer required to determine consistency after an improper shutdown, as the journal is used for reference to determine integrity.

3 Roots in ext2

In addition to creating a journaled filesystem for Linux, Tweedie (2000) describes one of the main goals in the design of the ext3 filesystem is the compatibility with the existing ext2 filesystem. This goal was justified, as much of the Linux user base was using ext2 at the time, and greater compatibility between the two filesystems would ensure seamless transition for users. This goal was successfully achieved, and an ext3 filesystem that is cleanly unmounted can easily be remounted as an ext2 filesystem. Conversely, an ext2 filesystem can easily be upgraded to ext3 *in-place*, while the volume is mounted (Bovet & Cesati, 2006; Robbins, 2001a; Tweedie, 2000).

This backwards and forwards compatibility was achieved by using the same source code base for ext3 as ext2. Therefore, ext2 and ext3 share many common properties. For example, both filesystems use the same on-disk and metadata format. Ext3 also inherits ext2's `fsck` (Robbins, 2001a; Tweedie, 2000). Although the desired effect of ext3 is to avoid the use of `fsck`, there are still times when it may be required, in the event that metadata is corrupted. Availability of `fsck`, with years of development further enhances the reliability of ext3, when compared to other journaled filesystems which do not have a similar capability (Robbins, 2001a).

4 Ext3 journal mechanics

Tweedie (1998, p. 5) explains that a journal has two concurrent responsibilities: "record the new content of filesystem metadata blocks while [it is] in the process of committing transactions". This sentiment is also echoed by Galli (2001, p. 55). With backwards/fowards compatibility in mind, journaling functionality has been added to ext2 to create the ext3 filesystem by implementing two main concepts, the Journaling Block Device, and transactions, providing functionality to satisfy the two responsibilities of the journal.

The ext3 journal itself is stored in an inode using a circular buffer data structure (Jones, 2008; Prabhakaran et al., 2005; Robbins, 2001a; Tweedie, 2000). An inode is an area on the disk where all the information about a file is stored, however, the actual file itself is not stored in an inode. Best (2002, p. 2) uses the analogy of a "bookkeeping file for a file", and identifies that an inode is in fact a file itself. The decision to store the journal in an inode contributes to the goal of backwards/forwards compatibility with ext2, and avoids the use of incompatible extensions to the ext2 metadata (Robbins, 2001a). It is common for the journal to be stored within the filesystem itself, however it is also possible to store the journal on a separate device or partition, and even have multiple filesystems sharing the same journal (Galli, 2001; Prabhakaran et al., 2005; Tweedie, 2000).

Additionally, the circular buffer could be described as a temporary holding area for the journal. Space is cleared and reclaimed in the buffer for the journal, once data and metadata are placed into their fixed locations on disk (Prabhakaran et al., 2005). The sections to follow describe the various implemented mechanics which make it possible for ext3 to function as a journaled filesystem.

4.1 Journaling Block Device (JBD)

In order to enable journaling in ext3, an API external to the filesystem was developed, called the *Journaling Block Device* (JBD). Its main purpose is to implement "a journal on any kind of block device", acting as an external layer to provide ext3 with journaling capabilities (Mauerer, 2008; Robbins, 2001a, p. 8). Tweedie (2000) asserts that the JBD is completely encapsulated from ext3. It does not know anything about how the filesystem works, and conversely, the filesystem does not know anything about journaling. The extent of ext3's knowledge of journaling is through *transactions*.

The JBD API allows the filesystem to communicate the modifications to be performed to the JBD as transactions, which in turn is recorded to the journal by the JBD. In order for the JBD to have an opportunity to manage the journal on the ext3 filesystem driver's behalf, ext3 requests permissions from the JBD before modifying certain data on the disk (Robbins, 2001a).

4.2 Journal approach

Robbins (2001a, p. 3) describes two methods of journaling, *logical journaling* and *physical journaling*. Logical journaling refers to the method where the journal stores "spans of bytes that need to be modified on the host filesystem" and is found to be used in other journaled filesystems such as XFS. In other words, this approach only records individual bytes that require modification, and is efficient at storing many, smaller modifications to a filesystem (Robbins, 2001a).

On the other hand, physical journaling is where entire blocks of modified filesystem are recorded in the journal, and is the journaling approach used by ext3. This means that when a small change is made, the entire journal block must be written (Bovet & Cesati, 2006; Robbins, 2001a). Although the journal will require more space, it requires less CPU overhead compared to logical journaling, since there is less complexity transferring a literal block from memory to disk (Robbins, 2001a), and "journal operations can be batched into large clusters" (Galli, 2001, p. 55).

4.3 Journal hierarchy

The contents of the journal form a hierarchy of information, starting with log records which record the lowest level of filesystem operation. Log records are then grouped by handles, which represent operations at high level system calls. Finally, transactions group handles together and represent a single, atomic update to the filesystem.

4.3.1 Log records

A log record is an individual low level operation to update a block and is the smallest unit of data that can be logged in the journal (Mauerer, 2008). As discussed previously, ext3's journal approach is physical journaling, where entire blocks of modified filesystem are recorded in the log. Therefore, an ext3 log record consists of "the whole buffer modified by the low level operation" (Bovet & Cesati, 2006, p. 770). Each log record contains a `journal_block_tag_t`, storing the associated logical block number within the filesystem as well as status flags (Bovet & Cesati, 2006).

4.3.2 Handles

A handle is a group of low level log records that represent a high level atomic operation at the system level. Mauerer (2008, p. 639) explains that if a `write` system call is invoked, any log records associated with that particular call would be grouped together to form a handle. Therefore, each system call must be dealt with in an atomic manner to prevent data corruption, and is assisted by recording handles in the journal. In other words, similar to transactions, a handle must perform the entire sequence of log record changes to disk, or none at all. Each handle is "represented by a descriptor of type `handle_t` (Bovet & Cesati, 2006, p. 770).

4.3.3 Transactions

Transactions are the main concept in journaled filesystems, which "corresponds to a single update of the filesystem" (Tweedie, 1998, p. 4). Transactions are formed by groups of handles (Bovet & Cesati, 2006; Mauerer, 2008), and log records that related to a particular handle must reside in a single transaction. This concept has been implemented into ext2 to create the ext3 filesystem and provides a format for the filesystem to communicate with the JBD API (Tweedie, 2000), and as a means of efficiency for the filesystem (Bovet & Cesati, 2006).

Similar to database transactions, a journaled filesystem groups together "collections of atomic sequence of events" and is executed as a single operation, rather than taking each individual filesystem into account (Katiyar, 2011a, p. 2). In other words, the entire sequence of a transaction must be completed, or none at all. As Tweedie (2000, p. 4) explains, "exactly one transaction results from any single filesystem request made by an application, and contains all of the changed metadata resulting from that request".

Each transaction contains a descriptor `transaction_t`, which in turn contains field `t_state`, describing the status of the transaction (Bovet & Cesati, 2006). The state of a transaction determines whether or not it is considered during recovery after an improper shutdown. Bovet and Cesati (2006, p. 771) describes each of the possible states of a transaction:

Complete A transaction is marked complete when all associated log records have been physically written to the journal and are considered to assist in recovery after an improper shutdown. The `t_state` field contains value `T_FINISHED` when a transaction is completed.

Incomplete A transaction is considered incomplete when not all associated log records have been physically written to the journal, or subsequent log records are still being added to the transaction. Incomplete transactions are discarded during recovery after an improper shutdown. Incomplete transactions have four possible values stored in the `t_state` field:

Table 1: Incomplete `t_state` values (Bovet & Cesati, 2006, p. 771-772).

Value	Desciption
`T_RUNNING`	Transaction is still accepting new handles
`T_LOCKED`	Transaction no longer accepting new handles, but some are unfinished
`T_FLUSH`	Transaction has recorded all handles, but is still in the process of recording remaining log records
`T_COMMIT`	Transaction has finished writing all associated log records of handles to disk, but transaction is yet to be marked completed on the journal

The journal may contain multiple transactions, but only one transaction may be active at any given time, with state `T_RUNNING`. The other transactions contained in the journal may be incomplete, waiting for buffers to write the associated log records to the journal (Bovet & Cesati, 2006).

A transaction is marked complete when all associated log records have been written to the journal, but updates to the fixed location on disk may still not have occurred. A complete transaction is only removed from the journal after verification that a complete transaction has been written to the fixed location on disk (Bovet & Cesati, 2006). This process is handled through checkpointing, discussed in a later section.

4.4 Journal structure

As each ext3 transaction is recorded into the journal, they are organized and described by the following metadata blocks: *journal superblock*, *descriptor block*, and *journal commit block* (Prabhakaran et al., 2005). Summary information such as block size and head/tail pointers are recorded by the journal superblock. As discussed previously, ext3 logs an entire block to journal if it has even the smallest update required, and summary information about these blocks are stored in the journal superblock (Galli, 2001; Prabhakaran et al., 2005)

The start of a transaction is marked by the descriptor block, containing information about the subsequent journaled blocks as well as their fixed locations on disk (Prabhakaran et al., 2005). The role of descriptor blocks is to describe other metadata journal blocks, supporting journaled recovery functions (Galli, 2001; Tweedie, 1998). What is stored after the descriptor block in a journal depends on the selected ext3 mode of operation, as described in more detail below. In data journaling mode, data and metadata blocks are stored after the descriptor block. In writeback and ordered modes, metadata blocks are stored after the descriptor block (Prabhakaran et al., 2005).

The journal commit block signals the end of a transaction, and is stored after the metadata and data blocks marked for update in that particular transaction. "Once the commit block is written, the journaled data can be recovered without loss" (Prabhakaran et al., 2005, p. 109).

4.5 Ext3 journal modes

Ext3 offers three modes of journal operation, which can be selected during the mounting of the filesystem. These modes are *writeback mode*, *ordered mode*, and *data journaling mode* (Jones, 2008; Mauerer, 2008; Prabhakaran et al., 2005). Each mode offers varying levels of data consistency guarantees and are interchangeable.

4.5.1 Writeback mode

In writeback mode, only metadata is journaled and data blocks are written directly to disk. While this mode preserves the filesystem structure and guarantees metadata consistency, it provides the "weakest consistency semantics of the three modes" (Prabhakaran et al., 2005, p. 108). In other words, it is still possible for data to be corrupted, because the order between journal and fixed location data writes are not enforced. For example, if a system crash occurs after metadata has been journaled, but before the data block is written, it is likely the data may contain garbage or previously written data (Jones, 2008; Prabhakaran et al., 2005). However, this mode provides "the best ext3 performance under most conditions" (Robbins, 2001b, p. 2).

4.5.2 Ordered mode

Similar to writeback mode, ordered mode only journals metadata. However, order between journal and fixed location data writes are enforced. This is the default mode, if a user does not select one during the mounting of the ext3 filesystem. Metadata and data block writes are grouped logically as transactions, as mentioned previously. When the time comes to *commit* the transaction (write metadata) to disk, data blocks must be written first before metadata is journaled (Robbins, 2001b). This ordering of writes effectively guarantees both metadata and data recovery consistency (Jones, 2008; Prabhakaran et al., 2005).

4.5.3 Data journal mode

Data journal mode offers the guarantee of data and metadata consistency, due to the journaling of both metadata and data. However, there are performance trade-offs with this mode since data is being written twice: once to the journal, and again to the fixed ex2 location. Data journal mode is generally considered the slowest of all ext3 journaling modes, however Robbins (2001b, p. 3) references an experiment which shows it can perform well where "interactive performance IO needs to be maximized".

4.6 Checkpointing

After a transaction has been committed, the data at the corresponding fixed location on disk may still be waiting to be updated. Therefore, the corresponding data for a particular transaction must remain in the journal until associated updates to fixed locations on disk have been made (Tweedie, 1998).

A task handled by the JBD, the checkpointing process verifies each completed transaction that all the related data transfers relating to that particular transaction have occurred. Once a transaction in complete status has been verified by checkpointing, the space taken up by the checkpointed compound transaction on the journal can be reclaimed for future transactions (Bovet & Cesati, 2006; Katiyar, 2011b; Prabhakaran et al., 2005; Tweedie, 1998, 2000). Various events can trigger the checkpointing process, such as low space in either the filesystem buffer or journal itself, or when a time-out expires, ensuring that it occurs periodically (Prabhakaran et al., 2005).

5 Ext3 operations

5.1 Writing data block of regular file in ext3

Bovet and Cesati (2006, p. 772-774) describe the processes involved when the ext3 filesystem is requested to write data blocks of a regular file in stepwise fashion:

1. A call to system function `write()` triggers `generic_file_write()` for the file object associated with the ext3 file
2. `ext3_prepare_write()` is called for the `address_space` object for each page of data relating to the write operation
3. A new atomic operation is initiated from `ext3_prepare_write()` by calling the JBD API function `journal_start()`, creating a new handle
    - First call to `journal_start()` will result in newly created handle to be added to active transaction
    - Subsequent calls to `journal_start()` for the same handle verifies that the `journal_info` field of the process descriptor is set and uses the referenced handle
4. `ext3_prepare_write()` calls `block_prepare_write()`, passing the address of `ext3_get_block()` function
    - `block_prepare_write()` prepares the buffers and buffer heads of the file's page
5. Kernel calls `ext3_get_block()` to determine the logical number of a block from the ext3 filesystem
    - `journal_get_write_access()` is called before issuing a low level write operation on a metadata block of the filesystem, to add the metadata buffer to a list of the active transaction
        - If the metadata is already included in an existing and incomplete transaction, the buffer is duplicated, ensuring that older transactions are committed with the old content
    - `journal_dirty_metadata()` is called after updating the buffer containing the metadata block, moving the metadata buffer to the dirty list of the active transaction and logging the operation in the journal
6. If using data journal mode, `ext3_prepare_write()` calls `journal_get_write_access()` on every buffer affected by the write operation, recalling that in data journal mode, metadata as well as data is recorded to the journal

7. `generic_file_write()` receives control again, updating the page page with the data stored in the User Mode address space, then calls `commit_write()` for the `address_space` object, depending on the ext3 mode:
    - In data journal mode, `commit_write()` is implemented by `ext3_journalled_commit_write()`, calling `journal_dirty_metadata()` on every buffer of data in the page
        - This ensures that data is also included in the proper dirty list of the active transaction and not in the dirty list of the owner inode
        - Also ensures that the corresponding log records are written to the journal
        - Again, recall that data journal mode writes both data and metadata to the journal
        - `ext3_journalled_commit_write()` then calls `journal_stop()` so that the JBD closes the handle for that particular operation
    - In ordered mode, `commit_write()` is implemented by `ext3_ordered_commit_write()`, calling `journal_dirty_data()` on every buffer of data in the page
        - This inserts the buffer in a proper list of the active transactions
        - The JBD then ensures that all data buffers in this list are written to disk before the metadata buffers of the transaction
        - The JBD does not include log records to the journal at this stage in ordered mode mode
        - `ext3_ordered_commit_write()` then calls `generic_commit_write()`, to insert the data buffers in the list of the dirty buffers of the owner inode
        - `ext3_ordered_commit_write()` then calls `journal_stop()` so that the JBD closes the handle for that particular operation
    - In writeback mode, `commit_write()` is implemented by `ext3_writeback_commit_write()`, calling `generic_commit_write()`
        - This inserts the data buffers in the list of the dirty buffers of the owner inode
        - `ext3_writeback_commit_write()` then calls `journal_stop()` so that the JBD closes the handle for that particular operation

8. `write()` system call finishes here, but the JBD continues processing
    - When all log records have been physically written to the journal, the transaction status will be changed to complete
    - When complete, `journal_commit_transaction()` is called

9. While in ordered mode, `journal_commit_transaction()` ensures I/O data transfers for all data buffers included in the list of the transaction first, and waits until all data transfers are completed

10. While in writeback or data journal mode, `journal_commit_transaction()` initiates I/O data transfers for associated metadata buffers for the transaction

- Additionally for data journal mode, I/O data transfers are also initiated for data buffers

11. The kernel periodically initiates checkpointing of all complete transactions residing in the journal, verifying that I/O transfers have successfully completed

    - If so, the transaction is removed from the journal to make room for new transactions

5.2 Recovery in ext3

Until a system failure occurs, the contents of the journal are not used (Bovet & Cesati, 2006). However, when the filesystem is uncleanly unmounted during a disk operation, the next mount of the filesystem will highlight transactions in the journal that have yet to be checkpointed to their fixed locations on disk (Katiyar, 2011a). In such cases, `e2fsck` is triggered, scanning the journal and replaying completed transactions required to make the filesystem consistent again (Bovet & Cesati, 2006). Katiyar (2011a, p. 8-9) describes the recovery process in stepwise fashion:

1. `journal_recover()` called
    - Readahead journal blocks in memory
2. `do_one_pass(PASS_SCAN)` called
    - This is the first scan of the journal, determines if recovery is required
    - If recovery is required, sanity checks are performed to determine which transactions need to be replayed and confirms if the journal is valid
        - Only completed transactions are considered, incomplete transactions are discarded (Bovet & Cesati, 2006)
        - Data structure `recovery_info` is populated with information required for the recovery
            * Such as `s_start`, defining which block number to begin recovery
    - If recovery is not required, `s_start = 0`
3. `do_one_pass(PASS_REVOKE)` called
    - Builds a list of revoked blocks which is referred to during recovery process, ensuring that old data in a block is not overwritten by new data which would cause corruption
4. `do_one_pass(PASS_REPLAY)` called
    - Replays transactions from the journal and copies data from the journal to fixed locations on disk
        - Read the corresponding block number from the filesystem
        - Copy contents of journal to buffer
        - Mark the buffer dirty
        - Buffer written to fixed location on disk

5. `journal_clear_revoke()` called
   - List of revoked blocks destroyed once replay is completed, space can be reclaimed
6. `sync_blockdev()` called
   - Sync block device
7. `journal_reset()` called
   - Resets in-memory fields of the journal, allowing it to operate as normal again when the filesystem is remounted

6 Conclusion

References

Best, S. (2002). Journaling File Systems. *Linux Magazine*(October).

Bost, T. (2012). *Linux for Windows systems administrators: Managing and monitoring the extended file system.* Retrieved 2015-09-14, from http://www.ibm.com/developerworks/linux/library/l-filesystem-management/index.html

Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux Kernel* (3rd ed.). Sebastopol, CA: O'Reilly.

Galli, R. (2001). Journal File Systems in Linux. *Upgrade*, *2*(6), 50–56.

Jones, T. M. (2008). *Anatomy of linux journaling file systems.* Retrieved 2015-09-14, from http://www.ibm.com/developerworks/linux/library/l-journaling-filesystems/index.html

Katiyar, M. (2011a). *Journalling Block Device (JBD).* Retrieved 2015-09-21, from http://www.linuxforums.org/articles/journalling-block-device-jbd-_1544.html

Katiyar, M. (2011b). *Journalling layer in ext3 (jbd).* Retrieved 2015-09-21, from http://mkatiyar.blogspot.com.au/2011/07/journalling-layer-in-ext3-jbd.html

Mauerer, W. (2008). *Professional Linux Kernel Architecture.* Indianapolis, IN: Wiley.

Prabhakaran, V., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2005). Analysis and Evolution of Journaling File Systems. In *Usenix annual technical conference* (pp. 105–120). Retrieved from https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/general/full_papers/prabhakaran/prabhakaran_html/main.html

Robbins, D. (2001a). *Common threads: Advanced filesystem implementor's guide, Part 7.* Retrieved 2015-09-14, from http://www.ibm.com/developerworks/library/l-fs7/

Robbins, D. (2001b). *Common threads : Advanced filesystem implementor's guide, Part 8.* Retrieved 2015-09-23, from http://www.ibm.com/developerworks/library/l-fs8/index.html

Tweedie, S. C. (1998). Journaling the Linux ext2fs filesystem. In *Linuxexpo '98* (pp. 1–8). Retrieved from http://www.stanford.edu/class/cs240/readings/ext2-journal-design.pdf

Tweedie, S. C. (2000). *EXT3 , Journaling Filesystem.* Retrieved 2015-09-14, from http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html