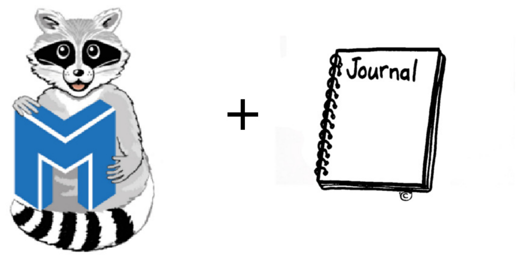
Journaling Support in MINIX 3



Author:
Niek Linnenbank

Supervisors:
David van Moolenbroek
Raja Appuswamy

February 22, 2011

1843370

Contents

1 Introduction

The filesystem is one of the most important components in modern computer systems. It is responsible for organizing data in a structured and efficient fashion. The operating system in turn provides the user with a high level API for accessing files saved on the filesystem. The operating system is also responsible for keeping the filesystem consistent, even in the event of unexpected crashes and power failures.

A traditional approach used by UNIX based operating systems is to keep the filesystem consistent is to scan the entire filesystem at startup time for any inconsistencies. This will ensure the filesystem to be consistent, yet as modern storage hardware grows in several terabytes size, scanning the entire filesystem could take several hours to complete. For many applications such downtime is unacceptible.

Fortunately there exists another class of filesystems which can avoid scanning the entire filesystem at startup time. In a journal based filesystem, the operating system writes any updates to meta-data blocks first to a so called journal log, before writing it to the actual filesystem. This technique guarantees that when an update to the filesystem has been written entirely to the journal, that the update will eventually be applied to the filesystem. In case the system crashes or experiences a power failure, the operating system will look in the journal for any completely commited updates, and applies them to the filesystem. At that point, the filesystem can be assumed to be consistent, thus no expensive scan is needed anymore.

An example operating system which has implemented a journaling filesystem is Linux. The extended 2 filesystem has been patched by Stephen Tweedie with journaling features, which is now called the extended 3 filesystem. The MINIX 3 operating system currently does not have a journaling filesystem. This document describes the implementation of the journaling MINIX 3 filesystem using the lessons learned from the Linux extended 3 filesystem implementation.

2 Implementation

The journaling implementation for MINIX consists of several components. Each are described in detail in the following subsections.

2.1 libjournal

The *libjournal* library is the central component of the journaling implementation. It contains all the logic needed for creating, opening, reading and writing a journal. The libjournal library is designed to be linked against any user program, which can be a MINIX filesystem server but also any regular user program. The main purpose of the libjournal library is to provide transactional guarantees when modifying any block device. Typically a filesystem using libjournal first writes any modification targeted for the filesystem to the log, which is called a *commit*. Only after the modification is written as a transaction to the log successfully the modifications are applied to the filesystem and is called a *checkpoint*. Whenever a crash occurs a filesystem using libjournal may quickly recover pending complete transactions from the log when remounting the filesystem, thus prevening expensive long *fsck* runs at startup.

2.1.1 Format

A journal is devided in separate same sized parts called blocks. The journal starts with a special block called the superblock which contains information about the journal, similar to a UNIX filesystem superblock. Figure 1 displays all the fields found in the superblock. The journal format used by the libjournal MINIX implementation is the same as Linux's JBD, which also allows reusing libjournal for a possible ext3 implementation on MINIX.

```
/*
 * The journal superblock.  All fields are in big-endian byte order.
 */
typedef struct journal_superblock_s
{
    /* 0x0000 */
    journal_header_t s_header;

    /* 0x000C */
    /* Static information describing the journal */
    u32_t s_blocksize;           /* journal device blocksize */
    u32_t s_maxlen;              /* total blocks in journal file */
    u32_t s_first;               /* first block of log information */

    /* 0x0018 */
    /* Dynamic information describing the current state of the log */
    u32_t s_sequence;            /* first commit ID expected in log */
    u32_t s_start;               /* blocknr of start of log */
};
```

Figure 1: The C journal superblock structure declaration.

Figure 2 shows a visual representation of the superblock information used in a journal. The *s_blocksize* determines the size of each block in the journal, including the superblock. It is important that the blocksize of the journal is the same as the filesystem using the journal to avoid data loss. The *s_first* and *s_maxlen* together describe the size of the journal. The *s_first* field represents the first block in the log which has journal log data and is typically the first block

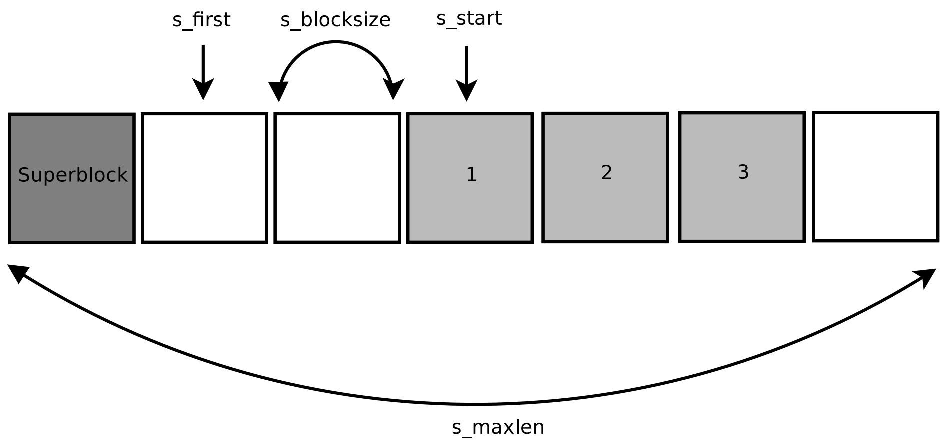


Figure 2: Format of a journal.

found after the superblock. The *s_maxlen* field contains the total number of blocks in the journal. Both are static read-only fields and should not be changed after initial creation of the journal.

To make efficient use of the available blocks in the journal it is used by libjournal as a circular log. This means that whenever the last block at the end of the journal is written, all write operations start again at the beginning of the log. Internally the libjournal library maintains a current head and tail of the log do make sure data blocks in the log are not overwritten which are still in use when cycling. Therefore when opening the journal the libjournal library must know where the journal information starts in the log. The *s_start* fields in the superblock is used for this purpose. The *s_sequence* field indicates which commit ID is the first to be expected in the log. Everytime a (group of) block(s) is written to the journal it also has a commit ID attached so that libjournal is able to distinguish old log records from the current.

2.1.2 API

The libjournal API is similar to the Linux JBD API with some minor differences. The primary function of the API is to provide a simple interface to libjournal to enable the user to make transactions. Figure 3 shows how to use the libjournal API to log a simple block modification using libjoural. The journaling session is started with *journal_init()* and *journal_destroy()*. If the journal contains valid transactions when it is opened from a previous crash they will be automatically applied to the user filesystem by *journal_load()*. Then for each transaction the user program invokes *journal_start()* and *journal_stop()* to mark the beginning and end of the transaction. Block modifications are done with *journal_get_access()* and *journal_dirty_metadata()* as shown in the figure.

The *journal_io_t jdev* and *fsdev* are structures containing I/O callbacks for reading and writing to the underlying block device, which may be a POSIX file or a MINIX device driver. They need to be initialized first before invoking any of the libjournal function as libjournal relies on correct operation of the I/O callbacks. For the complete libjournal API see appendix A.

```
#include <journal.h>

int main(void)
{
    journal_t *jp;
    journal_io_t *jdev;
    journal_io_t *fsdev;
    journal_block_t *jb;
    handle_t *h;

    /*
     * Initialize jdev, fsdev with desired I/O functions
     * for the underlying block storage device here (left out here for simplicity)
     */

    /* Open a journaling session for a 4MB journal. */
    jp = journal_init(jdev, fsdev, 0, 1024, 4096);
    if (!jp) return -1;

    /* Load the journal superblock. */
    if (journal_load(jp) != 0) return -1;

    /* Begin new transaction for 1 block. */
    h = journal_start(jp, 1);

    /* Modify block number 400 with 0xAA's. */
    jb = journal_get_access(h, 400);
    memset(jb->data, 0xAA, 4096);

    /* Mark the block dirty. Stop transaction. */
    journal_dirty_metadata(h, jb);
    journal_stop(h);

    /* Commit changes to the journal log. */
    journal_commit_transaction(jp);

    /* Now apply the block change to the filesystem. */
    journal_checkpoint_transaction(jp);

    /* Close journaling session. */
    journal_destroy(jp);
    return 0;
}
```

Figure 3: The libjournal API for making block modifications.

2.1.3 Transparent I/O

The specific method by which libjournal interacts with the block device are totally transparent to libjournal. The block device could be opened as a POSIX file from a user program, for example in the *mkfs.jmfs*, *fsck.jmfs* or test suite programs. But the block device could also be a MINIX device driver when it is used by a filesystem such as JMFS, which is accessed by message passing. The original Linux JBD implementation did not have this requirement as it is only used in the Linux kernel. In MINIX the libjournal can be (re)used both by core MINIX components such as a filesystem server but also by user level programs.

This is done with the *journal_io_t* structure. The structure contains the callbacks needed by libjournal to read and write blocks from the I/O device, which are *read_block()* and *write_block()*. Additionally the *write_all()* function allows writing multiple blocks at once to the I/O device for increased performance. For filesystems libjournal needs to interact with the block cache and it can do that with *read_cache()*, *write_cache()* and *flush()*. Finally the structure contains *malloc()* and *free()* callbacks. These are unfortunatly needed because *alloc_contig()* must be used instead of the regular *malloc()* in filesystem servers for scatter I/O and it is not available for regular user programs.

```
typedef struct journal_io_s
{
    /** Read a block from the cache. */
    journal_block_t * (*read_cache)(void *ctx, block_t nr, journal_t *jp);

    /** Put a block back on the cache. */
    int (*write_cache)(void *ctx, journal_block_t *block, journal_t *jp);

    /** Read block directly from the device. */
    journal_block_t * (*read_block)(void *ctx, block_t nr, journal_t *jp);

    /** Write block directly on the device. */
    int (*write_block)(void *ctx, journal_block_t *block, journal_t *jp);

    /** Write multiple blocks concurrently to the device. */
    int (*write_all)(void *ctx, journal_block_list_t *list, journal_t *jp);

    /** Write all currently cached user data to the device. */
    int (*flush)(void *ctx);

    /** Dynamically allocate memory. */
    void *(*malloc)(size_t size);

    /** Dynamically free memory. */
    void (*free)(void *ptr, size_t size);

    /** Optional context pointer. */
    void *context;
} journal_io_t;
```

Figure 4: The journal_io_t structure abstracts I/O functions.

2.1.4 Transaction Compaction

For efficient journaling it is desired to have transactions as large as possible, such that they can be written to the block device in one or a few I/O operation(s). The libjournal API allows user programs to start and end user transactions with the *journal_start()* and *journal_stop()* functions. Any block modifications by user transactions results in new internal copies of the modified blocks or updated copies if it already has one in the current libjournal transaction. The modified blocks will be added to the list of modified blocks in the currently running libjournal transaction. At some point libjournal has gathered enough user transactions in the larger libjournal transaction and will commit and checkpoint all user transactions as one large libjournal transaction to the actual journal.

2.2 jmfs

The *Journaled MINIX FileSystem* (JMFS) is based on the original MINIX filesystem version 3 and is fully backwards compatible. JMFS adds journaling capabilities to MFS in a similar way as ext3 in Linux. The JMFS sources are based on the current most recent MFS code but have a separate copy in the MINIX source tree and also a separate *jmfs* executable in */sbin/jmfs* when installed. This keeps the current MFS simple and allows isolated testing of JMFS without the risk of losing data on existing MFS partitions.

2.2.1 superblock

The JMFS implementation contains several new fields in the superblock. Figure 5 lists the new fields which were added to the superblock for jouraling support in JMFS. The *s_journal_inode* contains the inode number of the joural log file stored in the filesystem. This is a reserved inode and cannot be accessed by user programs. Next the *s_journal_size* field represents the size of the journal in blocks. After that comes the *s_orphan_list* field. The orphan list is used to deal with deleted files which are still open by one or more processes and are further described in section **??**. Finally the *s_last_check* contains the timestamp of the last filesystem check with the **fsck.jmfs** utility which is discussed in section 2.4.

```
EXTERN struct super_block
{
  /* ... */

  ino_t s_journal_inode;        /* inode of the journal file. */
  block_t s_journal_size;       /* size of the journal in blocks. */
  block_t s_orphan_list[2];     /* list of orphaned inodes. */
  time_t s_last_check;          /* timestamp of the last filesystem check. */

  /* ... */
} superblock;
```

Figure 5: JMFS superblock declaration.

2.2.2 journal inode

When mounting JMFS the libjournal library will need to load the journal log from the JMFS filesystem. In JMFS the journal log is saved in an inode instead of an extern device for user comfort. The journal inode is a preallocated immutable inode which is allocated by the **mkfs.jmfs** program when creating the JMFS filesystem. After creation the journal inode should never be modified by another other than libjournal to avoid data loss. When mounting JMFS the libjournal library needs to know which on-disk blocks are used by the journal inode such that it can read and write the log. Before opening the log the JMFS server assembles a list of all blocks used by the journal inode into a dynamically allocated array. The block map array is then used to find the correct journal log blocks on disk inside the *journal_io_t* callbacks of JMFS to the disk driver, as illustrated in figure 6. The journal inode block map array is kept in memory until JMFS is cleanly unmounted.

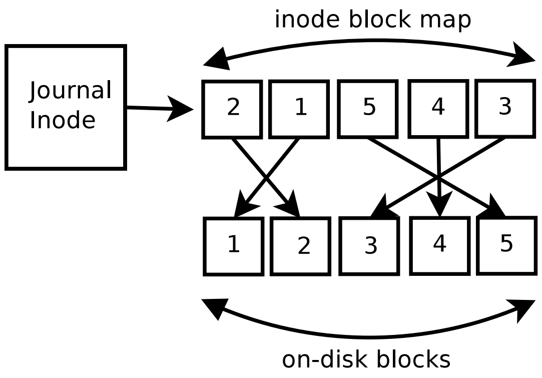


Figure 6: Journal inode block map translation to on-disk blocks.

2.2.3 block cache

The JMFS filesystem needs to perform many modifications to the block cache during normal operation. Whenever a block is read from disk it goes in the internal block cache, possibly removing another unused block entry from the cache. It is important for correctness that any blocks belonging to transactions in libjournal are not modified after they have been submitted to libjournal, as otherwise transactions could carry inconsistent block modifications. But when libjournal has a block pending for commit to the journal which was modified earlier, it should not remove it from the cache and re-read it from disk as it would fetch an old version of the block. Instead it should keep it in cache until the block was committed to the log and checkpointed to the filesystem before removing from the cache. Figure 7 illustrates the full interaction of the JMFS cache and libjournal in-memory blocks.

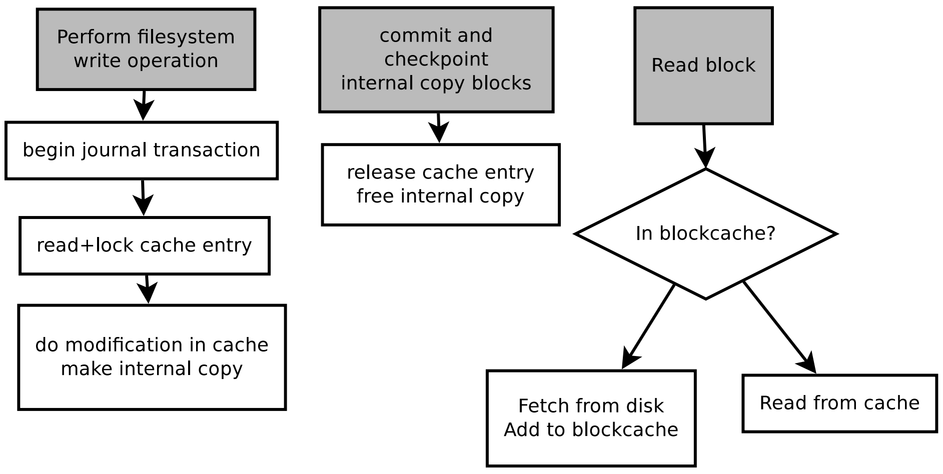

Perform filesystem write operation
begin journal transaction
read+lock cache entry
do modification in cache make internal copy
commit and checkpoint internal copy blocks
release cache entry free internal copy
Read block
In blockcache?
Fetch from disk Add to blockcache
Read from cache

Figure 7: Interaction of libjournal and JMFS cache.

2.2.4 revokes

The JMFS implementation currently only journals the meta data operations to keep performance overhead minimal. This means that any user data will go directly to the disk without interferance of the journal code. However this brings a subtile issue which needs to be dealt with in libjournal and JMFS. It could happen that a file is modified and that the transaction responsible is committed to the journal log but not yet checkpointed. Meanwhile in the current transaction the file is deleted and the block is reallocated by a different file as user data and is immediately written with fresh data from the user. If the transaction currently in the log gets replayed at recovery time, for example because the system has crashed, it will unintentionally overwrite the user data.

This scenario can be prevented by *revoking* previously committed blocks. When JMFS wants to write user data to a block and it discovers that the given block is also part of a transaction which is currently still in the journal, it will *revoke* it. The libjournal will write a special record to the log indicating that the previously logged block should not be replayed at recovery time. This effectively prevents overwriting user data with old transactions from the log.

2.2.5 orphan list

Another issue with delete operations in JMFS arrises when a file is deleted by one process but still opened by another process. When the process deletes the file it can make a transaction using libjournal to remove the directory entry for the file, but not deallocate the file contents yet as another process still has it opened. This is a problem as the delete operation needs to be an atomic operation for correctness. Therefore when removing the file it will mark the file as *orphaned* in the JMFS superblock orphan list inside the same transaction. Whenever the system crashes after the file has been deleted but before it was closed JMFS will see inside the orphans list that it needs to complete the delete operation for the given file by deallocating the blocks used by the file.

2.2.6 large transactions

Some operations in JMFS can be too large to fit inside one single transaction in the log. This can be a problem as for correctness we need to have any modifications on the filesystem be atomic. Such scenario's may arrise when a process invokes a large truncate or write operation on a file. In the current JMFS implementation this problem is solved by splitting the large truncate and write operations into smaller transactions.

2.3 mkfs.jmfs

The original MFS program for creating a new filesystem has been copied and modified for JMFS. The **mkfs.jmfs** program is now also responsible for allocating the journal inode and initializing the journal superblock. Also a new commandline flag **-j** is added to **mkfs.jmfs** to specify the size of the journal measured in blocks.

2.4 fsck.jmfs

The **fsck.jmfs** program is based on the original MFS version. Whenever a filesystem check is requested on a JMFS filesystem it first checks that the filesystem is in fact a JMFS filesystem. After that it will attempt to recover the journal before checking the filesystem for correctness. This is a critical step as it may happen that the filesystem had suffered from a crash and could contain transactions in the journal log. Additionally the **fsck.jmfs** program looks in the JMFS superblock to see how many days ago it was last fully checked. Currently the default is to enforce a **fsck.jmfs** every 30 days. This ensures that corruption due to hardware errors or other external factors are corrected, which the journaling layer cannot directly deal with.

3 Test Suite

The libjournal implementation contains an extensive set of test programs to ensure the correct behaviour under all circumstances. Every test program is compiled and linked against libjournal and uses POSIX file as the emulated block device for the journal with *journal_io_t's*. There are a total of 35 test programs written with at least one for every libjournal function. Every test program does *assert()* checking on every assumption the libjournal library makes. Additionally all block modifications and writes to the journal block device and client filesystem block device are extensively checked by each test program to ensure libjournal writes the correct information to the log and does not overwrite or corrupt anything else. The test suite uses as much randomness as possible to isolate problems with hardcoding and emulate more real-life scenario's. For example transactions are randomized in the test suite to write random data to random block numbers with libjournal.

To run the test suite make sure that libjournal is installed and then use the following commands:

```
# touch /tmp/fs /tmp/journal
# export FS_SIZE=8192 FS_PATH=/tmp/fs
# export JOURNAL_SIZE=1024 JOURNAL_PATH=/tmp/journal
# cd /usr/src/test/libjournal
# make clean all regress
```

The *FS_SIZE* and *FS_PATH* set the emulated filesystem block device size in blocks and path. It could also be pointed to a real block device in the */dev* directory. Similary *JOURNAL_SIZE* and *JOURNAL_PATH* specify the journal block device size in blocks and path. The **make** command runs the test suite. If there is no output displayed by the suite after completion the test is successfull. Otherwise an assertion error should be visible.

4 Performance

The following section describes the performance results of the journaling implementation. All measurements were done on the HP Compaq DC7900 mini tower with an Intel Core2Duo 3.33Ghz CPU, 4GB memory and a Western Digital ATA disk 8GB partition. The Loris test suite was used to measure performance of both JMFS and MFS. Each test was ran three times and the results we present are the average of the three test runs.

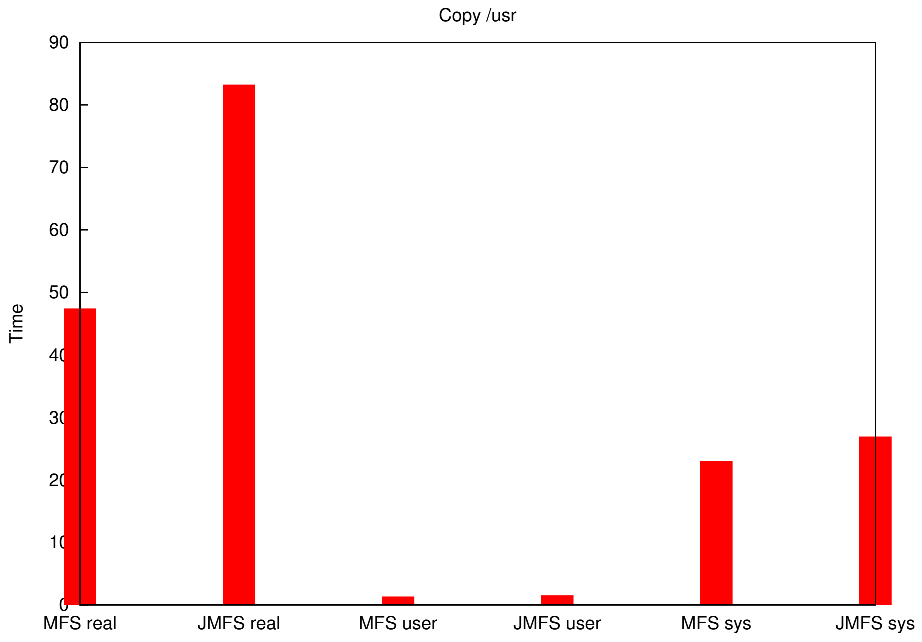


Figure 8:

The results in figure 8 illustrate the performance results of the Loris copy script. It basically copies the entire */usr* partition to a newly created (J)MFS partition. The MFS test run took a total of 47 seconds to complete versus 83 seconds for JMFS. This indicates an considerable overhead for JMFS, which is almost twice as slow compared to MFS. The reason JMFS is slower than MFS in this test is that for each metadata block update on the filesystem, JMFS needs to write two blocks: one to the journal and one to the real filesystem. MFS needs to only write one. Therefore JMFS should be twice as slow as MFS for metadata I/O operations, which is exactly the what we see in the results.
The results for the *make clean* test in figure 9 are much better for JMFS. In this test the total time for MFS is 104 seconds with JMFS at 114 seconds, only 10 seconds slower. The most plausible explanation that JMFS performs much better in this test is that the number of metadata operations in this test is lower compared to the number of userdata updates. Again in the *make world* test in figure 10 JMFS overhead is relatively low. MFS takes about 25 seconds to compile the entire MINIX system and JMFS 31 seconds. The most obvious reason would be that there are more user data operations compared to metadata operations. Next the loris find test in figure 11 shows little overhead for JMFS compared to MFS. The overhead in this test of JMFS is due to the update transactions needed for the *access times* in each file written. Finally the Loris *rm -rf* test gives much worse performance results in figure 12. For MFS it takes 23 seconds while for JMFS it takes a total of 46 seconds to remove the entire */usr* tree. This is exactly as expected as the remove operation is a purely metadata operation and should take twice as long for JMFS compared to MFS.

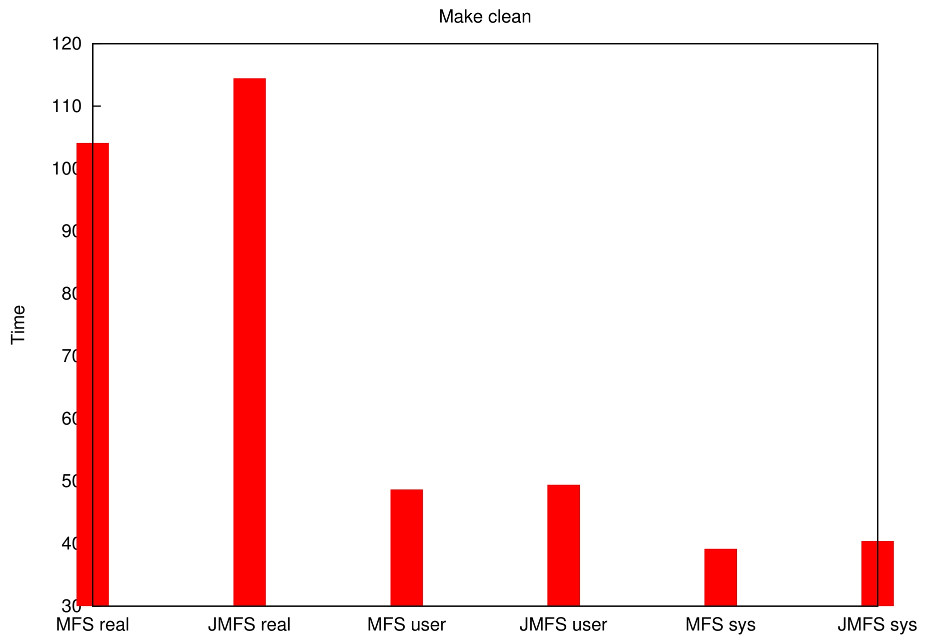


Figure 9:

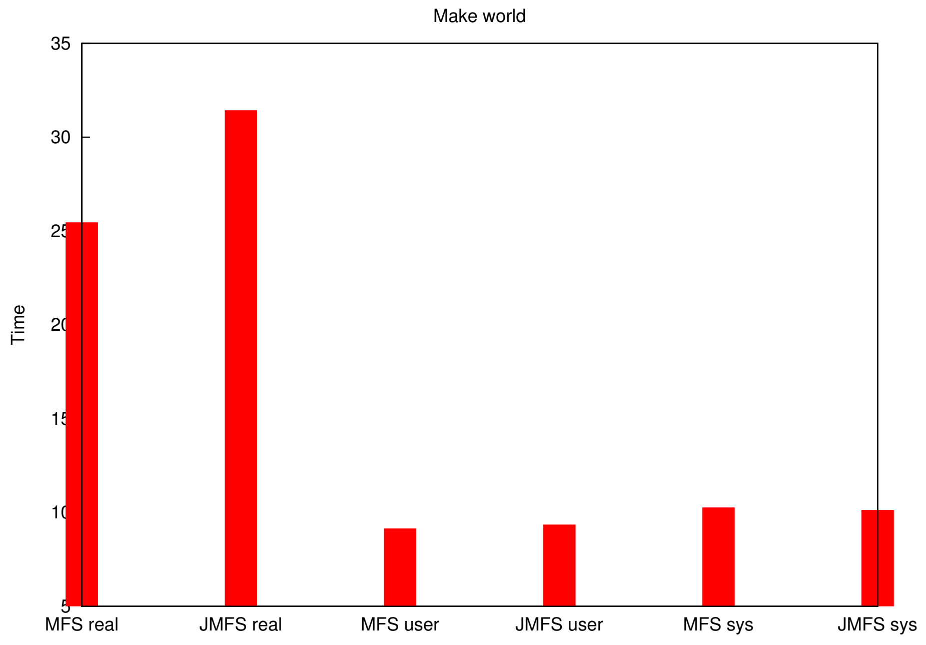


Figure 10:

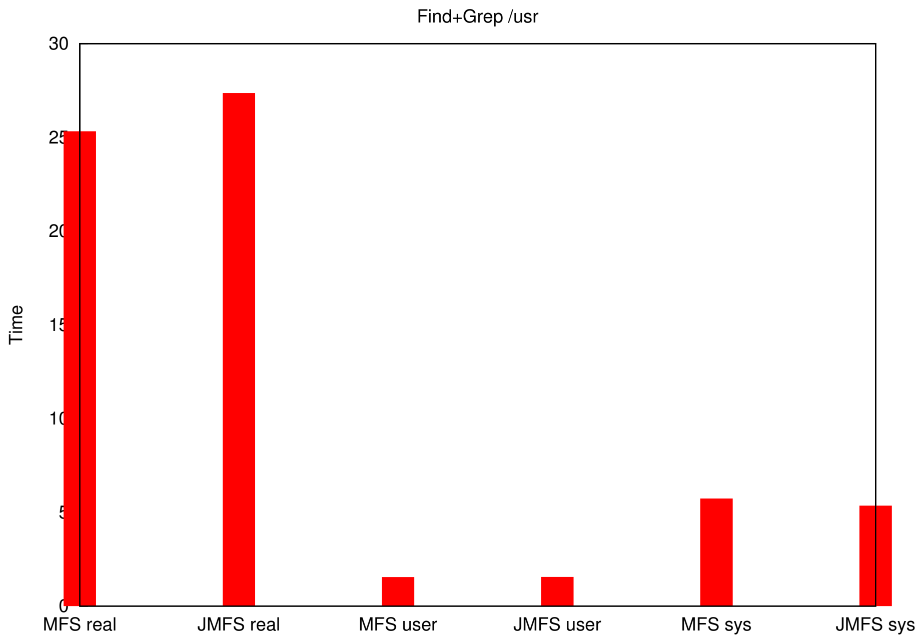


Figure 11:

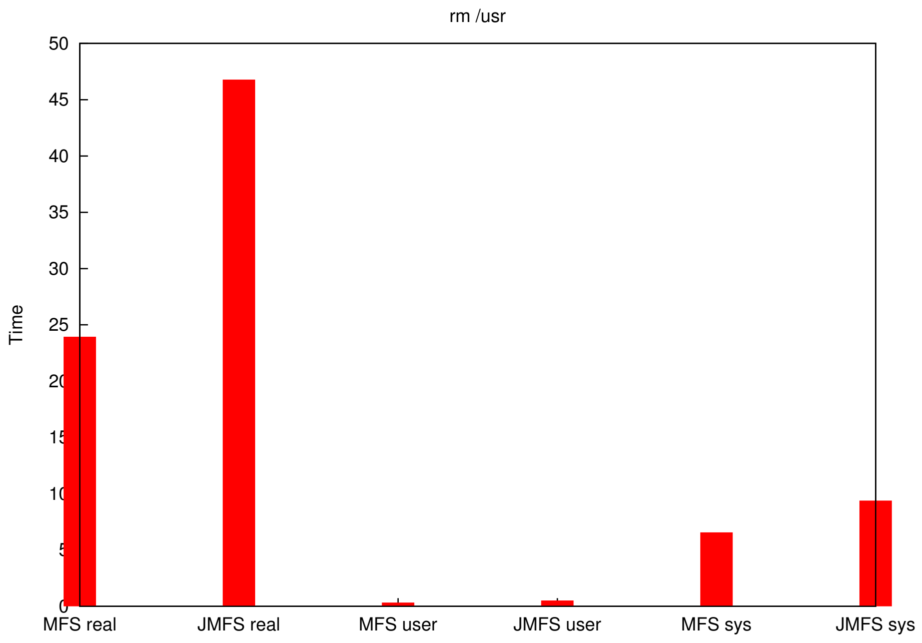


Figure 12:

5 Future Work

5.1 Filesystem specific setup utility

While the current journaling implementation for MINIX is functional, much works could be done to further improve it. For example, there should be a modification to the setup utility for MINIX which allows users to specify which filesystem to install per partition. The user could then install EXT2, JMFS or MFS. Having such utility also enables gradual integration of JMFS into MINIX, for example by having only */usr* with JMFS and critical data stored on */home* with MFS. During the first months of the JMFS release in MINIX any problems found affecting filesystem correctness could then be found and fixed and meanwhile minimizing data loss in case such problems occur.

5.2 Asynchroneous commit and checkpointing

The reason for the current I/O overhead in libjournal is that committing and checkpointing to the journal takes time. It is possible to avoid this overhead from a user perspective by *delaying* it. Currently in Linux this is done by having a background kernel thread which performs the commit and checkpointing. Whenever a user performs metadata transactions the filesystem keeps all metadata updates in memory but writes user data directly to disk. The commit and checkpoints are delayed until the most opportune moment, performance wise. The user will not be kept blocked waiting for the checkpointing and commits to complete. In MINIX it is possible to do the same, even in the current single process, single thread design. We could let JMFS send *asynchroneous I/O* requests to the disk driver to perform the commit and checkpoints. Another possiblity would be to do the same as Linux whenever kernel threads are implemented in MINIX.

5.3 JBD2 64-bit blocks

The latest version of the extended filesystem for Linux supports JBD2 which has 64-bit block numbers instead of 32-bit. It could be an interesting project in case a new MFS version with 64-bit blocks is implemented.

6 Conclusion

This document discusses the implementation of the journaling support in the MINIX filesystem. It contains a description of the implemented libjournal library, the JMFS server and the fsck.jmfs and mkfs.jmfs programs. Additionally the implementation has an extensive test suite consisting of 35 test programs to ensure the best correctness quality possible for libjournal. Current performance of the journaling implementation is acceptable for regular usage, however most of the overhead can be removed using asynchroneous commits and checkpoints. Finally, I'd like to add that I have learned a lot in this project and I hope that the MINIX developers will find it a usefull addition to their codebase.

A journal.h

```
/*
 * This file is part of libjournal.
 *
 * Written by Niek Linnenbank <nieklinnenbank@gmail.com>
 */

#ifndef _LIBJOURNAL_H
#define _LIBJOURNAL_H

#include <minix/types.h>
#include <sys/queue.h>

/**
 * Maximum number of tags per block descriptor.
 */
#define JOURNAL_NTAGS_PER_DESC(jp) ( ((jp)->j_blocksize - sizeof(journal_header_t)) \
    / (sizeof(journal_block_tag_t)))

/**
 * Maximum number of blocks per transaction.
 */
#define JOURNAL_NBLOCKS_PER_TRANS(jp) ((jp)->j_maxlen / 4)

/**
 * @brief Maximum number of total on-disk blocks needed in the log
 *        to store the given number of user blocks.
 */
#define JOURNAL_NBLOCKS_NEEDED(jp,num) \
    ((((jp)->j_running_transaction->t_outstanding_credits) + (num)) + \
    (((((jp)->j_running_transaction->t_outstanding_credits) + (num)) / \
       JOURNAL_NTAGS_PER_DESC(jp)) + 2))

/**
 * typedef handle_t - The handle_t type represents a single
 *                    atomic update being performed by some process.
 *
 * All filesystem modifications made by the process go
 * through this handle.  Recursive operations (such as quota operations)
 * are gathered into a single update.
 *
 * The buffer credits field is used to account for journaled buffers
 * being modified by the running process.  To ensure that there is
 * enough log space for all outstanding operations, we need to limit the
 * number of outstanding buffers possible at any time.  When the
 * operation completes, any buffer credits not used are credited back to
 * the transaction, so that at all times we know how many buffers the
 * outstanding updates on a transaction might possibly touch..
 *
 * This is an opaque datatype.
 */
typedef struct handle_s handle_t;       /* Atomic operation type */
```

```
/**
 * typedef transaction_t - Represents a compound set of atomic updates.
 *
 * This is an opaque datatype.
 */
typedef struct transaction_s transaction_t;  /* Compound transaction type */

/**
 * typedef journal_t - The journal_t maintains all of the journaling
 *                     state information for a single filesystem.
 *
 * journal_t is linked to from the fs superblock structure.
 *
 * We use the journal_t to keep track of all outstanding transaction
 * activity on the filesystem, and to manage the state of the log
 * writing process.
 *
 * This is an opaque datatype.
 */
typedef struct journal_s journal_t;      /* Journal control structure */

/**
 * Represents a single in-memory block.
 */
typedef struct journal_block_s
{
    block_t blocknr;  /** Block number. */
    u8_t *data;  /** Pointer to in-memory data. */
    void *context;  /** Optional context pointer. */
    char free_on_remove; /** Invoke free() on data when removed. */
    TAILQ_ENTRY(journal_block_s) b_next; /** Linked list entry. */
}
journal_block_t;

/** Double-List of journal blocks. */
typedef TAILQ_HEAD(journal_block_list_s, journal_block_s) journal_block_list_t;

/**
 * Abstracts I/O operations of the underlying device.
 */
typedef struct journal_io_s
{
    /**
     * Read a block from the cache.
     * @param ctx Optional context pointer of the journal_io_t.
     * @param nr Block number to read.
     * @param jp Journal_t pointer.
     * @return Journal_block_t pointer on success and NULL on failure.
     */
    journal_block_t * (*read_cache)(void *ctx, block_t nr, journal_t *jp);

    /**
     * Put a block back on the cache.
     * @param ctx Optional context pointer of the journal_io_t.
     * @param block Journal_block_t pointer.
     * @param jp Journal_t pointer.
     * @return Zero on success and non-zero on failure.
     */
    int (*write_cache)(void *ctx, journal_block_t *block, journal_t *jp);

    /**
     * Read block directly from the device.
     * @param ctx Optional context pointer of the journal_io_t.
     * @param nr Block number to read.
     * @param jp Journal_t pointer.
     * @return Journal_block_t pointer on success and NULL on failure.
     */
    journal_block_t * (*read_block)(void *ctx, block_t nr, journal_t *jp);

    /**
     * Write block directly on the device.
     * @param ctx Optional context pointer of the journal_io_t.
     * @param block Journal_block_t pointer.
     * @param jp Journal_t pointer.
     * @return Zero on success and non-zero on failure.
     */
    int (*write_block)(void *ctx, journal_block_t *block, journal_t *jp);

    /**
     * Write multiple blocks concurrently to the device.
     * @param ctx Optional context pointer of the journal_io_t.
     * @param list Journal_block_list_t pointer.
     * @param jp Journal_t pointer.
     * @return Zero on success and non-zero on failure.
     */
    int (*write_all)(void *ctx, journal_block_list_t *list, journal_t *jp);

    /**
     * Write all currently cached user data to the device.
     * @param ctx Optional context pointer of the journal_io_t.
     * @return Zero on success and non-zero on failure.
     */
    int (*flush)(void *ctx);

    /**
     * Dynamically allocate memory.
     * @param size Number of bytes to allocate.
     * @return Allocated memory pointer on success or NULL on failure.
     * @note We need this as jmfs needs alloc_contig() for rw_scatter
     *       but the userspace tests cannot link with libsys.
     */
    void *(*malloc)(size_t size);

    /**
     * Dynamically free memory.
     * @param ptr Previously allocated memory.
     * @param size Optional argument to specify number of bytes to free.
     */
```

```
    void (*free)(void *ptr, size_t size);

    /** Optional context pointer. */
    void *context;
}
journal_io_t;

/**
 * struct handle_s - The handle_s type is the concrete type associated with
 *                   handle_t.
 * @h_transaction: Which compound transaction is this update a part of?
 * @h_buffer_credits: Number of remaining buffers we are allowed to dirty.
 * @h_ref: Reference count on this handle
 */
struct handle_s
{
    /* Which compound transaction is this update a part of? */
    transaction_t *h_transaction;

    /* Number of remaining buffers we are allowed to dirty: */
    int h_buffer_credits;

    /* Reference count on this handle */
    int h_ref;
};

/*
 * The transaction_t type is the guts of the journaling mechanism.  It
 * tracks a compound transaction through its various states:
 *
 * RUNNING:     accepting new updates
 * LOCKED:      Updates still running but we don't accept new ones
 * RUNDOWN:     Updates are tidying up but have finished requesting
 *              new buffers to modify (state not used for now)
 * FLUSH:       All updates complete, but we are still writing to disk
 * COMMIT:      All data on disk, writing commit record
 * FINISHED:    We still have to keep the transaction for checkpointing.
 *
 * The transaction keeps track of all of the buffers modified by a
 * running transaction, and all of the buffers committed but not yet
 * flushed to home for finished transactions.
 */
struct transaction_s
{
    /* Pointer to the journal for this transaction. [no locking] */
    journal_t *t_journal;

    /* Sequence number for this transaction [no locking] */
    u32_t t_tid;

    /*
     * Transaction's current state
     * [no locking - only kjournald alters this]
     * FIXME: needs barriers
     * KLUDGE: [use j_state_lock]
     */
    enum
    {
        T_RUNNING,
        T_LOCKED,
        T_RUNDOWN,
        T_FLUSH,
        T_COMMIT,
        T_FINISHED
    }
    t_state;

    /*
     * Where in the log does this transaction's commit start? [no locking]
     */
    unsigned long t_log_start;

    /* Number of committed blocks used in the log by this transaction. */
    unsigned long t_log_blocks;

    /* Number of buffers on the t_buffers list [j_list_lock] */
    int t_nr_buffers;

    /*
     * Doubly-linked circular list of all buffers reserved but not yet
     * modified by this transaction [j_list_lock]
     */
    TAILQ_HEAD(reserved_head, journal_block_s) t_reserved_list;

    /*
     * Doubly-linked circular list of all metadata buffers owned by this
     * transaction [j_list_lock]
     */
    TAILQ_HEAD(buffers_head, journal_block_s) t_buffers;

    /*
     * Linked list containing revoked blocks.
     */
    TAILQ_HEAD(revoke_head, journal_block_s) t_revoke_list;

    /*
     * Number of buffers reserved for use by all handles in this transaction
     * handle but not yet modified. [t_handle_lock]
     */
    int t_outstanding_credits;

    /*
     * Forward and backward links for the circular list of all transactions
     * awaiting checkpoint. [j_list_lock]
     */
    TAILQ_ENTRY(transaction_s) t_next;
};
```

```
/*
 * On-disk structures
 */

#define JFS_MAGIC_NUMBER 0xc03b3998U /* The first 4 bytes of /dev/random! */

/*
 * Descriptor block types:
 */

#define JFS_DESCRIPTOR_BLOCK    1
#define JFS_COMMIT_BLOCK        2
#define JFS_SUPERBLOCK_V1       3
#define JFS_SUPERBLOCK_V2       4
#define JFS_REVOKE_BLOCK        5

/*
 * Standard header for all descriptor blocks:
 */
typedef struct journal_header_s
{
    u32_t h_magic;
    u32_t h_blocktype;
    u32_t h_sequence;
}
journal_header_t;

/*
 * The block tag: used to describe a single buffer in the journal.
 */
typedef struct journal_block_tag_s
{
    u32_t t_blocknr; /* The on-disk block number */
    u32_t t_flags;   /* See below */
}
journal_block_tag_t;

/*
 * The revoke descriptor: used on disk to describe a series of blocks to
 * be revoked from the log.
 */
typedef struct journal_revoke_header_s
{
    journal_header_t r_header;
    u32_t r_count; /* Count of bytes used in the block */
}
journal_revoke_header_t;

/*
 * Definitions for the journal tag flags word
 */
#define JFS_FLAG_ESCAPE         1       /* on-disk block is escaped */
#define JFS_FLAG_SAME_UUID      2       /* block has same uuid as previous */
#define JFS_FLAG_DELETED        4       /* block deleted by this transaction */
```

```
#define JFS_FLAG_LAST_TAG       8       /* last tag in this descriptor block */

/*
 * The journal superblock.  All fields are in big-endian byte order.
 */
typedef struct journal_superblock_s
{
    /* 0x0000 */
    journal_header_t s_header;

    /* 0x000C */
    /* Static information describing the journal */
    u32_t s_blocksize;              /* journal device blocksize */
    u32_t s_maxlen;                 /* total blocks in journal file */
    u32_t s_first;                  /* first block of log information */

    /* 0x0018 */
    /* Dynamic information describing the current state of the log */
    u32_t s_sequence;               /* first commit ID expected in log */
    u32_t s_start;                  /* blocknr of start of log */

    /* 0x0020 */
    /* Error value, as set by journal_abort(). */
    u32_t s_errno;

    /* 0x0024 */
    /* Remaining fields are only valid in a version-2 superblock */
    u32_t s_feature_compat;         /* compatible feature set */
    u32_t s_feature_incompat;       /* incompatible feature set */
    u32_t s_feature_ro_compat;      /* readonly-compatible feature set */
    /* 0x0030 */
    u8_t  s_uuid[16];               /* 128-bit uuid for journal */
    /* 0x0040 */
    u32_t s_nr_users;               /* Nr of filesystems sharing log */
    u32_t s_dynsuper;               /* Blocknr of dynamic superblock copy*/

    /* 0x0048 */
    u32_t s_max_transaction;        /* Limit of journal blocks per trans.*/
    u32_t s_max_trans_data;         /* Limit of data blocks per trans. */

    /* 0x0050 */
    u32_t s_padding[44];

    /* 0x0100 */
    u8_t  s_users[16*48];           /* ids of all fs'es sharing the log */
    /* 0x0400 */
}
journal_superblock_t;

/**
 * struct journal_s - The journal_s type is the concrete type
 *                    associated with journal_t.
 */
struct journal_s
```

```
{
    /* The superblock buffer */
    journal_block_t *j_sb_buffer;
    journal_superblock_t *j_superblock;

    /* Version of the superblock format */
    int j_format_version;

    /*
     * Transactions: The current running transaction...
     * [j_state_lock] [caller holding open handle]
     */
    transaction_t *j_running_transaction;

    /*
     * the transaction we are pushing to disk
     * [j_state_lock] [caller holding open handle]
     */
    transaction_t *j_committing_transaction;

    /*
     * ... and a linked circular list of all transactions waiting for
     * checkpointing. [j_list_lock]
     */
    TAILQ_HEAD(check_trans_head, transaction_s) j_checkpoint_transactions;

    /*
     * Journal head: identifies the first unused block in the journal.
     * [j_state_lock]
     */
    unsigned long j_head;

    /*
     * Journal tail: identifies the oldest still-used block in the journal.
     * [j_state_lock]
     */
    unsigned long j_tail;

    /*
     * Journal free: how many free blocks are there in the journal?
     * [j_state_lock]
     */
    unsigned long j_free;

    /*
     * Journal start and end: the block numbers of the first usable block
     * and one beyond the last usable block in the journal. [j_state_lock]
     */
    unsigned long j_first;
    unsigned long j_last;

    /*
     * Device, blocksize and starting block offset for the location where we
     * store the journal.
     */
    journal_io_t *j_dev;
    size_t j_blocksize;
    block_t j_blk_offset;

    /* Total maximum capacity of the journal region on disk. */
    size_t j_maxlen;

    /*
     * Device which holds the client fs.  For internal journal this will be
     * equal to j_dev.
     */
    journal_io_t *j_fs_dev;

    /*
     * Sequence number of the oldest transaction in the log [j_state_lock]
     */
    u32_t j_tail_sequence;

     /*
     * Sequence number of the next transaction to grant [j_state_lock]
     */
    u32_t j_transaction_sequence;

    /*
     * Journal uuid: identifies the object (filesystem, LVM volume etc)
     * backed by this journal.  This will eventually be replaced by an array
     * of uuids, allowing us to index multiple devices within a single
     * journal and to perform atomic updates across them.
     */
    u8_t j_uuid[16];
};

/*
 * journal.c
 */

/**
 * journal_t * journal_init() - creates and initialises a journal structure
 *
 * @jdev: Block device on which to create the journal.
 * @fsdev: Block device which hold journalled filesystem for this journal.
 * @start: Block nr Start of journal.
 * @length:  Lenght of the journal in blocks.
 * @blocksize: blocksize of journalling device
 * @returns: a newly created journal_t *
 *
 * journal_init() creates a journal which maps a fixed contiguous
 * range of blocks on an arbitrary block device.
 */
_PROTOTYPE( journal_t * journal_init, (journal_io_t *jdev, journal_io_t *fsdev,
       block_t start, size_t length,
       size_t block_size));
```

```
/**
 * int journal_create() - Initialise the new journal file
 * @journal: Journal to create. This structure must have been initialised
 *
 * Given a journal_t structure which tells us which disk blocks we can
 * use, create a new journal superblock and initialise all of the
 * journal fields from scratch...
 */
_PROTOTYPE( int journal_create, (journal_t *journal) );

/**
 * int journal_load() - Load an existing journal file
 * @journal: Journal to load. This structure must have been initialised
 *
 * Given a journal_t structure which tells us where the journal is
 * located on disk, we attempt to load the superblock, perform recovery
 * if needed and start a new journalling session.
 */
_PROTOTYPE( int journal_load, (journal_t *journal) );

/**
 * int journal_write_superblock() - Update the in-memory superblock and
 *     write it back to the journaling device.
 * @journal: Journal to update the superblock for.
 */
_PROTOTYPE( int journal_write_superblock, (journal_t *journal));

/**
 * int journal_destroy() - Close a journaling session.
 * @journal: Journal to close.
 */
_PROTOTYPE( int journal_destroy, (journal_t *journal));

/**
 * int journal_reset() - Reset a journaling session.
 * @journal: Journal to reset.
 *
 * Internal libjournal function to reset the dynamic fields of a journal_t.
 */
_PROTOTYPE( int journal_reset, (journal_t *journal));

/**
 * int journal_verify() - Check a journal_t for validity.
 * @journal: Journal_t to verify.
 *
 * Internal libjournal function to validate a journal_t pointer.
 */
_PROTOTYPE( int journal_verify, (journal_t *journal));

/*
 * transaction.c
 */

/**
```

```
 * int journal_start() - Begins a transaction.
 * @journal: Journal pointer.
 * @nblocks: Number of blocks required at maximum in this transaction.
 *
 * Returns a transaction handle.
 */
_PROTOTYPE( handle_t * journal_start, (journal_t *journal, int nblocks));

/**
 * journal_block_t * journal_get_access() - Retrieve a filesystem block for modification.
 * @handle: Transaction handle.
 * @blocknr: Block number to start modifying.
 *
 * Returns a pointer to a journal_block_t which may be modified by the client filesystem.
 */
_PROTOTYPE( journal_block_t * journal_get_access, (handle_t *handle, block_t blocknr));

/**
 * int journal_forget() - Removes a block from the current transaction.
 * @handle: Transaction handle.
 * @blocknr: Block number to remove.
 *
 * Returns zero on success and -1 on failure.
 */
_PROTOTYPE( int journal_forget, (handle_t *handle, journal_block_t *block));

/**
 * int journal_dirty_metadata() - Marks a journal_block_t modified.
 * @handle: Transaction handle.
 * @block: Modified block pointer.
 *
 * After modification of a journal_block_t it must be marked
 * modified in order to be written to the journaling log.
 */
_PROTOTYPE( int journal_dirty_metadata, (handle_t *handle,
 journal_block_t *block));

/**
 * int journal_stop() - Closes a transaction.
 * @handle: Transaction handle.
 */
_PROTOTYPE( int journal_stop, (handle_t *handle));

/**
 * transaction_t * journal_new_trans() - Create a new transaction_t.
 * @journal: Journal pointer.
 *
 * Internal libjournal function to initialize a new transaction_t.
 */
_PROTOTYPE( transaction_t * journal_new_trans, (journal_t *journal));

/**
 * void journal_free_trans() - Destroys a transaction_t.
 * @journal: Journal pointer.
 *
 * Internal libjournal function to deallocate a transaction_t.
 */
_PROTOTYPE( void journal_free_trans, (transaction_t *t));

/**
 * int journal_verify_handle() - Verify a handle_t.
 * @journal: Journal pointer.
 *
 * Internal libjournal function to validate a handle_t.
 */
_PROTOTYPE( int journal_verify_handle, (handle_t *handle));

/*
 * block.c
 */

/**
 * journal_block_t * journal_read_block() - Read a block from an I/O device.
 * @jp: Journal pointer.
 * @io: Journal I/O pointer.
 * @nr: Block number to read.
 *
 * Returns a journal_block_t for the given blocknr.
 */
#define journal_read_block(jp,io,nr) \
    (io->read_block(io->context, (nr), (jp)))

/**
 * int journal_write_block() - Write a block to an I/O device.
 * @jp: Journal pointer.
 * @io: Journal I/O pointer.
 * @blk: Block to write.
 */
#define journal_write_block(jp,io,blk) \
    (io->write_block(io->context, (blk), (jp)))

/**
 * journal_block_t * journal_new_block() - Creates a new journal_block_t.
 * @journal: Journal pointer.
 * @nr: Block number.
 *
 * Internal libjournal function to create a new journal_block_t.
 */
_PROTOTYPE( journal_block_t * journal_new_block, (journal_t *jp, block_t nr));

/**
 * journal_block_t * journal_from_block() - Creates a new journal_block_t using a buffer.
 * @journal: Journal pointer.
 * @nr: Block number.
 * @data: Block data buffer to (re)use.
 *
 * Internal libjournal function to create a new journal_block_t.
 */
```

```
_PROTOTYPE( journal_block_t * journal_from_block, (journal_t *jp, block_t nr,
   void *data));

/**
 * journal_block_t * journal_next_block() - Retrieve the next unused block in the log.
 * @journal: Journal pointer.
 *
 * Internal libjournal function to get the next unused block.
 */
_PROTOTYPE( journal_block_t * journal_next_block, (journal_t *journal));

/**
 * void journal_free_block() - Destroys a journal_block_t.
 * @jp: Journal_t pointer.
 * @block: Block pointer.
 *
 * Internal libjournal function to deallocate a journal_block_t.
 */
_PROTOTYPE( void journal_free_block, (journal_t *jp, journal_block_t *block));

/**
 * void journal_free_block_list() - Destroys each journal_block_t in given list.
 * @jp: Journal_t pointer.
 * @blk: Block pointer variable.
 * @lst: journal_block_list_t pointer.
 * @btmp: Temporary block pointer variable.
 *
 * Internal libjournal function to deallocate blocks in a journal_block_list_t.
 */
#define journal_free_block_list(jp,blk,lst,btmp) \
    TAILQ_FOREACH_SAFE((blk), (lst), b_next, (btmp)) \
    { \
        journal_free_block((jp),(blk)); \
    }

/**
 * int journal_verify_block() - Verify a journal_block_t.
 * @block: Journal block pointer.
 *
 * Internal libjournal function to validate a journal_block_t.
 */
_PROTOTYPE( int journal_verify_block, (journal_block_t *block));

/**
 * journal_block_t * journal_find_block() - Find a given block number in the journal.
 * @journal: Journal pointer.
 *
 * Finds the first occurence of a journal block with the given block number.
 * Search starts with the running transaction, committing transaction and
 * finally checkpointing transactions.
 */
_PROTOTYPE( journal_block_t * journal_find_block, (journal_t *jp, block_t nr));

/*
```

```
 * io.c
 */

/**
 * int journal_posix_read() - Read a block from a POSIX file.
 * @ctx: Context pointer is a file descriptor number.
 * @nr: Block number to read.
 * @jp: Journal pointer.
 *
 * Internal libjournal test I/O backend to read from a POSIX file.
 */
_PROTOTYPE( journal_block_t * journal_posix_read, (void *ctx, block_t nr,
   journal_t *jp));

/**
 * int journal_posix_write() - Write a block to a POSIX file.
 * @ctx: Context pointer is a file descriptor number.
 * @block: Block to write.
 * @jp: Journal pointer.
 *
 * Internal libjournal test I/O backend to write to a POSIX file.
 */
_PROTOTYPE( int journal_posix_write, (void *ctx, journal_block_t *block,
      journal_t *jp));

/**
 * int journal_posix_writeall() - Write multiple blocks at once to a POSIX file.
 * @ctx: Context pointer is a file descriptor number.
 * @block: Block list to write.
 * @jp: Journal pointer.
 *
 * Internal libjournal test I/O backend to write to a POSIX file.
 */
_PROTOTYPE( int journal_posix_writeall, (void *ctx, journal_block_list_t *list,
         journal_t *jp));
/**
 * int journal_posix_flush() - Makes sure that the file is synced to disk.
 * @ctx: Context pointer is a file descriptor number.
 *
 * Internal libjournal test I/O backend to flush to a POSIX file.
 */
_PROTOTYPE( int journal_posix_flush, (void *ctx));

/**
 * int journal_posix_malloc() - Allocate memory with malloc().
 * @size: number of bytes to allocate.
 *
 * Internal libjournal test I/O backend to allocate memory.
 */
_PROTOTYPE( void * journal_posix_malloc, (size_t size));

/**
 * int journal_posix_free() - Release memory with free().
 * @ptr: memory to release.
 * @size: Optional argument to specify number of bytes to free.
 *
 * Internal libjournal test I/O backend to release memory.
 */
_PROTOTYPE( void journal_posix_free, (void *ptr, size_t size));

/*
 * commit.c
 *
 * int journal_commit_transaction() - Commit the current transaction.
 * @journal: Journal pointer.
 *
 * After a transaction is commited, it will (eventually) be applied
 * onto the client filesystem.
 */
_PROTOTYPE( int journal_commit_transaction, (journal_t *journal));

/*
 * checkpoint.c
 *
 * int journal_checkpoint_transaction() - Checkpoint all commited transactions.
 * @journal: Journal pointer.
 *
 * A checkpoint operation will copy the modified blocks in a
 * committed transaction onto the real client filesystem.
 */
_PROTOTYPE( int journal_checkpoint_transaction, (journal_t *journal));

/*
 * recover.c
 *
 * int journal_recover() - Replay committed transaction after a crash.
 * @jp: Journal pointer.
 *
 * Replays committed transaction from the journal log after a crash.
 */
_PROTOTYPE( int journal_recover, (journal_t *jp));

/*
 * revoke.c
 */

/*
 * int journal_revoke() - Invalidate previously committed blocks.
 * @h: Transaction handle.
 * @blocknr: Block number to invalidate.
 *
 * Internal function only. Revokes a block from the journal
 * to prevent replaying the block at recovery time. End-users
 * should use journal_cond_revoke() instead.
 */
_PROTOTYPE( int journal_revoke, (handle_t *h, block_t blocknr));

/*
```

```
 * int journal_cond_revoke() - Invalidate previously committed blocks if found.
 * @h: Transaction handle.
 * @blocknr: Block number to invalidate.
 *
 * Invokes journal_revoke() on the given block number if it is
 * found in the currently running transaction or any pending-for-checkpoint.
 */
_PROTOTYPE( int journal_cond_revoke, (handle_t *h, block_t blocknr));

/*
 * int journal_cancel_revoke() - Cancels a previous revoke.
 * @h: Transaction handle.
 * @blocknr: Block number to cancel revoke.
 *
 * Cancels a previously revoked block. This typically happens
 * after a block is revoked and then journalled again.
 */
_PROTOTYPE( int journal_cancel_revoke, (handle_t *h, block_t blocknr));

#endif /* _LIBJOURNAL_H */
```