EXT3 FileSystem

蔡德聖

tsaits@csie.nctu.edu.tw

Outline

- Introduction
- Data structure
- Commit transaction flow
- Checkpoint
- Recovery

EXT3

- It was written by Dr Stephen C. Tweedie for 2.2 kernels
- The filesystem was ported to 2.4 kernels by Peter Braam, Andreas Dilger and Andrew Morton , with much valuable assistance from Stephen Tweedie
- A journal file system

Design

- Goal : use EXT2 and complete backwards and forwards compatibility between EXT2 and EXT3
- one is the ***abstract journaling layer*** and one, a simple set of modifications to EXT3 to ***add transactions.***
- transactions
- batch all updates(handles) off into very large transactions and just send them all out at once

Original Inode For test.file:

inode: 777
permissions
file size
last access time
last modification
data blocks: 3110, 3111, 3506

Updated Inode For test file:

inode: 777
permissions
file size
last access time
last modification
data blocks: 3110, 3111, 3506, 3790, 3791

Log Records:

```
Inode 777:    intent-to-commit
Block 3111:   data update (changes)
Block 3506:   data update (changes)
Block 3790:   data update (changes)
Block 3791:   data update (changes)
Inode 777:    update data block list
              to 3110, 3111, 3506,
              3790, 3791
Inode 777:    access time 21:59
              7-Jun-2000
Inode 777:    modification time 21:59
              7-Jun-2000
Inode 777:    committed
```

Why EXT3

- Availability – reduce long time fsck
- Data Integrity – data consistency
- Speed – optimizes hard drive head motion
  1. Data = writeback (heavy synchronous writes)
  2. Data = ordered (throw out garbage)
  3. Data = journal (need large space for this)
- Easy Transition – mke2fs

Commit

1. writing all of the things which that transaction modified to the journal, and then
2. writing a commit record

- Represent a complete consistency to the disk

Checkpoint

- Why : a limited amount of space in the log
- flushing all the contents of the log out to the main disk
- That’s handled by the JFS layer

Recovery

- Find revoke blocks in the log
- Replay any un-revoke blocks in the log

Data structure (jbd.h)

```
        on-disk 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
         journaling buffer types
#define BJ_None             0           /* Not journaled */
#define BJ_SyncData         1           /* Normal data: flush before commit */
#define BJ_AsyncData        2           /* writepage data: wait on it before commit */
#define BJ_Metadata         3           /* Normal journaled metadata */
#define BJ_Forget           4           /* Buffer superceded by this transaction */
#define BJ_IO                           5               /* Buffer is for temporary IO use */
#define BJ_Shadow           6           /* Buffer contents being shadowed to the log */
#define BJ_LogCtl           7           /* Buffer contains log descriptors */
#define BJ_Reserved         8           /* Buffer is reserved for access by journal */
#define BJ_Types            9
```

journal superblock

```
typedef struct journal_superblock_s{
journal_header_t        s_header ;
/* Static information describing the journal */
__u32          s_blocksize;              /* journal device blocksize */
__u32          s_maxlen;                 /* total blocks in journal file */
__u32          s_first;                  /* first block of log information */
/* Dynamic information describing the current state of the log */
__u32          s_sequence;               /* first commit ID expected in log */
__u32          s_start;                  /* blocknr of start of log */
__s32          s_errno;
/* Remaining fields are only valid in a version-2 superblock */
__u32          s_feature_compat;         /* compatible feature set */
__u32          s_feature_incompat;       /* incompatible feature set */
__u32          s_feature_ro_compat;      /* readonly-compatible feature set */
__u8           s_uuid[16];               /* 128-bit uuid for journal */
__u32          s_nr_users;               /* Nr of filesystems sharing log */
__u32          s_dynsuper;               /* Blocknr of dynamic superblock copy*/
```

journal superblock

```
__u32        s_max_transaction;     /* Limit of journal blocks per trans.*/
__u32        s_max_trans_data;      /* Limit of data blocks per trans. */
__u32        s_padding[44];
__u8s_users[16*48];                 /* ids of all fs'es sharing the log */
} journal_superblock_t;
```

transaction_t

```
struct transaction_s
{
journal_t *             t_journal;          /* Pointer to the journal for this transaction. */
tid_t                   t_tid;
enum {
                T_RUNNING,
                T_LOCKED,
                T_RUNDOWN,
                T_FLUSH,
                T_COMMIT,
                T_FINISHED
        }t_state;
unsigned long           t_log_start;        /* Where in the log does this transaction's commit start? */
struct inode *          t_ilist;            /*list of all inodes owned by thistransaction */
int                     t_nr_buffers;
struct journal_head *   t_reserved_list;    /* Doubly-linked circular list of all buffers reserved but not modified by
```

states for transaction

RUNNING: accepting new updates
LOCKED: Updates still running but we don't accept new ones
RUNDOWN: not used
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.

transaction_t

```
struct journal_head *   t_buffers;          /* list of all metadata buffers */
struct journal_head *   t_sync_datalist;    /* list of all data buffers  */
struct journal_head *   t_async_datalist;   /* list of all writepage data buffers   still to be written */
struct journal_head *   t_forget;
struct journal_head *   t_checkpoint_list;//all buffers still to be  flushed before this transaction can be
                                                                                          checkpointed
struct journal_head *   t_iobuf_list;   /*temporary buffers currently undergoing IO in the log */
struct journal_head *   t_shadow_list;   /*  list of metadata buffers being shadowed by log IO. */
struct journal_head *   t_log_list;   /* control buffers being written to the log. */
int                     t_updates;
int                     t_outstanding_credits;
transaction_t           *t_cpnext, *t_cpprev;   // inks for the circular list of all transactions awaiting
                                                                                          checkpoint.
int                     t_handle_count;
};
```

journal_t

```
struct journal_s
{
unsigned long           j_flags;
int                     j_errno;
/* The superblock buffer */
struct buffer_head *    j_sb_buffer;
journal_superblock_t *  j_superblock;
int                     j_format_version;   /* Version of the superblock format */
int                     j_barrier_count;
struct semaphore        j_barrier;
transaction_t *         j_running_transaction;  /* Transactions: The current running transaction... */
transaction_t *         j_committing_transaction;  /* the transaction we are pushing to disk  */
transaction_t *         j_checkpoint_transactions;  /* all transactions waiting for checkpointing. */
wait_queue_head_t       j_wait_transaction_locked;
wait_queue_head_t       j_wait_logspace;  /* Wait queue for waiting for checkpointing to complete */
wait_queue_head_t       j_wait_done_commit; /* Wait queue for waiting for commit to complete */
```

journal_t

```
wait_queue_head_t       j_wait_checkpoint;  /* wait queue waiting for checkpointing  */
wait_queue_head_t       j_wait_commit;
wait_queue_head_t       j_wait_updates; /* Wait queue to wait for updates to complete */
struct semaphore        j_checkpoint_sem;
struct semaphore        j_sem;
unsigned long           j_head;         /* first unused block*/
unsigned long           j_tail;          /* oldest still-used block */
unsigned long           j_free;
unsigned long           j_first, j_last;        /* Journal start and end */
kdev_t                  j_dev;
int                     j_blocksize;
unsigned int            j_blk_offset;
kdev_t                  j_fs_dev;
unsigned int            j_maxlen;
struct inode *          j_inode;
tid_t                   j_tail_sequence;
```

journal_t

```
tid_t           j_transaction_sequence;
tid_t           j_commit_sequence; /* most recently committed transaction */
tid_t           j_commit_request;   /* most recently transaction wanting commit */
__u8            j_uuid[16];
struct task_struct *        j_task; /* Pointer to the current commit thread for this journal */
int                         j_max_transaction_buffers;
unsigned long               j_commit_interval;
/* The timer used to wakeup the commit thread: */
struct timer_list *         j_commit_timer;
int                         j_commit_timer_active;
struct list_head            j_all_journals;
struct jbd_revoke_table_s *  j_revoke; /* The revoke table: maintains the list of revoked blocks in the current
                                          transaction. */
};
```

Journal_head (journal-head.h)

struct journal_head {
Struct buffer_head * b_bh;
Int b_jcount;
unsigned b_jlist; /* Journaling list for this buffer */
char * b_frozen_data; /* Copy of the buffer data frozen for writing to the log. */
char * b_committed_data; // avoid overwriting uncommitted deletes
transaction_t * b_transaction; // transaction which owns this buffer's metadata
transaction_t * b_next_transaction;
struct journal_head *b_tnext, *b_tprev; //buffers on a transaction's data,metadata or forget queue.
transaction_t * b_cp_transaction; // which transaction to checkpoint this
struct journal_head *b_cpnext, *b_cpprev; /*still remaining to be flushed before an old transaction can be checkpointed*/*

};

start commit – kjournald()

```
int kjournald(void *arg)
{
        journal_t *journal = (journal_t *) arg;
        transaction_t *transaction;
        struct timer_list timer;

        current_journal = journal;

        lock_kernel();
        daemonize();  //作一些kernel  thread 所需要動作
        spin_lock_irq(&current->sigmask_lock);
        sigfillset(&current->blocked);
        recalc_sigpending(current);
        spin_unlock_irq(&current->sigmask_lock);

        sprintf(current->comm, "kjournald");

       /* Set up an interval timer which can be used to trigger a
         commit wakeup after the commit interval expires */
        init_timer(&timer);
```

kjournald()

```
        timer.data = (unsigned long) current;
        timer.function = commit_timeout;
        journal->j_commit_timer = &timer;

        /* Record that the journal thread is running */
        journal->j_task = current;
        wake_up(&journal->j_wait_done_commit);

        printk(KERN_INFO "kjournald starting.  Commit interval %ld seconds\n",
                                journal->j_commit_interval / HZ);
        list_add(&journal->j_all_journals, &all_journals);

        /* And now, wait forever for commit wakeup events. */
while (1) {
                if (journal->j_flags & JFS_UNMOUNT)
                        break;

                jbd_debug(1, "commit_sequence=%d, commit_request=%d\n",
                        journal->j_commit_sequence, journal->j_commit_request);
```

kjournald()

```
                if (journal->j_commit_sequence != journal->j_commit_request) {
                                jbd_debug(1, "OK, requests differ\n");
                                if (journal->j_commit_timer_active) {
                                                journal->j_commit_timer_active = 0;
                                                del_timer(journal->j_commit_timer);
                                }

                                journal_commit_transaction(journal);
                                continue;
                }
                wake_up(&journal->j_wait_done_commit);
                interruptible_sleep_on(&journal->j_wait_commit);

                jbd_debug(1, "kjournald wakes\n");

                /* Were we woken up by a commit wakeup event? */
                if ((transaction = journal->j_running_transaction) != NULL &&
                    time_after_eq(jiffies, transaction->t_expires)) {
                                journal->j_commit_request = transaction->t_tid;
                                jbd_debug(1, "woke because of timeout\n");
                }
}       //end while
```

kjournald()

```
if (journal->j_commit_timer_active) {
                journal->j_commit_timer_active = 0;
                del_timer_sync(journal->j_commit_timer);
        }

        list_del(&journal->j_all_journals);

        journal->j_task = NULL;
        wake_up(&journal->j_wait_done_commit);
        jbd_debug(1, "Journal thread exiting.\n");
        return 0;
}         //end kjouranld
```

journal_commit_transaction()

```
void journal_commit_transaction(journal_t *journal)
{
   ……
//首先 lock the current transaction and wait for all updates to complete
lock_journal(journal);        /* Protect journal->j_running_transaction */
lock_kernel();                    // 用在 multi-processors
J_ASSERT (journal->j_running_transaction != NULL);
commit_transaction = journal->j_running_transaction;
J_ASSERT (commit_transaction->t_state == T_RUNNING);
commit_transaction->t_state = T_LOCKED;
        // wait for all updates to complete
while (commit_transaction->t_updates != 0) {
        unlock_journal(journal);
        sleep_on(&journal->j_wait_updates);
        lock_journal(journal);
        }
J_ASSERT (commit_transaction->t_outstanding_credits <=
                                        journal->j_max_transaction_buffers);
```

journal_commit_transaction()

Update a journal's superblock fields and write it to disk, waiting for the IO to complete.

```
/*  journal_flush ?, update superblock */
        if (journal->j_flags & JFS_FLUSHED) {
                jbd_debug(3, "super block updated\n");
                journal_update_superblock(journal, 1);
        } else {
                jbd_debug(3, "superblock not updated\n");
        }

//release some buffers to get more mem space, reserved_list and already checkpointed buffers
       while (commit_transaction->t_reserved_list) {
                jh = commit_transaction->t_reserved_list;
                JBUFFER_TRACE(jh, "reserved, unused: refile");
                journal_refile_buffer(jh);
        }
       spin_lock(&journal_datalist_lock);
        __journal_clean_checkpoint_list(journal);
        spin_unlock(&journal_datalist_lock);
```

Find all the written-back checkpoint buffers in the journal and release them

journal_commit_transaction()

Write revoke records to the journal for all entries in the current revoke hash, deleting the entries as we go

```
//   commit phase 1
/* force the revoke list out to disk.*/
journal_write_revoke_records(journal, commit_transaction);
//after above, these can be reused by a new running trasaction , and we can safely start committing
        commit_transaction->t_state = T_FLUSH;
         wake_up(&journal->j_wait_transaction_locked);      //wait for a locked transaction to start committing
         journal->j_committing_transaction = commit_transaction;
         journal->j_running_transaction = NULL;
         commit_transaction->t_log_start = journal->j_head;
         unlock_kernel();

//    commit phase 2  -- start flush things to disk  : sync data buffers , and async data buffers write_out_data:
write_out_data:
         spin_lock(&journal_datalist_lock);
write_out_data_locked:                              // write data buffers first
         bufs = 0;
         next_jh = commit_transaction->t_sync_datalist;
         if (next_jh == NULL)
                  goto sync_datalist_empty;
         last_jh = next_jh->b_tprev;
```

journal_commit_transaction()

```
do {
        struct buffer_head *bh;

        jh = next_jh;
        next_jh = jh->b_tnext;
        bh = jh2bh(jh);
        if (!buffer_locked(bh)) {
                if (buffer_dirty(bh)) {                 //if dirty , wait to write ,else remove
                        BUFFER_TRACE(bh, "start journal writeout");
                        atomic_inc(&bh->b_count);
                        wbuf[bufs++] = bh;
                } else {
                        BUFFER_TRACE(bh, "writeout complete: unfile");
                        __journal_unfile_buffer(jh);
                        jh->b_transaction = NULL;
                        __journal_remove_journal_head(bh);
                        refile_buffer(bh);
                        __brelse(bh);
                }
        }
```

journal_commit_transaction()

```
                if (bufs == ARRAY_SIZE(wbuf)) {             // wbuf[64]
                J_ASSERT(commit_transaction->t_sync_datalist != 0);
                commit_transaction->t_sync_datalist = jh;
                                break;
                }
} while (jh != last_jh);
if (bufs || current->need_resched) {
                jbd_debug(2, "submit %d writes\n", bufs);
                spin_unlock(&journal_datalist_lock);
                unlock_journal(journal);
                if (bufs)
                                ll_rw_block(WRITE, bufs, wbuf);
                if (current->need_resched)
                                schedule();
                journal_brelse_array(wbuf, bufs);
                lock_journal(journal);
                spin_lock(&journal_datalist_lock);
                if (bufs)
                                goto write_out_data_locked;
        }
```

journal_commit_transaction()

```
// wait for previous I/O to complete
jh = commit_transaction->t_sync_datalist;
        if (jh == NULL)
                goto sync_datalist_empty;

        do {
                struct buffer_head *bh;
                jh = jh->b_tprev;                       /* Wait on the last written */
                bh = jh2bh(jh);
                if (buffer_locked(bh)) {
                                spin_unlock(&journal_datalist_lock);
                                unlock_journal(journal);
                                wait_on_buffer(bh);
                                /* the journal_head may have been removed now */
                                lock_journal(journal);
                                goto write_out_data;
                } else if (buffer_dirty(bh)) {
                                goto write_out_data_locked;
                }
        } while (jh != commit_transaction->t_sync_datalist);
        goto write_out_data_locked;
```

journal_commit_transaction()

```
sync_datalist_empty:
// write async buffers and wait it to complete
while ((jh = commit_transaction->t_async_datalist)) {
        struct buffer_head *bh = jh2bh(jh);
        if (buffer_locked(bh)) {
                spin_unlock(&journal_datalist_lock);
                unlock_journal(journal);
                wait_on_buffer(bh);
                lock_journal(journal);
                spin_lock(&journal_datalist_lock);
                continue;       /* List may have changed */
        }
        if (jh->b_next_transaction) {
                // later transaction may want the buffer for "metadata"
                __journal_refile_buffer(jh);
```

journal_commit_transaction()

```
} else {
                        BUFFER_TRACE(bh, "finished async writeout: unfile");
                        __journal_unfile_buffer(jh);
                        jh->b_transaction = NULL;
                        __journal_remove_journal_head(bh);
                        BUFFER_TRACE(bh, "finished async writeout: refile");
                        __brelse(bh);
                }
        }  // end while
spin_unlock(&journal_datalist_lock);

// commit phase 3 --  write metadata buffers to log
        commit_transaction->t_state = T_COMMIT;
        descriptor = 0;
        bufs = 0;
```

journal_commit_transaction()

```
while (commit_transaction->t_buffers) {
                ***jh = commit_transaction->t_buffers;***
                /* If we're in abort mode, we just un-journal the buffer and
                   release it for background writing. */
        if (is_journal_aborted(journal)) {    // if abort , just skip these write actions
                                JBUFFER_TRACE(jh, "journal is aborting: refile");
                                journal_refile_buffer(jh);
                                if (!commit_transaction->t_buffers)
                                        goto start_journal_io;
                                continue;
        }
        if (!descriptor) {                              // to get a descriptor for record metadata
                                struct buffer_head *bh;

                                J_ASSERT (bufs == 0);

                                jbd_debug(4, "JBD: get descriptor\n");
```

journal_commit_transaction()

```
                    descriptor = journal_get_descriptor_buffer(journal);
                    if (!descriptor) {
                                __journal_abort_hard(journal);
                                continue;
                    }

                    bh = jh2bh(descriptor);
                    jbd_debug(4, "JBD: got buffer %ld (%p)\n",
                                bh->b_blocknr, bh->b_data);
                    header = (journal_header_t *)&bh->b_data[0];
                    header->h_magic     = htonl(JFS_MAGIC_NUMBER);
                    header->h_blocktype = htonl(JFS_DESCRIPTOR_BLOCK);
                    header->h_sequence  = htonl(commit_transaction->t_tid);

                    tagp = &bh->b_data[sizeof(journal_header_t)];
                    space_left = bh->b_size - sizeof(journal_header_t);
                    first_tag = 1;
                    set_bit(BH_JWrite, &bh->b_state);
                    wbuf[bufs++] = bh;

                    /* Record it so that we can wait for IO  completion later */
                    BUFFER_TRACE(bh, "ph3: file as descriptor");
                    journal_file_buffer(descriptor, commit_transaction, BJ_LogCtl);
}   // end get a descriptor
```

journal_commit_transaction()

```
        /* Where the buffer is to be written  */
                blocknr = journal_next_log_block(journal);
…….
// write a metadata buffer to journal
// a new buffer head is constructed (t_iobuf_list) for I/O
// and original put to t_shadow_list
flags = journal_write_metadata_buffer(commit_transaction,   jh, &new_jh, blocknr);
set_bit(BH_JWrite, &jh2bh(new_jh)->b_state);
wbuf[bufs++] = jh2bh(new_jh);         // add new buffer to write to disk
……..
if (bufs == ARRAY_SIZE(wbuf) ||
    commit_transaction->t_buffers == NULL || space_left < sizeof(journal_block_tag_t) + 16){
                for (i=0; i<bufs; i++) {
                                struct buffer_head *bh = wbuf[i];
                                set_bit(BH_Lock, &bh->b_state);
                                clear_bit(BH_Dirty, &bh->b_state);
                                bh->b_end_io = journal_end_buffer_io_sync;
                                submit_bh(WRITE, bh);   // wait later I/O to complete
                }
```

journal_commit_transaction()

```
if (current->need_resched)
                    schedule();
        lock_journal(journal);
        /* Force a new descriptor to be generated next  time round the loop. */
        descriptor = NULL;
        bufs = 0;
        } // end if
}// end while


// now wait I/O(control buffer and metadata buffer) to commit a transaction
//  commit phase 4 – wait I/O and then release them
```

journal_commit_transaction()

```
wait_for_iobuf:  //metadata buffer
      while (commit_transaction->t_iobuf_list != NULL) {
                  struct buffer_head *bh;
                  jh = commit_transaction->t_iobuf_list->b_tprev;
                  bh = jh2bh(jh);
                  if (buffer_locked(bh)) {
                                    unlock_journal(journal);
                                    wait_on_buffer(bh);
                                    lock_journal(journal);
                                    goto wait_for_iobuf;
                                          }
                  clear_bit(BH_JWrite, &jh2bh(jh)->b_state);
                  journal_unfile_buffer(jh);     // remove buffer from transaction
                  wake_up(&bh->b_wait);
                  __brelse(bh);
      }  //end while
```

journal_commit_transaction()

```
// commit phase 5 -- wait for the revoke record and descriptor record buffers
wait_for_ctlbuf:
        while (commit_transaction->t_log_list != NULL) {
                struct buffer_head *bh;

                jh = commit_transaction->t_log_list->b_tprev;
                bh = jh2bh(jh);
                if (buffer_locked(bh)) {
                                unlock_journal(journal);
                                wait_on_buffer(bh);
                                lock_journal(journal);
                                goto wait_for_ctlbuf;
                }
 // now descriptor writeout done
        }   // end while
```

journal_commit_transaction()

```
// commit phase 6  -- write commit record
…..
// commit record only need first 512 bytes
        for (i = 0; i < jh2bh(descriptor)->b_size; i += 512) {
                journal_header_t *tmp = (journal_header_t*)jh2bh(descriptor)->b_data;
                tmp->h_magic = htonl(JFS_MAGIC_NUMBER);
                tmp->h_blocktype = htonl(JFS_COMMIT_BLOCK);
                tmp->h_sequence = htonl(commit_transaction->t_tid);
        }
        JBUFFER_TRACE(descriptor, "write commit block");
        {
                struct buffer_head *bh = jh2bh(descriptor);
                ll_rw_block(WRITE, 1, &bh);
                wait_on_buffer(bh);
                __brelse(bh);                   /* One for getblk() */
                journal_unlock_journal_head(descriptor);
        }
…..
```

journal_commit_transaction()

```
// commit phase 7 – handle checkpoint list
while (commit_transaction->t_forget) {
        transaction_t *cp_transaction;
        struct buffer_head *bh;

        jh = commit_transaction->t_forget;
        J_ASSERT_JH(jh,jh->b_transaction == commit_transaction ||
                jh->b_transaction == journal->j_running_transaction);

        if (jh->b_committed_data) {
                kfree(jh->b_committed_data);
                jh->b_committed_data = NULL;
                if (jh->b_frozen_data) {
                        jh->b_committed_data = jh->b_frozen_data;
                        jh->b_frozen_data = NULL;
                }
        } else if (jh->b_frozen_data) {
                kfree(jh->b_frozen_data);
                jh->b_frozen_data = NULL;
        }

}
```

journal_commit_transaction()

```
spin_lock(&journal_datalist_lock);
cp_transaction = jh->b_cp_transaction;
if (cp_transaction) {
                JBUFFER_TRACE(jh, "remove from old cp transaction");
                J_ASSERT_JH(jh, commit_transaction != cp_transaction);
                __journal_remove_checkpoint(jh);
}

bh = jh2bh(jh);
if (buffer_jdirty(bh)) {
                JBUFFER_TRACE(jh, "add to new checkpointing trans");
                __journal_insert_checkpoint(jh, commit_transaction);
                JBUFFER_TRACE(jh, "refile for checkpoint writeback");
                __journal_refile_buffer(jh);
} else {
                J_ASSERT_BH(bh, !buffer_dirty(bh));
                J_ASSERT_JH(jh, jh->b_next_transaction == NULL);
                __journal_unfile_buffer(jh);
                jh->b_transaction = 0;
                __journal_remove_journal_head(bh);
                __brelse(bh);
}
spin_unlock(&journal_datalist_lock);
```

journal_commit_transaction()

```
// commit phase 8 – all done
……
commit_transaction->t_state = T_FINISHED;
unlock_journal(journal);
        wake_up(&journal->j_wait_done_commit);
} // end journal_commit_transaction
```

log_do_checkpoint()

```
int log_do_checkpoint (journal_t *journal, int nblocks)
{
struct buffer_head *bhs[NR_BATCH];              // 64
……
repeat:
        transaction = journal->j_checkpoint_transactions;
        if (transaction == NULL)
                goto done;
        last_transaction = transaction->t_cpprev;
        next_transaction = transaction;

        do {
                struct journal_head *jh, *last_jh, *next_jh;
                int drop_count = 0;
                int cleanup_ret, retry = 0;

                transaction = next_transaction;
                next_transaction = transaction->t_cpnext;
                jh = transaction->t_checkpoint_list;
                last_jh = jh->b_cpprev;
                next_jh = jh;
                do {
                                jh = next_jh;
                                next_jh = jh->b_cpnext;
                                retry = __flush_buffer(journal, jh, bhs, &batch_count,&drop_count);
                } while (jh != last_jh && !retry);
```

Try to flush one buffer from the checkpoint list to disk.
Return 1 if something happened which requires us to abort the current scan of the checkpoint list.

log_do_checkpoint()

```
                if (batch_count) {
                        __flush_batch(bhs, &batch_count);
                        goto repeat;
                }
                if (retry)
                        goto repeat;
                /*
                 * We have walked the whole transaction list without
                 * finding anything to write to disk.  We had better be
                 * able to make some progress or we are in trouble.
                 */
                cleanup_ret = __cleanup_transaction(journal, transaction);
                J_ASSERT(drop_count != 0 || cleanup_ret != 0);
                goto repeat;  /* __cleanup may have dropped lock */
        } while (transaction != last_transaction);

.......
}
```

Clean up a transaction's checkpoint list.

journal_recover()

```
int journal_recover(journal_t *journal)
{
        int                                             err;
        journal_superblock_t *          sb;

        struct recovery_info            info;

        memset(&info, 0, sizeof(info));
        sb = journal->j_superblock;

        /*
         * The journal superblock's s_start field (the current log head)
         * is always zero if, and only if, the journal was cleanly
         * unmounted.
         */

        if (!sb->s_start) {
                jbd_debug(1, "No recovery required, last transaction %d\n",
                                ntohl(sb->s_sequence));
                journal->j_transaction_sequence = ntohl(sb->s_sequence) + 1;
                return 0;
        }
```

journal_recover()

```
        err = do_one_pass(journal, &info, PASS_SCAN);
        if (!err)
                err = do_one_pass(journal, &info, PASS_REVOKE);
        if (!err)
                err = do_one_pass(journal, &info, PASS_REPLAY);

        jbd_debug(0, "JBD: recovery, exit status %d, "
                  "recovered transactions %u to %u\n",
                  err, info.start_transaction, info.end_transaction);
        jbd_debug(0, "JBD: Replayed %d and revoked %d/%d blocks\n",
                  info.nr_replays, info.nr_revoke_hits, info.nr_revokes);

        /* Restart the log at the next transaction ID, thus invalidating
         * any existing commit records in the log. */
        journal->j_transaction_sequence = ++info.end_transaction;

        journal_clear_revoke(journal);
        return err;
}
```

clear the revoke table so that it can be reused by the running filesystem.

Reference

- Source code – 2.4.7-10
- http://www.linux-mag.com/2000-08/journaling_01.html
- Paper: Journaling the Linux ext2fs Filesystem

  -- Setphen C. Tewwdie