Overview

Interaction with the log manager

A few specifics on algorithms and data structures

In-core log

In-core log details

Other interfaces

On-disk log representation

Log record header

Log operation header

Not yet implemented

this page left intentionally blank

Interaction with the log manager

1. Request a reservation
   - ubiquitous mount structure
   - transaction identifier (uuid?-- returned from log manager?)
   - length of data in bytes (assert on long alignment?)
   - ticket pointer (returned from log manager)
   - client identifier (currently only transaction manager supported)
   - flags
     - sleep (not yet implemented)
     - no sleep
     - permanent reservation (not yet implemented-more later)
2. Possibly multiple writes to log manager using the ticket provided
   - mount structure
   - array of region pointers
   - number of region pointers
     - address
     - length in bytes
     - log sequence number of specific region (more later)
   - ticket
3. Finish with reservation and commit transaction
   - mount structure
   - ticket
4. Possibly multiple callback notifications to be called after data is on disk
   - mount structure
   - log sequence number
   - callback function
   - callback argument

A few specifics on algorithms & data structures

- Reservations

  Long running transactions use:

  - permanent reservations
  - ask for two times what they need for one transaction

- Tickets
  - single threaded data structures
  - allocated in page size chunks and never deallocated
  - stored on a linked list, hung off the main log structure
  - contain:
    - transaction identifier
    - reservation in bytes
    - client identifier

- Main log structure hung off mount structure

```
typedef struct log {
   log_ticket_t  *l_freelist;    /* free list of tickets */
   in_core_log_t *l_iclog;       /* 1st copy of in-core log */
   in_core_log_t *l_iclog2;      /* 2nd copy of in-core log */
   in_core_log_t *l_iclog_curr;  /* current active log */
   sema_t        l_iclogsema;    /* grab to change iclog state */
   dev_t         l_dev;          /* dev_t of log */
   int           l_logsize;      /* total size in bytes of log */
   int           l_cycle;        /* Cycle number of log writes */
   int           l_currblock;    /* current logical block of log */
   int           l_logreserved;  /* log space currently reserved */
   xfs_lsn_t     l_sync_lsn;     /* lsn of last LR w/ buffers committed */
} log_t;
```

In-core log

Why?

Used to help performance of log manager and buffer cache.

How?

Batch multiple log operations together into larger writes to disk

- decreases the number of writes going to disk.
- decreases the time buffers stay pinned in memory.

```
typedef struct in_core_log {
     union {
          log_rec_header_t      sic_header;
          char                  sic_sector[LOG_HEADER_SIZE];
     } ic_s;
     char       ic_data[LOG_RECORD_SIZE-LOG_HEADER_SIZE];
     xfs_lsn_t  ic_current_lsn;  /* lsn for this log record */
     int        ic_size;         /* total size of data buffer */
     uint       ic_offset;       /* start to begin writing data */
     char       ic_state;        /* state of this IC log */
     char       ic_refcnt;       /* reference count */
     buf_t      *ic_bp;          /* private buffer pointer */
} in_core_log_t;


#define ic_header ic_s.sic_header
```

still need to union ic_data with a log_op_header_t

In-core log details

- Two complete in-core structures
  - while one is syncing to disk, writes can occur to the other in-core log.
  - need some type of simple state machine which is MP-safe.
- Multiple log writes can be happening at the same time to the same in-core log.
  - use offset to determine where next write can start.
  - use refcnt to remember how many writes are happening concurrently.
- The following structures/fields will be under state machine control:
  - states of both in-core logs
  - offsets of both in-core logs
  - reference counts of both in-core logs
  - current active in-core log
  - callback functions of both in-core logs
- Need to think more about the case where both logs are syncing to disk and a new write comes in.

Other interfaces

Force log manager to write in-core log to disk

- perform synchronously
- force it down now
- force it down within some timeout (not yet implemented)

Assign new transaction to a given reservation

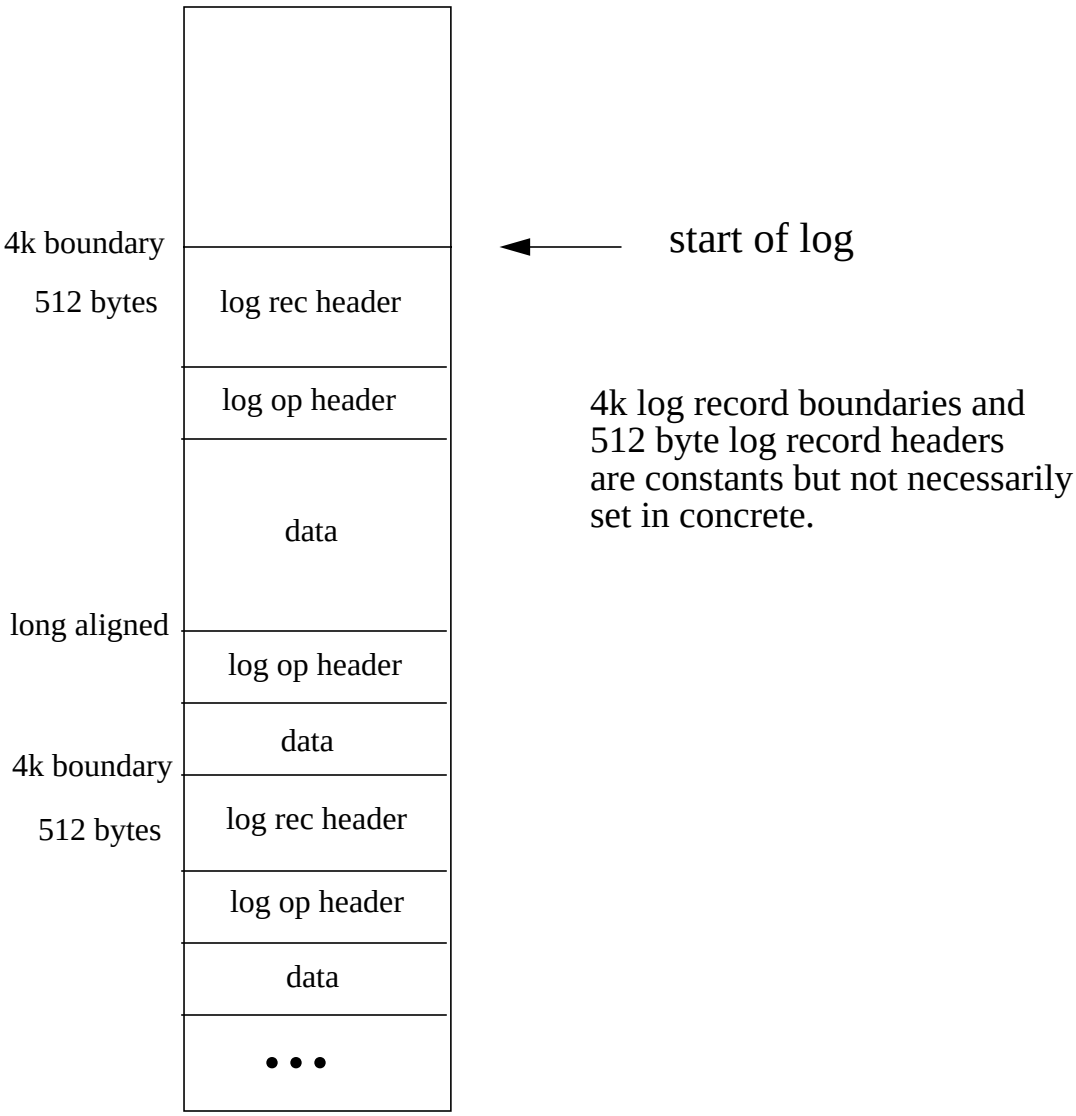
- mount structure
- ticket
- old transaction identifier
- new transaction identifier (uuid?--from log manager?)

Mount log filesystem

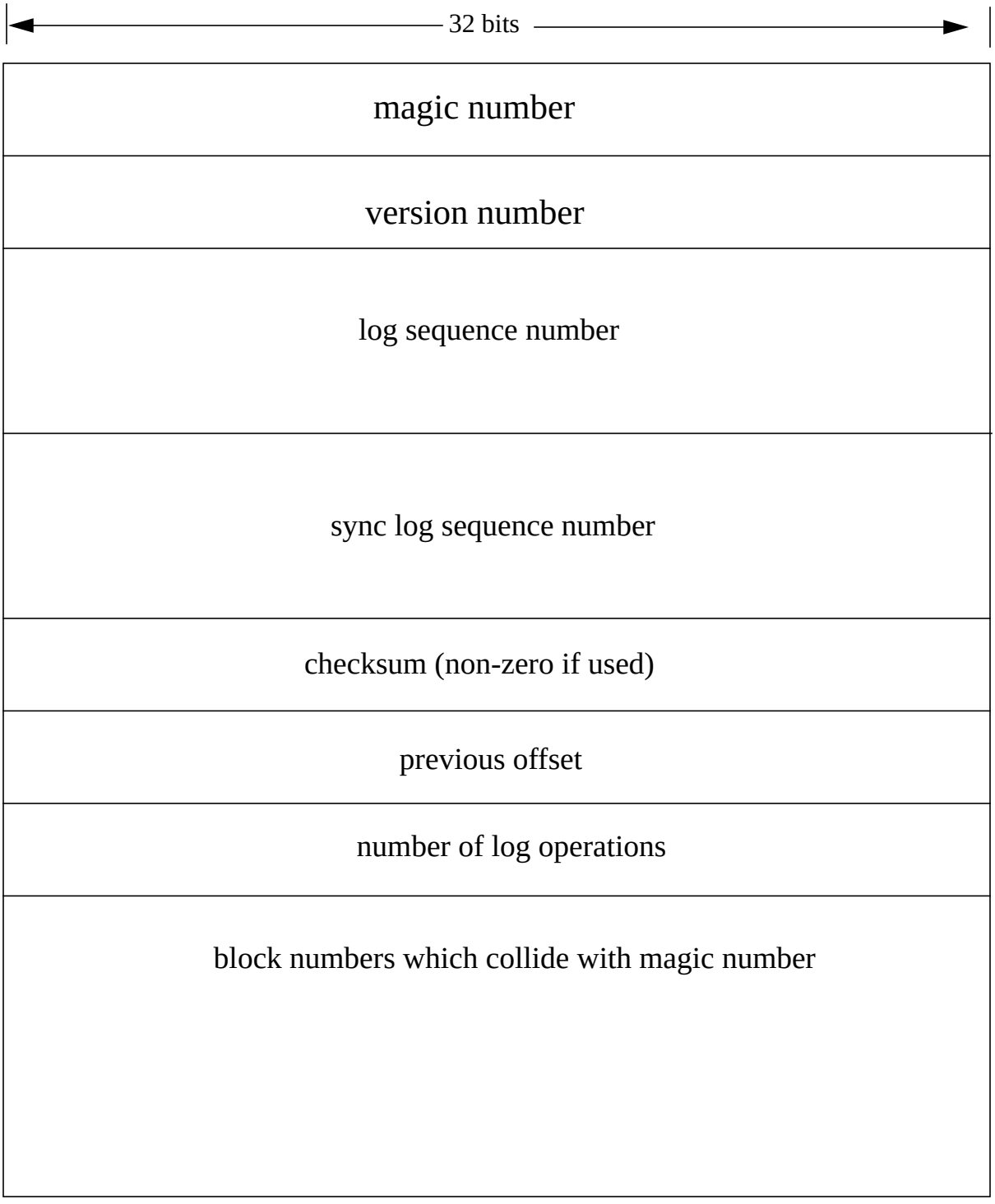
- mount structure
- log device number
- flags (none defined yet)

Initialize log filesystem - does nothing now

On-disk representation

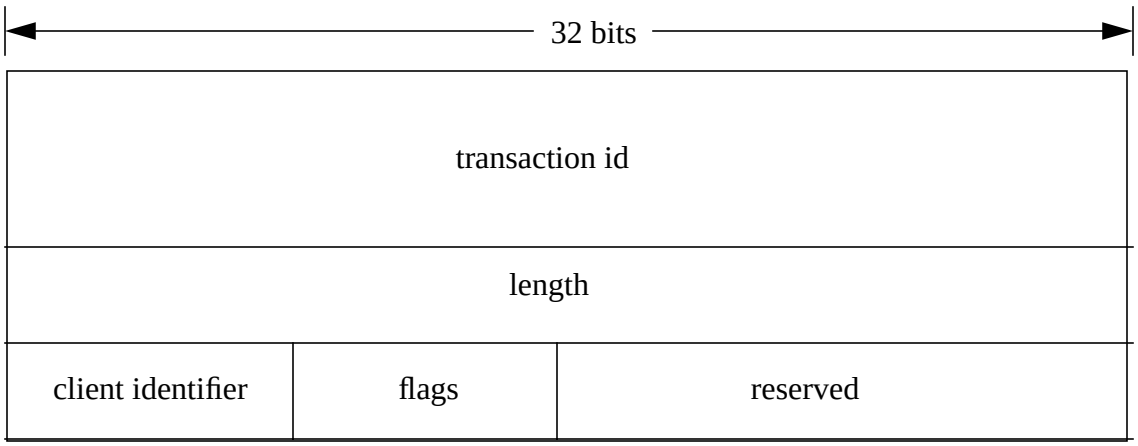


Log record header

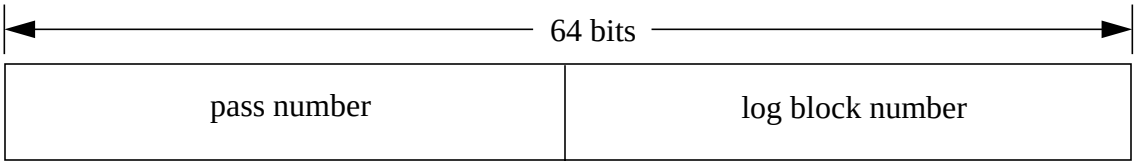


Reservations are internally increased by the size of a log record header (512 bytes)

Log operation header



Log Sequence Number



Log sequence numbers are used at recovery time to find the end of the log.

The log block number of the start of the log record is embedded in the lsn, so the tail of the log can be tracked more easily.

Not yet implemented

Recovery code

Finding the end of the on-disk log (algorithm described in design doc)

Complete in-core log state machine (including callback functions)

Rolling log/tail management