The Log Manager (xLM)

Michael Nishimoto

1.0 Introduction

The purpose of a log manager is to provide a service which allows fast and reliable reconstruction of a filesystem after a crash. It will also provide higher performance for some operations, like updates to metadata. In order to accomplish this, other services log changes to filesystem metadata (inodes, directories, free-space pool). The log manager groups multiple change requests into single synchronous writes to its own space on the disk. Once logged, the clients of the log manager do not need to immediate flush their dirty buffers. They assume the action will be performed such that filesystem consistency will be maintained in the event of a crash. Changes to user data will not be logged since losing it would not compromise filesystem integrity.

2.0 Log Representation

The log space is split into two parts, the **on-disk log** and the **in-core log**. When an operation needs to be logged, it first gets handed to the in-core log. This log is nothing more than a FIFO queue which holds log operations until some event causes the queue to get flushed to the on-disk log. Operations stored in the in-core log are known as **active operations**.

The volume manager provides a preallocated linear address space (0-n blocks) for the on-disk log, which is kept separate from the data space. From a high-level view, the on-disk log appears to be a circular queue, which wraps when hitting the end. Operations which are stored in the on-disk log are known as **committed operations**.

When a sequence of actions or operations need to be bundled and committed as one, we call this a **transaction** (See “Transaction Layer” on page 10.)

3.0 Log Records

In order to reduce write activity, active operations are grouped together into **log records** and then written to disk. While still in-core, these records are known as **active**. Once on disk, they are known as **committed** log records. Three events may happen which cause the in-core queue to be synced to disk.

- The in-core queue is near capacity.
- An operation comes into xLM which requests an immediate flush to disk.
- Some internal timeout occurs.

3.1 Contents of Log Records

Log records are split into two parts, **log record header** and **log record data**. Log record headers contain information common to the entire log record, whereas log record data contains concatenated log operations.

3.1.1 Specifics of a log record header

- Log sequence number of this log record
- Log sequence number of the last fully synced log record
- Length of log body in 64-bit word lengths
- Magic number in first word of header. If a (sector, block) begins with a specific magic number, it will (by definition) be a log record header. (see 4.1.2 & 5.1)
- Blocks in log body which have had their first word overwritten (see 4.1.2)
- Possibly -- number of log operations in log body
- Header is 512 bytes in length -- one disk sector

3.1.2 Specifics of log record data

The log operations contained in log record data are split into two parts, **log operation header** and **log operation data**. Log operation data is a stream of post-image double-word aligned bytes; a post-image is a snapshot of meta-data after a modification has been made. Note that this implies a time ordering of writes. Buffer metadata can only be synced to disk after its associated log record has been written to disk. Log operation headers contain:

- Who asked xLM to log the operation.
- Length of log operation data in bytes
- Transaction ID
- Type of operation (if we choose to have operation logging)

3.2 Log Sequence Numbers

Log sequence numbers are 64 bits in length. The lower 32 bits are a log block number, while the upper 32 bits make up a number which starts with one and increments each time the on-disk log wraps. A **log block number** is the logical number of the block (0-n). Assuming 512 byte blocks, this would limit the size of the on-disk log to 2 terabytes. If we were to assume the on-disk log could wrap once a second, which is abnormally high, the upper 32 bits would wrap in about 136 years. Consequently, it appears that the chosen representation for a log sequence number should be quite adequate for the life-time of xLM.

3.3 On-disk Representation

Each log record is preceeded with an informational **header** (size not to exceed one sector) which contains:

- A log record magic number
- A log sequence number
- The length of the log record
- Offset to previous log record header or (-1) if none
- The number of log operations (maybe)
- A boolean, stating whether this log record has been check-summed, and possibly a checksum.

It has not been determined whether log records will be of equal length.

3.4 Synchronization Records

Buffer synchronization information will be kept in the log; so upon recovery, we know approximately what metadata has been written to disk. This is only an optimization mechanism which will help speed recovery. Without this information, the entire on-disk log would need to be replayed.

Each log record header will contain the log sequence number of the last log record to have all its associated metadata buffers written to disk. Past log managers have logged sync records which are a special log record stating the same information as above.These would have been placed in the log at regular intervals. We have chosen to put the buffer sync information in every log record header to make it easier to find and to allow the information to be kept more current. Synchronization information would need to include a log sequence number and the block address of where the log record for that LSN started.

3.5 In-core Log

The in-core log will consist of two kernel malloc’ed regions which are alternatively written. Each represents an entire log record. This will allow writes to the in-core log to continue while a log record is committed to disk.

4.0 Recovery Manager

After a crash, the recovery manager reads the on-disk log and replays it in some manner to reconstruct the filesystem. It will hand idempotent actions to other services without knowing the contents of the various operations.

4.1 Finding the end of the on-disk log

Since recovery must be fast, quickly finding the end of the on-disk log is a requirement. The last valid on-disk log record will be one where there is either

- No legal log header after this log record
- A checksum in this log header with a log body which corresponds to the checksum

4.1.1 The Tolerant way

Reserve two blocks A & B at the beginning of the log space for administration information. Writing alternatively into A and B, place information about the current end of the log *every once in a while*. When recovering from a crash, read the two blocks and take the one with the larger timestamp. Seek to the winning block location and start scanning forward, looking for the end of the log .

4.1.2 A less encumbered run-time model

In order to eliminate the additional seeking which is required in the Tolerant scheme, xLM will use a more intelligent algorithm when searching for the end of the log. This proposal is simple. Perform a binary search over the entire log space until the search segment is small enough to search linearly. Assuming a gigabyte log, it would take 11 random seeks to get down to a 0.5 MB linear scan. This time should be negligible during the bootup phase.

Once the binary search selects a given log block number, seek to this location and start scanning forward, looking for a valid log header. Log headers will always begin with some 32-bit length magic number on a block boundary. Since the log manager knows about the control and data being placed in the log, it will replace any magic number with a zero if the number would have fallen on a block start boundary. In the log header, a note will be made about the replacement.

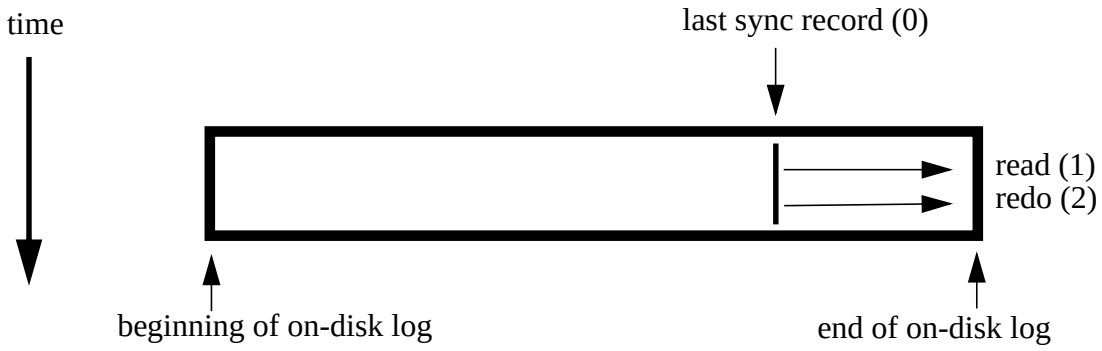
4.1.3 The end of the log

Version one will not perform checksumming on the log header. Instead, it assumes there must be a following valid log header. In order to insure a synchronous write has occurred, xLM will need to guarantee that a next log record has been committed. As such, xLM will need to begin a following log record write soon after completing the original commit.

If the original commit was marked urgent by some request to xLM, the validating log record write is started immediately, as soon as the current in-core log is consistent. This may mean an empty log record is committed to disk. If the original commit was not urgent, xLM may set a timer to determine the maximum amount of time to pass before the next commit takes place. Cascading urgent requests may have the behavior that many small log record commits are done continuously. However, any empty log record commit does not need to be validated.

4.2 Reading the on-disk log after a crash

After finding the last sync record (see 4.1-action 0), read forward looking for the end of the OD log (action 1). While reading, find those transactions and operations which are bounded by bookends. Complete operations are non-transactional actions which are bounded by bookends. Transactions are made up of multiple operations and need to have a commit record. Now, redo or replay each of the complete operations and committed transactions in order to the end of the log (action 2). ;’



5.0 Other Algorithms

5.1 Initializing the log space

Because log record headers have a magic number in the first word of each block, xLM must know that this number doesn’t appear accidentally at startup. Although this operation may take quite a while, the initialization is important and can be performed by the xFS mkfs program. The minimum guarantee is that the first sector of each block be zeroed.

5.2 How to grow the log space

Growing the log space will not be in the first version of xLM; however, here are some ideas of what may happen from the perspective of the log manager.

xLM will ask xLV to grow its log space. If the log space can be grown in place, xlv will return notice of this fact. xLM will grow the log in place, extending past the end of its previous last block. This leaves xlv with the alternative of carving out another section of the drive, so the log space can grow. Since xlv provides a linear log address space, xLM will not be affected when the physical sectors representing the log are not contiguous.

However, if xlv decides that the log must move, the log manager will accommodate. Another thread can start copying the current log space to the new location while writes continue to the current log. At some point when the current log space is completely copied, the log manager can just start writing to the new log location and notify the volume manager that the old log space is no longer needed.

5.3 Block change logging

This type of logging is required to help speed recovery of plexed filesystems after a crash. The filesystem keeps track of which data blocks are being written to disk and which ones have actually made it. During recovery, it analyzes a block change log to determine which blocks need to be copied to restore plex consistency. See Volume Manager for more details.

As far as the log manager is concerned, it will get a call from the buffer cache to log some data block locations immediately before the blocks are synced to disk. As such, the buffer cache must keep track of buffers which represent plex writes. The filesystem does not want to log all data block changes if they do not need to be. Once data blocks are written to a plex disk, xVM will call the log manager to notify it which plex blocks have been synced.

Re-iterating, it is **not** the job of xLM to determine which data blocks need to be logged.

Assumption: Because we log data block changes immediately before a buffer gets synced to disk, the log can ignore these records when wrapping. The log should be long enough to guarantee that writes have succeeded. If we logged a block change operation when the buffer is written, the log would need to guarantee the buffer was written out before clobbering the entry.

After a crash, xVM will call xLM to ask about the data blocks which did not make it to disk.

6.0 Registration of Log Manager Clients

Registration is a procedure where virtual processes (other services) make themselves known to the log manager at initialization time. The provided information will allow the log manager to call back these services during disaster recovery when the on-disk log is begin played back. Registration can be viewed as a log operation that informs xLM of details which will not be conveyed on regular operations.

In the first version, all registration will be done statically through some predefined table. As an enhancement to this scheme, a dynamic registration mechanism will need to be developed to address the issues present with loadable modules and a distributed environment.

7.0 Log Manager Interaction with Buffer Cache

7.1 Wrapping the on-disk log

Wrapping the on-disk log is one of the more complex events with which the log manager must deal. While one thread may be writing log records to the head of the circular log buffer, another thread will be attempting to push out dirty buffers; so log records at the tail can be written over. Any good mechanism, which manages the log space, will try hard to disallow the head of the log writer to approach the tail of the log. Regardless of this effort, the log manager must guarantee that the head and tail never meet unless it is ready to abort actions which have already been committed to the log. Some algorithms for solving this problem are:

- Aborting the operation associated with the log record we are about to over-write.
- Reserving enough log space when a client attempts to grab a buffer for writing, so xLM can guarantee the operation will complete.
- Divide the log into sections and keep one full section between the head and tail. This scheme sets an upper bound on the number of active filesystem transactions.

8.0 Interfaces & Constants

8.1 Normal operation

```
/* Log Clients */
#define TRANSACTION_MANAGER 1
#define VOLUME_MANAGER 2

/*
*  1. Reserve an amount of on-disk log space and return a ticket corresponding to the
*     reservation.
*  2. Potentially, push buffers at tail of log to disk.
*/
errno_t xfs_log_reserve(
   (struct mountp *mp)    filesystem_id,
   (int)                  length,
   (int)                  log_client,
   (uuid_t)               tid,
   (uint)                 flags,
   (uint *)               ticket)

filesystem_id -
length       - approximate number of bytes a specific log operation will take
log_client   - registration number. A compiled table will allow mapping from # to
                    function to call during recovery.
tid          - transaction id
ticket       - identifier which gives permission to use a specific reservation during a
                    log write.
flags: XFS_LOG_RES_SLEEP   - If space is not available, sleep
       XFS_LOG_RES_NO_SLEEP- If space is not available, return error immediately
       XFS_LOG_RES_PERM_RESERV- Permanent reservation. Never deduct space on
                                xfs_log_write()s. This is used when a transaction
                                will be long-lived (deleting files) and determining a
                                complete reservation ahead of time may not be
                                feasible.

/*
*     Replace old transaction id with new transaction id for the reservation identified
* by ticket. This is like changing the name on an airplane ticke reservation.
*/
errno_t xfs_log_new_transaction(
   (uint)    ticket,
   (uuid_t)  old_tid,
   (uuid_t)  new_tid)


/*
*     Write the regions in a vector to the in-core log.
*/
```

```
errno_t xfs_log_write(
   (iovec [])       write_vector,
   (int)            nentries,
   (int)            ticket)

nentries     - length of write_vector; number of regions to write

struct iovec {
       caddr_t      b_addr;      /* beginning address of region */
       uint         length;      /* length in bytes of region */
       long long    lsn;         /* log sequence number of region */
}
lsn          - log sequence number for

NOTE:
Since the regions in one xfs_log_write() call may get written in multiple log records to
the on-disk log, the log manager will return the log sequence numbers for each region in
the iovec array.


/*
* Force the current in-core log records to disk immediately.
*/
errno_t xfs_log_force(
   (uint)    flags)

flags:
       XFS_LOG_SYNC  - Don't return until all log records are written to on-disk log.


/*
*  1. Release the remaining portion of a reservation.
*  2. Commit log operation => write out some end log operation marker.
*/
errno_t xfs_log_done(
   (uint)    ticket,
   (uint)    flags)

ticket -
flags:
       XFS_LOG_SYNC  - Don't return until log records with transaction id associated
                            with ticket are written to on-disk log. Has noimplication
                            of urgency to write. Without this flag, calls are
                            asynchronous
       XFS_LOG_FORCE - Start writing log record to on-disklog.
       XFS_LOG_URGE  - Start thinking about writing logrecord to on-disk log. This
                            will be implemented by using atimer which will start the
                            write within a guaranteed amount oftime. The assumption is
                            that performance can be gained in thesystem by not forcing
                            a write at this time.

/*
*  Call callback_func with callback_data as argument after log sequence number denoted
*      by lsn is forced to the permanent log.
*/
errno_t xfs_is_lsn_in_core(
   (long long)      lsn,
   (void *)         callback_func,
   (void *)         callback_data)
```

```
lsn           - log sequence number
callback_func - function to call after this reservation commits to the on-disk log.
callback_data - data pointer to be handed to callback_func when function is called.
```

8.2 Debugging mode

```
/*
*      Write the entire log space. This routine will be needed to debug the log
*   recovery code. Arbitrary byte sequences can be written to disk.
*/
errno_t xfs_log_write_all(
    (caddr_t) buf,
    (uint)    length)
```

8.3 Low level algorithms

8.3.1 Algorithm to keep track of tickets

Have static array of 512 or 1024 entries long. Each element is a link in a linked list. Have counter which starts at 0 and start incrementing each time a new ticket is handed out. Ticket 0 is assigned to slot 0. Ticket 1 is assigned to slot 1. Ticket 1025 is assigned to slot (1025 << 10) == 1. At 1024 new transactions a second on a specific filesystem, we roll the tickets in about 8 years of constant uptime. A free list of tickets should be kept. Each element needs to have:

- Ticket number (32 bits)
- Transaction id (64 bits)
- Reservation in bytes (32 bits)
- Client id (32 bits)
- Next pointer (32-64 bits)

9.0 Utilities

9.1 Log Utilities

Two log utilities need to be written in order to help build a reliable log manager. The first is simple, a log printer.

This program will read the on-disk log structures and print out the data in a nicely formatted style. It should have the ability to format the data at different complexity levels. High level views will be necessary for viewing large logs. A log printing program will want to be shipped with the system to facilitate debugging off-site filesystem crashes. (if there are any ;-)

Another required program is a log operation injector. This will provide a mechanism for placing arbitrary or even pathological log operations onto the disk. Crashing in the middle of a log sync operation could leave a portion of the log in an indeterministic state.

9.2 Requirements of other Utilities

The new mkfs must initialize the on-disk log to all zeroes (see 5.1).

10.0 Transaction Layer

Long running operations, like file truncation or removal, need to have transactional semantics. Therefore, xFS provides this functionality through a transaction mechanism, which is separate from xLM, the log manager. The mechanism is combined with buffer cache routines and exists as a layer on top of the log manager.

When a transaction needs to be performed, a client asks xTM for a transaction id. The id will be a uuid. Before returning to the client, xTM will log a transaction start to the log manager. The client uses the transaction id and sends its log operations directly to the log manager. When a transaction is completed, the client calls xTM, which sends a transaction complete log operation. Depending on the type of transactional operation, the final commit may block until the metadata buffers associated with the transaction are flushed to disk. Alternatively, we may need a call-back mechanism for clients who require notification of completion.

More details are provided in the chapter on the xFS Transaction Mechanism.