JFS Log

How the Journaled File System performs logging

Steve Best sbest@us.ibm.com
IBM Linux Technology Center

Note: This paper is to appear in the Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, Ga., October 2000

Abstract

This paper describes the logging done by the Journaled File System (JFS). By logging, JFS can restore the file system to a consistent state in a matter of seconds, versus minutes or hours with non-journaled file systems. This white paper gives an overview of the changes to meta-data structures that JFS logs.

Introduction

The Journaled File System (JFS) provides a log-based, byte-level file system that was developed for transaction-oriented, high performance systems. Scaleable and robust, one of the key features of the file system is logging. JFS, a recoverable file system, ensures that if the system fails during power outage, or system crash, no file system transactions will be left in an inconsistent state. The on-disk layout of JFS will remain consistent without the need to run fsck. JFS provides the extra level of stability with a small decrease in performance when meta-data changes need to be logged.

JFS uses a transaction-based logging technique to implement recoverability. This design ensures a full partition recovery within seconds for even large partition sizes. JFS limits its recovery process to the file system structures to ensure that at the very least the user will never lose a partition because of a corrupted file system. Note, that user data is not guaranteed to be fully updated if a system crash has occurred. JFS is not designed to log user data and therefore is able to keep all file operations to an optimal level of performance.

A general architecture and design overview of JFS is presented in [JFSOverview].

The development of a recoverable file system can be viewed as the next step in file system technology.

Recoverable File Systems

A recoverable file system, such as IBM's Journaled File System-(JFS), ensures partition consistency by using database-logging techniques originally developed for transaction processing. If the operating system crashes, JFS restores consistency by executing the logredo procedure that accesses information that has been stored in the JFS log file.

JFS incurs some performance costs for the reliability it provides. Every transaction that alters the on-disk layout structures requires that one record be written to the log file for each transaction's sub-operations. For each file operation that requires meta-data changes, JFS starts the logging transaction by calling the TxBegin routine. JFS ends the transaction's sub-operation by calling the TxEnd routine. Both TxBegin and TxEnd will be discussed in more detail in a later section.

Logging

JFS provides file system recoverability by the method of transaction processing technique called **logging**. The sub-operations of any transactions that change meta-data are recorded in a log file before they are committed to the disk. By using this technique, if the system crashes, partially completed transactions can be undone when the system is rebooted. A transaction is defined as an I/O operation that alters the on-disk layout structures of JFS. A completed list of operations that are logged by JFS will be discussed later. One example of an operation that JFS logs is the unlink of a file.

There are two main components of the JFS logging system: the **log file** itself and the **transaction manager**. The log file is a system file created by the mkfs.jfs format utility.

There are several JFS data structures that the transaction manager uses during logging and they will be defined next.

Extents, Inodes, Block Map

A "file" is allocated in sequences of extents. An **Extent** is a sequence of contiguous aggregate blocks allocated to a JFS object as a unit. An extent is wholly contained within a single aggregate (and therefore a single partition); however, large extents may span multiple allocation groups. An extent can range in size from 1 to 2(24) -1 aggregate blocks.

Every JFS object is represented by an inode. Inodes contain the expected object-specific information such as time stamps and file type (regular vs. directory, etc.).

The Block Allocation Map is used to track the allocated or freed disk blocks for an entire aggregate.

The Inode Allocation Map is a dynamic array of Inode Allocations Groups (IAGS). The IAG is the data for the Inode Allocation Map.

A more complete description of JFS' on-disk layout structures is presented in [JFSLayout].

Transaction Manager

The Transaction Manager provides the core functionality that JFS uses to do logging.

A brief explanation of the transaction flow follows:

A call to TxBegin allocates a transaction "block", tblk, which represents the entire transaction.

When meta-data pages are created, modified, or deleted, transaction "locks", tlck's, are allocated and attached to the tblk. There is a 1 to 1 correspondence between a tlck and a meta-data buffer (excepting that extra tlcks may be allocated if the original overflows). The buffers are marked 'nohomeok' to indicate that they shouldn't be written to disk yet.

In txCommit (), the tlck's are processed and log records are written (at least to the buffer). The affected inodes are "written" to the inode extent buffer (they are maintained in separate memory from the extent buffer). Then a commit record is written.

After the commit record has actually been written (I/O complete), the block map and inode map are updated as needed, and the tlck'ed meta-data pages are marked 'homeok'. Then the tlcks are released. Finally the tblk is released.

Example of creating a file

A brief explanation of the create transaction flow follows:

```
TxBegin(dip->i_ipmnt, &tid, 0);

tblk = &TxBlock[tid];

tblk->xflag |= COMMIT_CREATE;

tblk->ip = ip;

 /* Work is done to create file */

rc = txCommit(tid, 2, &iplist[0], 0);

TxEnd(tid);
```

File System operations logged by JFS

The following list of file system operations changes meta-data of the file system so they must be logged.

- File creation (create)
- Linking (link)
- Making directory (mkdir)
- Making node (mknod)
- Removing file (unlink)
- Rename (rename)
- Removing directory (rmdir)
- Symbolic link (symlink)
- Set ACL (setacl)
- Writing File (write) (not on normal conditions)
- Truncating regular file

Log File maximum size

The format utility mkfs.jfs creates the log file size based on the partition size.

The default of the log file is .4 of the aggregate size and this value is rounded up to a megabyte boundary. The maximum size that the log file can be is 32M. The log file size is then converted into aggregate blocks.

For example, the size of the log file for 15G partition using the default is 8192 aggregate blocks using 4k-block size.

Logredo operations

Logredo is the JFS utility that replays the log file upon start-up of the file system. The job of logredo is to replay all of the transactions committed since the most recent synch point.

The log replay is accomplished in one pass over the log, reading backwards from log end to the first synch point record encountered. This means that the log entries are read and processed in Last-In-First-Out (LIFO) order. In other words, the records logged latest in time are the first records processed during log replay.

Inodes, index trees, and directory trees

Inodes, index tree structures, and directory tree structures are handled by processing committed redopage records, which have not been superseded by noredo records. This processing copies data from the log record into the appropriate disk extent page(s).

To ensure that only the last (in time) updates to any given disk page are applied during log replay, logredo maintains a record (union structure summary1/summary2), for each disk page which it has processed, of which portions have been updated by log records encountered.

Inode Allocation Map processing

The extent allocation descriptor B+ tree manager for the Inode Allocation Map is journaled, and a careful write is used to update it during commit processing. The inode map control page (dinomap_t) is only flushed to disk at the umount time. For iag_t, persistent allocation map will go to disk at commit time.

Other fields (working allocation map. sum map of map words w/ free inodes, extents map inode free list pointers, and inode extent free list pointers) are at working status (i.e. they are updated in run-time). So the following meta-data of the inode allocation map manager needs to be reconstructed at the logredo time:

- The persistent allocation map of inode allocation map manager and next array are contained in Inode Allocation Groups.
- Allocation Group Free inode list
- Allocation Group Free Inode Extent list
- Inode Allocation Group Free list
- Fileset imap

Block Allocation Map (persistent allocation map file) is for an aggregate. There are three fields related to the size of persistent allocation map (pmap) file.

1. superblock.s_size: This field indicates aggregate size. It tells number of sector-size blocks for this aggregate. The size of aggregate determines the size of its pmap file. Since the aggregate's superblock is updated using sync-write, superblock.s_size is trustable at logredo time.
2. dbmap_t.dn_mapsize: This field also indicates aggregate size. It tells the number of aggregate blocks in the aggregate. Without extendfs, this field should be equivalent to superblock.s_size. With extendfs, this field may not be updated before a system crash happens. So logredo could need to update it. Extendfs is the JFS utility that could provide the functionality to increase the file system size. Ideally, the file system should have its size increased by using the Logical Volume Manager (LVM).
3. dinode_t.di_size: For an inode of pmap file, this field indicates the logical size of the file. (I.e. it contains the offset value of the last byte written in the file plus one. So di_size will include the pmap control page, the disk allocation map descriptor control pages and dmap pages. In JFS, if a file is a sparse file, the logical size is different from its physical size.

Note: The di_size does not contain the logical structure of the file, i.e. the space allocated for the extent allocation descriptor B+ tree manager stuff is not indicated in di_size. It is indicated in di_nblocks.

The block allocation map control page, disk allocation map descriptor (dmap) control pages and dmap pages are all needed to rebuild at logredo time.

Overall, the following actions are taken at logredo time:

- Apply log record data to the specified page.
- Initialize freelist for directory B+ tree manager page or root.
- Rebuild inode allocation map manager.
- Rebuild block allocation map inode.

In addition, in order to ensure the log record applies only to a certain portion of page one time, logredo will start NoRedoFile,

The three log record types: REDOPAGE, NOREDOPAGE, NOREDOINOEXT, and UPDATEMAP, are the main force to initiate these actions.

If the aggregate has state of FM_DIRTY, then fsck.jfs will run after the logredo process since logredo could not get 100% recovery.

The maps are rebuilt in the following way: At the init phase, storage is allocated for the whole map file for both inode allocation map manager and block allocation map inode and then the map files are read in from the disk. The working allocation map (wmap) is initialized to zero. At the logredo time, the wmap is used to track the bits in persistent allocation map (pmap). In the beginning of the logredo process the allocation status of every block is in doubt. As log records are processed, the allocation state is determined and the bit of pmap is updated. This fact is recorded in the corresponding bits in wmap. So a pmap bit is only updated once at logredo time and only updated by the latest in time log record.

At the end of logredo, the control information, the freelist, etc. are built from the value of pmap; then pmap is copied to wmap and the whole map is written back to disk.

The status field s_state in the superblock of each file-system is set to FM_CLEAN provided the initial status was either FM_CLEAN or FM_MOUNT and logredo processing was successful. If an error is detected in logredo the status is set to FM_LOGREDO. The status is not changed if its initial value was FM_DIRTY. fsck should be run to clean up the probable damage if the status after logredo is either FM_LOGREDO or FM_DIRTY.

Log record Format

The log record has the following format:

<LogRecordData><LogRecLRD>

At logredo time, the log is read backwards. So for every log record JFS reads, LogRecLRD, which tells the length of the LogRecordData.

Logredo handles Extended Attributes (EA)

There is 16-byte EA descriptor which is located in the section I of dinode. The EA can be inline or outline. If it is inlineEA then the data will occupy the section IV of the dinode.

The dxd_t.flag will indicate so. If it is outlineEA, dxd_t.flag will indicate so and the single extent is described by EA descriptor. The section IV of dinode has 128 bytes. The xtroot and inlineEA share it. If xtree gets the section IV, xtree will never give it away even if xtree is shrunk or split. If inlineEA gets it, there is a chance that later inlineEA is freed and so xtree still can get it.

For outlineEA, FS will synchronously write the data portion so there is no log record for the data, but there is still an INODE log record for EA descriptor changes and there is a UPDATEMAP log record for the allocated pxd. If an outlineEA is freed, there are also two log records for it. One is INODE with EA descriptor zeroed out; another is the UPDATEMAP log record for the freed pxd.

For inlineEA, the data has to be recorded in the log record. It is not in a separate log record. Just one additional segment is added into the INODE log record. So an INODE log record can have at most three segments. When the parent and child

inodes are in the same page, there is one segment for parent base inode; one segment for child base inode; and maybe one for the child inlineEA data.

More detail flow of Logredo

Below are the major steps that logredo must complete.

- Validate that the log is not currently in use.
- Recover if the JFS partition has increased in size.
- Open the log.
- Read the superblock and check the following fields: version, magic, state. If state is LOGREDONE, update the superblock and exit.
- Find the end of the log and initialize the data structures used by Logredo.
- Replay log. This reads the log backwards and processes records as it goes. Reading stops at the place specified by the first SYNCPT that is encountered.
- Start processing log records. There are only seven possible log records.
- After each loop through the different log records, check to see if the transaction just completed was the last for the current transaction, then flush the buffers.
- After processing all of the log records, check to see any 'dtpage extend' records were processed. If so go back and rebuild their freelists.
- Run logform so the following disk pages starting from the beginning of the log are formatted as follows:

  page 1 - log superlock
  page 2 - A SYNC log record is written
  page 3 - N - set to empty log pages.

- Flush data page buffer cache.
- Finalize the file system by updating the allocation map and superblock.
- Finalize the log by updating the following fields: end, state, and magic.

Next, all of the possible log records that the logredo utility must handle are described.

LOG_COMMIT record is used to insert the transaction ID (tid) from commit record into the commit array.

LOG_MOUNT record is the last record to be processed.

LOG_SYNCPT record is the log synch point.

LOG_REDOPAGE record contains information used to do the following operations:

- Update inode map for inodes allocated/freed.
- Update block map for an inode extent.
- Establish NoRedoFile or NoRedoExtent filters.
- Update block map for extents described in extent allocation descriptor B+ tree manager (xtree) root or node extent allocation descriptor list.

LOG_NOREDOPAGE record starts a NoRedoPage filter for xtree or dtree node.

LOG_NOREDOINOEXT record starts a NoRedoPage filter for each page in the inode extent being released.

LOG_UPDATEMP record is the update map log record, which describes the file system block extent(s) for the block map that needs to be marked.

Summary

When recovering after a system failure, JFS reads through the log file and if needed redoes each committed transaction. After redoing the committed transactions during a file system recovery, JFS locates all the transactions in the log file that were not committed at failure time and rolls back (undoes) each sub-operation that had been logged.

By logging JFS can restore the file system to a consistent state in a matter of seconds, by replaying the log file. By logging only meta-data changes JFS is able to keep the performance of this file system high.

Acknowledgements

The JFS Linux development team comprises:
Steve Best – sbest@us.ibm.com
Dave Kleikamp – shaggy@us.ibm.com
Barry Arndt – barndt@us.ibm.com

References

[JFSOverview]: “JFS overview” Steve Best, http://www-4.ibm.com/software/developer/ /library/jfs.html

[JFSLayout]: “JFS layout” Steve Best, Dave Kleikamp, http://www-4.ibm.com/software/developer/ /library/jfslayout/index.html

"JFS" Steve Best, published by Journal of Linux Technology April 2000 issue http://linux.com/jolt/

“Journal File Systems” Juan I. Santos Florido, published by Linux Gazette http://www.linuxgazette.com/issue55/florido.html

“Journaling Filesystems” Moshe Bar, published by Linux Magazine August 2000 issue http://www.linux-mag.com/2000-08/toc.html

“Journaling File Systems For Linux” Moshe Bar, published by BYTE.com May 2000 http://www.byte.com/column/servinglinux/BYT20000524S0001

Source code for JFS Linux is available from http://oss.software.ibm.com/developerworks/opensource/jfs

JFS mailing list. To subscribe, send e-mail to majordomo@oss.software.ibm.com with “subscribe” in the Subject: line and “subscribe jfs-discussion” in the body.

Trademark and Copyright Information