Inferno Flash File System: Architecture Notes: Release 1.0

Nigel Roles
Vita Nuova Holdings Ltd

Table Of Contents

Figures

Error! No table of figures entries found.

1 Introduction

This document describes how the Inferno Flash File System, as defined in [1], is mapped onto various flash media.

There is a portable section that defines, amongst other things, the exact format of the log messages, and one or more media specific sections. This corresponds to the implementation, which is divided into a portable library (`/liblogfs`), and a media specific plug-in library (e.g. `/libnandfs`).

2 Portable Part

2.1 Blocks and pages

The universal view of the underlying medium is that it consists of a set of blocks, each composed of the same number of pages. Each page has the same size.

Blocks must be erased before any of the pages in the block can be written to. Erasing a block sets all the data bits to 1.

When writing, it is only possible to change a 1 to a 0. Logfs ensures this by only writing to any particular area once.

Blocks can be read or written as a whole.

Pages can be written as a whole. Ranges of bytes may be read from a particular page.

2.2 Tags and paths

Blocks have metadata, which consists of a *tag* and a *path*. The *tag* is byte quantity, and defines the block type, dividing blocks into free, log, data, and boot blocks. The tag values are

Name	Value
`LogfsTnone`	`0xff`
`LogfsTboot`	`0x01`
`LogfsTlog`	`0x06`
`LogfsTdata`	`0x18`
Reserved	`0x67`
Reserved	`0x79`
Reserved	`0x1f`

These values have been carefully chosen to provide additional properties that might be of assistance when implementing medium specific parts. See [2] for details. Suffice it to say here that changing these values, or adding new ones is not easily possible.

`LogfsTnone` represents free blocks, which are also erased. These may be converted to any of the other block types.

`LogfsTboot` represents a boot block.

`LogfsTlog` and `LogfsTdata` represent the log and data blocks of the log structured file system.

Block *path*s are reference numbers used to manage the blocks. A path may be up to 34 bits in size, and has the following format:

```
rr rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrcc
```

`cc` is the copy generation number, used to detect and resolve duplicates created when blocks are copied without the old one being erased, as can happen when the power fails at inconvenient times. The generation number is incremented modulo 4 every time a block is

copied. During initialisation, if blocks with the same tag are detected with the same `rrrr`, and different `cc`, the older of the two is erased and formatted as `LogfsTnone`. A block with copy generation number `g0` is only older than a block with copy generation number `g1` if

```
g1 == (g0 + 1) % 4
```

`rrrr` is the block reference number, and is interpreted differently according to the block type. For boot and data blocks it is the block number, and counts from 0. For log blocks, it has the form

```
bbbbbbbb bbbbbbbb bbbbbbbb bbbbbbgg
```

where `gg` is the log generation number. This distinguishes between the active log and the swept log, if a log sweeping operation is interrupted, in much the same way as the copy generation protects against interruptions during general block copying. During initialisation, at most two different log generations may be found for a file system to be regarded as consistent.

2.3 Log messages

As documented in [1], there are 7 log messages. Whole messages are packed into pages of log blocks. If a message does not fit in the current page, the message `LogTend` is added to the end of the page (if there is room), and the next page advanced to. `LogTwrites` can be trimmed to fit by reducing the amount of data in them.

Each message has the general form:

type[1] *size*[2] *path*[4] *payload*[n]

all numeric quantities are stored least significant byte first, as per Styx 2000 messages. For more information on the notation see *intro*(5). The *size* of the above message would be 4 + n. The types are represented by vaguely meaningful ASCII characters. In all but `LogTstart` (and `Tend` which has no path at all), *path* is the lower 32 bits of the path part of the `Qid` which represents the referenced file in the Styx 2000 protocol; again, see *intro*(5). Note that the top 32 bits of a qid path are always 0.

2.3.1 `LogTstart`

This is an optional message used to label each log block, if the underlying medium cannot do this. Logfs on NAND does not use this message. The message has the form

`'s'` *size*[2] *path*[4] *nerase*[4]

size always has value 8. The *path* is the for the log block as per section 2.2. *nerase* is the number of times the block has been erased. If present, `LogTstart` occurs once per block, and is the first message in the block.

2.3.2 `LogTcreate`

This message indicates the creation of a file:

`'c'` *size*[2] *path*[4] *perm*[4] *newpath*[4] *mtime*[4] *cvers*[4] *name*[s] *uid*[s] *gid*[s]

Path is the qid path for the directory in which the file is being created. *Newpath* is the path for the new file.

Cvers is the create version. When a file is first created, *cvers* is 0; every time it is re-created or truncated on opening, *cvers* is incremented. All writes include the *cvers* so it is clear which version of the file the write belongs to. This allows obsolete writes to be easily detected and omitted when sweeping a log. *Name* is the name of the file; *uid* and *gid* are the name and group of the creator.

2.3.3 `LogTremove`

This message indicates the removal of a file:

`'r'` *size*[2] *path*[4] *mtime*[4] *muid*[s]

Mtime and *muid* are the time of removal of the file, and the remover of the file; these affect the parent directory of the removed file.

2.3.4 `LogTtrunc`

This message indicates the truncation on opening of a file:

`'t'` *size*[2] *path*[4] *mtime*[4] *cvers*[4] *muid*[s]

See section 2.3.2 for a description of *cvers*.

2.3.5 `LogTwrite`

This message indicates a write to a file:

`'w'` *size*[2] *path*[4] *offset*[4] *count*[2] *mtime*[4] *cvers*[4] *muid*[s] *flashaddr*[4] [*data*[n]]

The *data* is optional as only writes under a certain size are stored in the log, the rest being held in 1 or more adjacent pages of a data block. There is an upper limit of [128] bytes for a `Twrite` with embedded data.

The *flashaddr* indicates the location of the storage. It has the form

`laaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa`

where `l` is 1 if this data is in the log (and hence the *data* field of the message is present), or 0 if the data is in a data block. `Aaaaa` is the address of the data; this is computed as follows:

```
((b * ppb + p) * bpp + o)
```

where `b` is the block path (log or data), `ppb` is the number of pages per block, `p` is the page number, `bpp` is the number of bytes per page, and `o` is the offset within the page. Thus `aaa` is a logical byte address.

2.3.6 `LogTwstat`

This message indicates a change of the file metadata:

`'w'` *size*[2] *path*[4] *name*[s] *perm*[4] *uid*[s] *gid*[s] *mtime*[4] *muid*[s]

As per *wstat*(5), if any numeric fields are set to all ones, or any string fields have length 0, that metadata is not changed.

Whilst it is not erroneous to store a message with all fields set to *don't change*, implementations should normally intercept this case and interpret it as an indication to ensure that all data for this file is stored on the medium (known as *wstat flush*). In the case of logfs, this means flushing any buffered log data, and recursively applying *wstat flush* to the underlying medium.

2.3.7 `LogTend`

This message indicates that there are no more messages in particular log page:

`'e'`

Note this message has no *size* or *path*. `LogTend` is not present if the preceding message completely fills the page.

2.4 Formatting file systems

File systems are formatted by setting a selection of blocks to type `LogfsTboot`, and the remainder to `LogfsTnone`.

2.5 Initialising file systems

2.5.1 Consistency checking

The block tags are examined to partition the blocks into boot, log, data, and free blocks. The boot, log, and data partitions must meet further conditions for the file system as a whole to be declared consistent.

Within each partition the block paths are examined to identify duplicates that can be resolved by erasing the older one. For boot blocks, if this results in a set of blocks sequentially numbered from 0, then the boot partition is declared consistent. For data blocks, gaps in the sequence are allowed, as these represent data blocks which were used, but then were freed.

The log blocks are further partitioned into generations. There can be at most two generations, and the generation number must be sequential. If there are two generations, there has been an interruption during sweeping of the log, and some more care is required.

If there is a single generation, it is consistent if the blocks number sequentially from 0.

If there are two generations, the newer one (the swept log) must number sequentially from 0. If the older one (the active log) numbers sequentially from a number higher than the highest number in the swept log, the file system is consistent. If the swept log ends on the same number that the active log starts on, the last block of the swept log must be erased and formatted free to achieve consistency. Any other combination is inconsistent.

2.5.2 Replaying the log(s)

The file tree is recreated by replaying the log(s) if any. If there are two logs, the swept log is replayed first. In such cases, dependent on the exact nature of the interruption, there may be

some duplicated information. This can result in inconsistencies (e.g. `LogTremove`, `LogTtrunc`, `LogTwrite`, or `LogTwstat` messages referring to non-existent files) that can safely be ignored. This will last for at most the first block of the active log.

It is not normal to find such inconsistencies when replaying a single log, though the robustness of the approach means that in most cases the inconsistencies can be handled by not applying the log message.

2.5.2.1 Replaying `LogTcreate`

1. Check that the parent path exists and is a directory
2. Check that the new path does not exist
3. Add new entry to in-memory file tree

2.5.2.2 Replaying `LogTremove`

1. Check that path exists and is not the root
2. Check that parent is a directory (this is an internal error)
3. If the path is a directory, check it is empty
4. Remove entry from in-memory file tree

2.5.2.3 Replaying `LogTtrunc`

1. Check that path exists
2. Check it is not a directory
3. Set the in-memory `cvers` to the message `cvers`
4. Reset the in-memory file length and extent list

2.5.2.4 Replaying `LogTwrite`

1. Check that path exists
2. Check it is not a directory
3. Check that the `cvers` of the message matches the `cvers` of the in-memory entry (if it does not, then this is an obsolete write
4. Insert this new extent into the extent list
5. Update the `mtime` and `muid`
6. Update the `length`, if this write made the file bigger

2.5.2.5 Replaying `LogTwstat`

1. Check that path exists
2. Update any fields which are marked to change

2.5.3 Finding the used data pages

Once the file tree has been recreated from the log messages, the set of data pages in use can be determined by examining the extents of each file in the file tree. All unreferenced data pages must be regarded as free and dirty (i.e. in need of erasing before further use). It is a useful invariant that data blocks always have some valid (not free and dirty) data in them.

Thus, after finding all the referenced data pages, any data blocks that are entirely free and dirty should be erased and converted to free blocks.

2.5.4 Setting the log insertion point

The insertion point for adding to the log is the next page after then last page replayed. Note that adding to the log in the presence of a partially swept log is entirely acceptable.

This completes initialisation; there is an in-memory file tree, one log (or two logs, with disjoint block numbers so it is clear which log embedded write data is held in), and a map of unused and unused data pages.

2.6 Processing Styx 2000 messages

The Styx 2000 messages trigger the following logfs specific actions:

2.6.1 Finding free space for the log

Once a log block has filled, a new one is required to extend the log. The `/liblogfs` implementation looks first for a free block which can be converted into a log block. For this to possible there must be at least [5] blocks free; this is called the *data extend threshold*. This apparently conservative threshold allows some flexibility to

1. replace bad blocks (see 2.9)
2. scavenge data blocks (see 2.6.7.2)

If no such block is available, the threshold is lowered to the *log extend threshold*, but only if the kind of message being added to the log is likely to make it smaller when swept (see 2.10). If, still, there is no block is available, the log is swept (see 2.7) to see if this increases the free space. If this does not help, then the file system is regarded as full. Note that if the invariant described in 2.5.3 is maintained, there cannot be any data blocks that can be converted to log blocks, so the file system really is full.

2.6.2 `Tclunk`

There is no logfs specific action for `Tclunk`, other than the `Tremove` actions implied by a fid open in `ORCLOSE` mode.

2.6.3 `Tflush`

There is no logfs specific action for `Tflush`.

2.6.4 `Topen`

If a file is opened in `OTRUNC` mode, meets the criteria for truncation, and is non-empty:

1. the in-memory `cvers` and `qid.vers` anre incremented
2. the in-memory `mtime` and `muid` are updated
3. a `LogTtrunc` log message is generated
4. the data pages reached from the extent list are marked free and dirty (i.e. reclaimable)
5. the extent list is reset
6. the in-memory file length is set to 0

2.6.5 `Tcreate`

Assuming all the normal tests for the rights to create a file are passed, a new in-memory file entry is created, and a `LogTcreate` message generated. The `path` is generated by choosing a currently unused value.

2.6.6 `Tread`

Reading requires no logfs specific action. Some care is needed when locating the data described in the extent list:

1. if implementations buffer log pages the required data may not have made it to the medium yet, and will be in the buffer
2. if the data is embedded in a `LogTwrite`, then the block indicates whether it is in the active or swept log

2.6.7 `Twrite`

Fundamentally, writing involves

1. allocating and writing free data pages if the data exceeded the embedded write threshold
2. writing a `LogTwrite` message to describe this
3. identifying the data pages which are now unreferenced, and marking them free and dirty
4. updating the extent list to indicate the new data location
5. updating the length if the write made the file bigger
6. updating the `mtime` and `muid`
7. incrementing `qid.vers`

This is actually quite complex, but at least this is the only complex bit of logfs.

2.6.7.1 Several writes are required

Firstly, the `Twrite` may be too big to fit anywhere in one piece, so the operation will be broken down into one or more smaller writes. The only downside here is that a large write can fail part way through.

Once the amount of data to write drops below the embedded data threshold, this is still no guarantee that the message will fit in the current log page. Implementations are recommended to split the remaining write, choosing the size of the first write to exactly fill the remaining space in the unflushed log page.

2.6.7.2 Allocating free data pages

While the amount of data left to write exceeds the [128] byte embedded data threshold, data pages need allocating. Allocation algorithms are subject of much debate; it is sufficient here to describe that used in `/liblogfs`. The *flashaddr* is a byte address, so combined with a length can describe any range of bytes, regardless of how many page and block boundaries are crossed. In `/liblogfs` the allocation is restricted to not crossing block boundaries. Because of the ‘write once before erase’ nature of media such as NAND flash, allocation must be in terms of whole pages. In order of priority the algorithm searches for

1. The largest contiguous run of pages in existing data blocks
2. The existing data block with the most free and dirty pages. This block is then scavenged, by copying the still valid pages to a free block, labelling the free block a data block with the same path, and erasing and formatting the original one free. This converts the free and dirty pages to free and not dirty (i.e. available). Note that when finding a free block a lower threshold of [2] is used; this is called the *transfer threshold*. Thus step 2 can fail, and step 3 succeed, in finding space.
3. A free block that is then converted to a data block. This is subject to the *data extend threshold,* as mentioned in section 2.6.1.

At this point, if no pages have been found, then all the pages of all the data blocks are full, and the free block count is below the data extend threshold. There is nothing for it but to sweep the log (see 2.7) in the hope that the log compresses enough to increase the number of free blocks to at least the data extend threshold. If it doesn't then the file system is full.

Note that the threshold for a block transfer is [2], not 1. This is to allow headroom for a block *replacement*, should this need to happen at an inconvenient time; see 2.9.

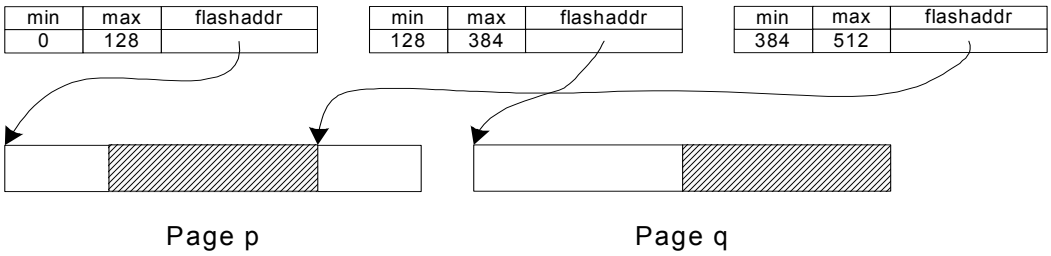
2.6.7.3 Identifying the unreferenced data pages

Unfortunately, this turns out to be the hardest part. There are two straightforward cases:

1. Writing at or past the end of the file will not orphan any data pages
2. Replacing the entire file will orphan all data pages associated with the file

After that it gets tricky. It doesn't strictly matter if some orphaned data is not identified, as the next file system initialisation will find it (see section 2.5.3), but typical devices which these file systems operate in (e.g. PDAs) are never turned off. In summary, all the data must be found.

Why is it hard? A picture will help. Imagine creating a file of 512 bytes, and then overwriting a patch in the middle of the file (say at offset 128, size 256). This patch is big enough to require a data page as well. Further, the page size of the device is 512 bytes. You have three extents, referring to two different data pages:



Now imagine overwriting bytes [384, 512) again. The third extent is replaced entirely, but it is wrong to imagine that page p is now free. Worse, it is not sufficient to look at the immediately preceding extent to see if it refers to the same page; in this example you have to look back two extents to realise that the page is still being used. In general, whilst not infinite, the worst case is still quite large, as there can be `pagesize / 2` extents referring to the same page.

What doesn't help is that a single extent can reference many pages; when split, the two fragments might well refer to different pages.

What does help is that we know all the referring extents for a particular page are from the same file, so the search for relevant extents is confined to a single extent list.

Ideally, each page would have a reference count, but this would be hard to maintain correctly, and prohibitive in terms of memory use. Also, given that the two straightforward cases listed previously represent quite a lot of normal file writes, it seems acceptable for unusually aligned writes to cost a little more.

The algorithm used in `/liblogfs` is as follows:

```
for (e in all extents touched by write) {
      before = portion of e before the write
      after = portion of e after the write
      middle = bit left over after removing before and after
      for (p in all pages referenced by e)
            add p to list of pages, associate a count of 0
      for (p in all pages referenced by before)
            add one to count
      for (p in all pages referenced by middle)
            subtract one from count
      for (p in all pages referenced by after)
            add one to count
}
for (e in all extents of file)
      for (p in all pages referenced by e)
            if (p is in list)
                  add one to count
for (l in all list entries)
      if (l.count is 0)
            free page l.page
```

This essentially recreates the reference counts for the all the pages, after the extent list has been adjusted to account for the write. If a page has no references, it can be freed. This is not the lightest of algorithms, but is accurate, and is probably best improved upon by detecting more special cases.

In line with the data block invariant, any data blocks that become entirely free and dirty as a result of this operation should be erased and formatted as free blocks.

2.7 Sweeping the log

Sweeping the log is well described in [1]. When allocating new data blocks for the swept log, the *transfer threshold* is used, as it is known that the log gets no bigger as it is swept, i.e. as swept log blocks are created, active log blocks are being freed at the same or faster rate.

2.8 Closing down a file system

Before disconnecting the power, or unmounting the file system, the buffered log page must be flushed.

2.9 Handling bad blocks

Bad block detection is the responsibility of the medium specific portion, but the portable part will handle the consequences. The portable portion assumes that problems will be detected before becoming irrecoverable, allowing the opportunity to copy the data to another block, and mark the current one bad. This pretty much assumes an error detecting and correcting code is in use somewhere on or below the medium specific level. The portable portion accepts indications of potential failure on read, write, and erase. In the cases of read and write, the data is copied to a free block. In all cases, the medium specific portion is invited to mark the block bad.

Free block allocation for the purposes of replacing a block is not subject to any threshold. Nevertheless, it is always possible to contrive examples of where a file system will run out of free blocks with which to make a replacement. How much protection is available from this is down to the *transfer threshold*. It is an indication of how many free blocks should be available for replacements at any one time.

2.10 Dealing with full systems

If a file system is full, then there are at most *data extend threshold* free blocks. If any free space is required, the log will have to be swept, and can be, because this will only require *transfer threshold* free blocks. However, this might not create any more space. Removing files is the only option at this stage. Still, if log records need to be written to confirm a file removal, and there is no space to extend the log, an impasse is reached. This is not good, as a reasonable response to a lack of space is to delete some files; being unable to delete the files because there is no space to do it not ideal.

It is tempting to say that the thresholds be temporarily lowered so that the log can be swept, and space freed. The problem is that if the sweep does not recover any space, we are worse off as there are now more blocks committed to log and data than we want. It's necessary to know that the sweep will recover space, which is not an easy question to answer, without walking the entire log to simulate the process. Another possibility is to allow only certain messages into the log when times are bad. These would be `LogTremove` and `LogTtrunc`; whilst not guaranteed to result in a smaller log, there is a strong likelihood. This is the current implementation, as hinted at in section 2.6.1. Thus when the file system fills, a typical user's approach is to remove some files.

3 NAND Flash

All NAND issues are covered by [2].

4 References

[1] C H Forsyth, "*A Flash File System For Inferno*", Vita Nuova, 27th September 2002

[2] N G Roles, "*Inferno Flash File System: NAND Flash Layout Specification*", 1.0, Vita Nuova