A Flash File System for Inferno (Draft)

C H Forsyth

Vita Nuova

27 September 2002
(Revised 26 October 2002)
(Revised 14-20 November 2002)

1. Introduction

Flash memory has some peculiar properties, which are given here only briefly. It is organised as a collection of erase units or blocks. Unlike RAM or magnetic disc, data cannot be arbitrarily overwritten. For instance, with NOR flash, a data bit can be changed from 1 to 0, but never from 0 to 1, unless the whole block is erased, resetting all its bits to 1. NOR flash is often directly addressable. The rules are different again for NAND flash, where blocks are typically subdivided into pages of 256 or 512 bytes, and each page can be written at most two or three times (with restrictions on bit transitions similar to NOR flash). It is most efficient to write data to pages many bytes at a time. In some implementations NAND flash pages can have several regions, with separate write-limits, and the devices typically include a small 'auxiliary' region to store ECC and other implementation data, that can be updated separately from the main data area of the page. Because of the need to maintain an ECC, however, pages or subpages often cannot be written more than once, even though the bit transition rules are followed for the data, simply because the bits in the ECC are themselves subject to the same restrictions, and the ECC for the modified data is unlikely to satisfy the constraints. Consequently, it is often easier in practice to regard NAND flash as a collection of write-once blocks, quite different from NOR flash.

The existing Inferno file systems for local permanent storage are *kfs*(3) and *dossrv*(4), both designed for use on conventional rewritable storage devices (eg, hard drives and diskettes). They can, however, be used with flash memory devices if access is mediated by the Flash Translation Layer implemented by *ftl*(3). It gives the file systems the illusion that they are using a normal rewritable medium, essentially by remapping a data block to a new location in flash each time it is overwritten, using a logical to physical map, and recovering the flash space taken by the overwritten data in a garbage collection or scavenging phase done in erase-unit sized chunks. The file system implementations still need to be tweaked for the scheme to be viable; for instance, insisting on updating file and directory access times will increase the turnover of erase units and thus cause excessive wearing, and so flash implementations avoid that.

The use of such file systems over a flash translation layer not only adds a new layer of complexity, but merely compensates for the peculiar nature of flash rather than trying to use it to best advantage. An attractive alternative is to consider using a log-structured file system,[1,2] inspired by the use of a journal in transaction-oriented database systems. In its simplest form it stores file system data and metadata in the form of a log: each change to data or metadata is noted in a record appended to the log. For example, a log might contain records summarising file creation and deletion requests, each giving a reference to a parent directory and the name of a file; data written to the file system might appear in a record that refers to the file, an offset within it, and the data written at that offset. The current logical file system structure can be reconstituted at any time (eg, after a crash) by replaying the log and re-executing the requests contained in it. Application requests to read data from a file are satisfied by returning the appropriate portions of the 'write' records found in the log. Note that data is never overwritten; instead, later entries in the log supersede earlier ones. Although log-structured file systems were originally designed to speed certain file system operations and enhance reliability on conventional storage media, the append-only property of the log looks appropriate for flash. Furthermore, log-structured systems also provide for garbage collection to

compress the size of the log and reclaim space occupied by obsolete data, which can be made to satisfy the requirement on flash for erase-unit sized recycling, and wear-balancing. Log-structured or journalling file systems have therefore become attractive as the basis for a flash-oriented file system.[3,4] They are not the only choice, however,[5] and require special care when used with NAND flash.

This paper describes the design of such a file system for use with Inferno.

2. Basic requirements

The first set of requirements is independent of its being a flash file system:

- The file system implements a full Inferno file system: names contain Unicode characters and case is preserved; lookup is case-sensitive (eg, unlike *dosfs*(4)); and names of at least 256 characters should be allowed (as supported by the Styx protocol revision); the Inferno mode permissions defined by *sys-stat*(2) and *stat*(5) are implemented and enforced; file and directory modification times are implemented; and user and group names are implemented.
- The file system is implemented by a Styx server, as usual for Inferno. This immediately imposes the requirement that it implement the contents of Section 5 of the Inferno Programmer's Manual, which defines the protocol. (If it is subsequently implemented inside the kernel, the requirement stands but the ultimate implementation is different, being based on the kernel's `Dev` interface instead of protocol messages.) In the rest of this document, I shall assume the protocol message interface, and also that the validity of the message including its fids has been checked (this can be done easily using primitives in *styxservers*(2), for instance).

There is a further set of requirements from a particular customer:

- There must be a way for an application to discover the amount of flash space used and available.
- The implementation should support different media types, certainly including both NAND and NOR flash, but also including a file system inside an ordinary file (eg, for emulation environments), or recorded on some other kind of removable medium (eg, flash disk).
- Access times should not be updated when directories are read; it might even be desirable not to update them on flash when files are read. It is worth noting that this immediately fails the requirements of *stat*(5), but since many existing file servers and device drivers do something similar, we might take the same licence.

There is a further requirement that has non-trivial effect on the design:

- The start-up time must be kept low.

3. Specifications

The specifications will often use the abstract syntax notation of VDM[6] since it is reasonably expressive and compact, yet straightforward to map into Limbo and C data structures. The specification is developed as follows. The Styx protocol messages are analysed to extract the subset of interest here. Next, an abstract file system is defined that represents essential properties of an Inferno file system. A set of abstract commands that operate on such a file system is derived from the protocol messages. Those help determine the contents of a log of file system activity. The effect of replaying a log to restore a file system state is specified, followed by a a specification of scavenging free space when the log fills. Finally, a change to the previous specifications is proposed to give faster initialisation by storing the log separately from file system data.

3.1. Protocol messages

The file system protocol messages are specified informally but with reasonable precision in section 5 of the Programmer's Manual. A client operates on the file server using T-messages, so we shall concentrate on those, and in particular on the subset that changes the state of the file system, eliminating requests that control the protocol, or simply retrieve data or attributes. Here is the resulting set taken from *intro*(5):

```
size[4] Tcreate tag[2] fid[4] name[s] perm[4] mode[1]
size[4] Topen tag[2] fid[4] mode[1]
size[4] Twrite tag[2] fid[4] offset[8] count[4] data[count]
size[4] Tclunk tag[2] fid[4]
size[4] Tremove tag[2] fid[4]
size[4] Twstat tag[2] fid[4] stat[n]
```

`Topen` is included because the mode can include `OTRUNC`, which truncates the given file. `Tclunk` is included because it might cause buffered data to be written out, depending on the implementation, and if the `ORCLOSE` option is specified when a file is opened, `Tclunk` can cause the file to be removed. (Note that in some implementations access times might be updated by `Tread` and other retrieval operations but at least initially I ignore that here.)

3.2. File system

The initial specification is based on aspects common to all Inferno file systems, as discussed in *sys-intro*(2), *intro*(5), and elsewhere in the Programmer's Manual. An entry in an Inferno file system is either a file or a directory and all entries have a particular set of attributes (see *stat*(5)), including its name, type and permissions, the user ID of its owner, modification time, and others. One important attribute is the *qid* (see *intro*(5)), a triple of integers (*path*,*vers*,*type*) where the *path* uniquely identifies the file in the file server. A representative subset of the attributes will be used here, to simplify the presentation; in particular, only the *path* component of a *qid* will be shown.

Rather than going through the full step-wise refinement from the most abstract model of an Inferno file system, essential properties useful here will only be stated:

1. A complete file system is a tree rooted at a single directory.
2. All files (including directories) have unique qid paths.

Thus, a directory *Entry* can be defined as followed:

$$
\begin{aligned}
&Entry :: path : Path \;\; name : String \;\; mode : Int \;\; uid : String \;\; mtime : Int \;\; (Dir \mid File) \\
&Path = Int \\
&Dir :: files : Entry\text{-}set \\
&File :: data : Data\text{-}set \\
&Data :: offset : Int \;\; data : Bytes
\end{aligned}
$$

where *Bytes* is a sequence of bytes. The file system is rooted at a single entry:

$$FS :: root : Entry$$

Furthermore, paths are unique (by Inferno file system rules):

$$(\forall e_1 \in Entry)(\forall e_2 \in Entry)(path(e_1) = path(e_2) \rightarrow e_1 = e_2)$$

We can therefore use the path to refer to an *Entry* unambiguously.

3.3. File system commands

The aim now is to derive the abstract syntax of file system commands from the protocol messages. The commands contain only the parameters of each protocol operation that are essential for file system modification. For instance, the size and tag fields are not relevant: they help to delimit messages and match them up, but they modify only file server state not file system state. More important, each message refers to a file indirectly through a *fid*, a number that a client gives to each file it has active, as discussed in *intro*(5). Fids can be reused, but when a message is interpreted, the fid always refers to a particular file. That file is uniquely known by a path that must be valid (if the fid is valid and the protocol is correctly implemented). Since we are ultimately to operate on files, not fids, we can therefore replace each fid instance by the corresponding path to form a command.

The `mode` operand of the `Tcreate` protocol message need not appear: it is used to open the resulting file, and does not affect the file system state, unless `OTRUNC` is specified, but that produces the same result as the create call itself. A separate command is needed, however, to reflect the appearance of `OTRUNC` in

`Topen` messages.

This results in the following set of commands:

$$\begin{aligned}
&\mathit{Command} = \mathit{Create} \mid \mathit{Remove} \mid \mathit{Trunc} \mid \mathit{Write} \mid \mathit{Wstat}\\
&\mathit{Create} :: \mathit{path}:\mathit{Path}\ \ \mathit{name}:\mathit{String}\ \ \mathit{perm}:\mathit{Int}\\
&\mathit{Remove} :: \mathit{path}:\mathit{Path}\\
&\mathit{Trunc} :: \mathit{path}:\mathit{Path}\\
&\mathit{Write} :: \mathit{path}:\mathit{Path}\ \ \mathit{offset}:\mathit{Int}\ \ \mathit{data}:\mathit{Bytes}\\
&\mathit{Wstat} :: \mathit{path}:\mathit{Path}\ \ \mathit{perm}:[\mathit{Int}]\ \ \mathit{uid}:[\mathit{String}]\ \ \mathit{gid}:[\mathit{String}]\ \ \mathit{mtime}:[\mathit{Int}]
\end{aligned}$$

Note that the path in *Create* is that of the directory where the new entry is created. There is no *Clunk* command: the effect of a `Tclunk` can be represented by an optional *Write* (if buffered) followed by an optional *Remove* (if `ORCLOSE` was set on open).

3.4. Log contents

The initial file system state is defined to be an empty root directory. Each subsequent *Command* modifies that state. The log is to satisfy the property that for a final state resulting from a given sequence of commands, replaying the corresponding log entries from the initial state will reconstruct that final state. The operations that can change the state of the file system are exactly those in *Command*. There must be therefore be a form of log entry to capture the effect of each *Command*. The simplest initial approach is just to log the contents of each *Command*. Thus, the log entry for a *Write* command would record the path, offset, and the data to be written to the file at the given offset. That still needs a little care to make it work.

Specifically, given a protocol message we substituted a file's path for its fid to obtain a *Command*; that is easy when actually running the protocol, because the server keeps a map relating fids and active files (and thus their paths), which it updates as fids come and go. In particular, if a file is created, the execution of a *Create* command produces a new entry with a new path, and if that file is then written, that new path will appear in the subsequent *Write*. That map is not available when a log is replayed. Thus, the effect of the *Create* must also be logged, so that the *Write* can be done on the right file. The log entry for *Create* therefore includes the resulting (new) path:

$$\mathit{L\_Create} :: \mathit{path}:\mathit{Path}\ \ \mathit{name}:\mathit{String}\ \ \mathit{perm}:\mathit{Int}\ \ \mathit{uid}:\mathit{String}\ \ \mathit{mtime}:\mathit{Int}\ \ \mathit{newpath}:\mathit{Path}$$

The parameters to *Remove*, *Trunc*, *Write* and *Wstat* commands are sufficient to reconstruct the state and just need to be logged. (To maintain modification times correctly, log operations must also contain a log entry time, but that is a relatively simple extension.)

3.5. Replaying the log

The file server starts with an initial state: an empty root directory $\mathit{root} \in \mathit{Entry}$. It also maintains a map:

$\mathit{paths}:\ \mathit{Path} \rightarrow_m \mathit{Entry}$, where $(\forall (p \mapsto e) \in \mathit{paths})(p = \mathit{path}(e))$.

Initially, $\mathit{paths} = \{\mathit{path}(\mathit{root}) \mapsto \mathit{root}\}$.

The server interprets each log entry in turn, using the map to find the appropriate *Entry* for each operation. For every log entry *l*, require $(\exists ! e \in \mathit{Entry})(\mathit{path}(l) \mapsto e) \in \mathit{paths}$. (If not, an internal error results, because a valid log entry must refer to an existing file in the current file server state, or the log does not accurately reflect the file system state when the entry was written.) Note that *e* must be somewhere in the current value of *root*. The replay operation does the following for each entry type:

- $\mathit{L\_Create}(\mathit{path}, \mathit{name}, \mathit{perm}, \mathit{uid}, \mathit{mtime}, \mathit{newpath})$
  Require $\mathit{is\text{-}Dir}(e)$.
  Require $\mathit{newpath} \notin$ **dom** *paths* (otherwise the log creates the same file twice).
  Add *f* to $\mathit{files}(e)$ where

$$f = \begin{cases} \mathit{mk\text{-}Entry}(\mathit{newpath}, \mathit{name}, \mathit{perm}, \mathit{uid}, \mathit{mtime}, \mathit{mk\text{-}Dir}(\varnothing)) & \text{if } (\mathit{perm} \,\&\, \mathit{DMDIR}) \neq 0 \\ \mathit{mk\text{-}Entry}(\mathit{newpath}, \mathit{name}, \mathit{perm}, \mathit{uid}, \mathit{mtime}, \mathit{mk\text{-}File}(\varnothing)) & \text{if } (\mathit{perm} \,\&\, \mathit{DMDIR}) = 0 \end{cases}$$

  Add $(\mathit{newpath} \mapsto f)$ to *paths*.

- *Remove*(*path*)
  Require $(\exists! d \in Entry)(is\text{-}Dir(d) \wedge e \in files(d))$. Remove e from $files(d)$, and remove $(path \mapsto e)$ from *paths*.
- *Trunc*(*path*)
  Require *is-File*(e), and set $data(e) = mk\text{-}File(\emptyset)$.
- *Write*(*path*,*offset*,*data*)
  Require *is-File*(e), and adjust $data(e)$ to reflect the effect of the write: adding the new *data*, and if it overlaps any existing data, trimming, splitting or discarding existing member of $data(e)$ as required.
- *Wstat*(*path*,...)
  Update the attributes of e from l.

Once the whole log has been replayed, the current file system state will be the same as it was when the last log entry was written.

3.6. Scavenging new space for the log

The file system state changes as files are created, updated, and removed, causing corresponding log entries to be written. Eventually, the log will fill the storage space allocated to it. That need not mean, however, that the file system is 'full'. Later log requests can render earlier requests obsolete or redundant. For instance, if a file is created then removed, the *Create* and *Remove* entries cancel out: the resulting file system state is the same as if the file had never been created (ignoring any side-effect on the modification time of the parent directory). When a *Write* is followed by another *Write* and the file addresses overlap, part or all of the data in the first *Write* is obsolete; if all the data is overwritten, by one subsequent *Write* or by many, that first *Write* log entry itself is obsolete. Similarly, when a file is written, and then truncated, the *Truncate* log entry makes the earlier *Write* requests irrelevant, including the associated data. Space can be recovered by building a new log that excludes all obsolete entries: read the old log in order and copy only currently relevant entries into the new log. Each entry is tested against the current state of the file system: that is, against the content of the *root*. That shows exactly what should be visible. This can be done efficiently using the *paths* map defined above, provided it is updated by the execution of *Create* and *Remove* commands, including during file system creation from the empty initial state when the log is built for the first time.

In all cases, for log entry l, if $path(l)$ is not in **dom** *paths* then discard l, because the file no longer exists. Otherwise $\exists(path \mapsto e) \in paths$, where $e \in Entry$ is in the tree *root*, and the rules are:

- *L_Create*(*path*,*name*,*perm*,*uid*,*mtime*,*newpath*)
  If $newpath \notin$ **dom** *paths* discard l. Otherwise, $\exists(newpath \mapsto f) \in paths$. Write a new *L_Create* record with the parameters of l adjusted to reflect the current attributes of f (eg, name, permissions, etc.), so that it reflects any *Wstat* requests applied since l to achieve the current state.
- *Remove*(*path*)
  Internal error: the initial play of this log entry ought to have deleted $path \mapsto e$ from *paths*, and *Remove* should have been discarded by the preliminary test of *path* above.
- *Trunc*(*path*)
  Discard l: any data written to the file before this *Trunc* will have appeared in preceding *Write* log entries, which will themselves have been discarded because they will not appear in the current state of e.
- *Write*(*path*,*offset*,*data*)
  The write is potentially useful, provided the *offset* and *data* contribute to the current content of e (implying that an implementation must somehow record which log entry contributed to $data(e)$). Otherwise, discard l.
- *Wstat*(*path*,...)
  Discard l: the preceding *L_Create* has been adjusted to reflect the effect of l, or else the effect of a subsequent *Wstat* has superseded l. Either way, l is redundant.

The result of doing this for the entire log is a compressed log that contains exactly those entries needed to reconstruct the current file system state when replayed. The simplest procedure is to copy all log entries

into a new sequence of log blocks, allocating new blocks as required. For flash media, this suggests that an implementation must reserve some blocks to allow copying to start when the limit on the current log has been reached. As log blocks are copied, the old blocks are erased and put in the reserve pool.

The compressed log contains only *L_Create* and *Write* log entries: only those are needed to recreate an arbitrary file system state. This matches our intuition: those operations build up content by creating directories of files that are filled with data, whereas the other operations just rearrange or modify that content. Note that it would therefore be possible to build a new log from scratch by simply traversing the tree *root* and writing log entries to represent its current contents, ignoring the current log contents. Working from the existing log is a linear process, not requiring recursion, but more important, the scheme above can be modified to compress a selected *tail* of the current log, allowing generational collection. When scavenging a partial log, the specification above must change significantly in its handling of *Trunc*, *Wstat* and *Remove*; for instance, they can only be declared redundant if it is known that the corresponding *L_Create* has been scavenged in the same partial scan. This might be done, for instance, by counting the scavenging scans and tagging each *Entry* with the scan number that last created it. Alternatively, *Trunc* can simply give the truncated file a new path and use a log entry similar to *L_Create*, that when replayed will create the file if necessary, and truncate it; when the log entry is read, if the *newpath* no longer exists, the entry can be discarded.[4] Obsolete *Write* requests can always be detected and discarded based on the current file system state. It also will still be necessary periodically to select at random a slowly-changing block to move, to encourage even wearing of flash.

3.7. Faster startup

As it stands, the only data structure in the file system implementation is the log alone, because data written is logged in the entry for the write. Thus, when stored in flash, all flash blocks are log blocks, and consequently every block must be read to reconstruct the log. Not all the data in each block must be examined—the actual data content in a write entry can be skipped and its location noted—but nevertheless the replay process must work through every used block. That might contradict the requirement for low initialisation time. An alternative method is to store the data in separate blocks from the log entries, distinguishing between data blocks and log blocks.

The resulting log entries are as follows (showing the additional *mtime* values mentioned above):

$$
\begin{aligned}
&L\_Command = L\_Create \mid L\_Remove \mid L\_Trunc \mid L\_Write \mid L\_Wstat \\
&L\_Create :: path{:}Path\ name{:}String\ perm{:}Int\ uid{:}String\ mtime{:}Int\ newpath{:}Path \\
&L\_Remove :: path{:}Path\ mtime{:}Int \\
&L\_Trunc :: path{:}Path\ mtime{:}Int \\
&L\_Write :: path{:}Path\ offset{:}Int\ count{:}Int\ where{:}Location\ mtime{:}Int \\
&L\_Wstat :: path{:}Path\ mtime{:}Int
\end{aligned}
$$

The data in a *Write* command is written in separately allocated space in a data block, and that location is logged in some form in the *L_Write* entry. Scavenging of *L_Write* is similar to scavenging of *Write*, since for a given $w \in L\_Write$, if there is an existing $e \in Entry$ corresponding to $path(w)$, then all of $where(w)$ contributes to $data(e)$, in which case w is copied as-is; or none of $where(w)$ appears in $data(e)$, in which case w is discarded; or $where(w)$ is trimmed to reflect only its contribution to $data(e)$.

3.8. Data space

Allocating data blocks separately requires they be subject to their own form of scavenging for free space. Recall that the file system state, including the data allocation state, is represented by the current set of *Entry* values. As the initial file system is built, and during replay of a log, a map can be maintained that tracks the space used and obsolete in each flash block, allowing selection of suitable blocks for scavenging. When the contents of a data block is moved and compressed, eliminating obsolete data, there are several ways of noting the move. The easiest is to give each data block a unique label, which remains constant across moves, and include the label in a *Location*.

A hybrid scheme is possible in which the data from tiny writes is kept in the log and larger writes allocate data in separate data blocks.

Whether the hybrid scheme is adopted or not, if data is allocated to separate blocks, the following

observations help to guide an implementation:

- Blocks are partitioned into three sets: *log*, *data*, and *free*. They are identified by special tag values in the blocks (NOR flash) or in the auxiliary area (NAND flash). (We can ignore the existence of any *spare* and *boot* blocks, since they are handled outside the log file system proper.)
- A given number of free blocks must be reserved for log and data scavenging: they are called *transfer* blocks, and are a tiny subset of the *free* blocks, at least two blocks. Having more than one transfer block allows the log and data to be scavenged independently.
- A data block can contain valid data, data that has become obsolete (eg, by file truncation or overwriting), and free space.
- When a block is found to be filled with obsolete material, it is erased and returned to the pool of free blocks.

A key observation is that data is valid if and only if it appears in an extent of a file *Entry* in the current file system state. Data could be allocated a flash page at a time, but in practice an extent-based scheme allocating in units of anywhere from a flash page to a flash block gives good results, and allows the free space map to be represented by a bitmap. In the current implementation an array of bitmaps is used, one per flash block, where the bitmap has one bit per page and fits in a 32-bit integer. The entries in the map are examined in rotation during a search for free space. If there is none, data blocks are scavenged: until enough space has been made available, a block with obsolete data is selected, and only its valid portions copied to a transfer block, making the remainder free again. If there is no space to be had by scavenging, the file system is full.

As material in a block becomes obsolete, it is better not to recover it immediately, since more material might become obsolete shortly (eg, when a file is rewritten). Instead it is better to wait until either the whole block becomes obsolete, or the obsolete space is needed because there is no other free space. This recovers as much obsolete material as possible for each data block erase and move.

3.9. Interruption

An *interruption* is an unexpected shutdown of the system, device, or file server (driver). It is important that the file system state be recovered after any interruption. Only sequences of write operations are important, since an interruption occurring during a read will have no effect on the file system state.

As noted earlier, the set of blocks is partitioned into log, data, free, and transfer blocks. The operations that might be interrupted are those that write to them. Let us look at each in turn:

- *Erasing or formatting a free block*

  A partially erased or formatted block can be erased and reformatted as free. (The details of detection and formatting conventions depend on the underlying flash implementation.)

- *Writing to the log*

  If a write to the log is interrupted, the log will be incomplete, and in some cases the last log entry will be incomplete. The implementation needs a way to detect these cases. It could write a special log entry to close the log. That does not allow repair but does allow the system to tell the user.

- *Copying a log block when scavenging*

  If a sweep of the log is interrupted, it is possible for the tail of the list of swept blocks to have the same log sequence number as the head of the list of unswept blocks. That implies an interruption during the sweep of that block, or after the sweep but before the original block could be erased and put in the free pool. The block at the tail of the swept blocks should be removed and returned to the free pool, and the sweep restarted, either completely, or if partial scavenging is implemented, with the head of the list of unswept blocks.

- *Writing to a file*

  A write to a file causes data to be written to a page in a data block and a separate log entry to be made. For consistency, data must be written before the corresponding log entry is made. That maintains the invariant that the log always accurately reflects the state of the data blocks. Note that if the interruption occurs between writing the data and writing the log entry, replaying the log will not show the data as allocated, which is correct, although the implementation will exceptionally need to

copy via a transfer unit first.

- *Writing to a data block*
  This is covered by the previous item.
- *Copying a data block to recover space*
  Data blocks are labelled as discussed earlier. If the interruption occurs during or just after a copy, before the original has been erased and freed, there will be two blocks with the same label. If the implementation distinguishes old and new instances of a block, consistency is guaranteed if the newer is always discarded (ie, erased and freed), since the older instance is known to be complete. The copy can then be restarted.
- *Moving a log or data block for load wearing*
  This is similar to the previous instance.

In several cases, both log and data blocks require a unique label, and furthermore need a way of distinguishing 'old' and 'new' instances when scavenging and copying. The current implementation simply numbers log blocks and data blocks (as separate sets) from zero to give them unique labels. For log blocks, scavenging starts from the head of the log (block 0) and works through each log block in turn, discarding obsolete log entries as discussed above, and writing the valid entries to a new sequence of log blocks. Thus, sweeping through the log will partition the log into the set of blocks that have been swept and the set of those about to be swept. Similarly, when scavenging data blocks, each block is copied to a transfer unit and then the old block is erased and freed. Appending a three-value counter to the sequence numbers will therefore suffice to tag the different generations of the log unambiguously: a member of the 'swept' partition will have a counter value one greater than those that are 'unswept'. The same method can distinguish old and new instances of a data block: the new instance will have a value one greater than the old. The implementation actually uses two bits (four values) to allow shifts and masking to be used instead of more expensive division and remainder operations.

On NOR flash implementations with a granularity of a few bytes or a few words for each write, each log entry can be written immediately. On NAND flash, owing to the restrictions on (sub)page writes, that cannot be done, and instead log entries must be buffered to the next (sub)page boundary. If that buffer cannot be flushed during a shutdown, on restart the file system state will be made consistent by the actions above, but some data will be lost. When the buffer is flushed to NAND flash, it will be rounded up to the nearest write boundary, so a small amount of space in the log will be unusable until scavenged.

4. Wear balancing

The NAND flash implementation happens to keep an erase count in the auxiliary area of the first page of each block. Physical blocks are allocated round-robin, though, given the circular search for free space. Old, stable data and log pages tend to remain where they are over time, and as in several other flash implementations, that must be countered by periodically moving them, typically by selecting a block for moving that would otherwise be skipped during the circular search. That might obviate the need for erase counts.

5. Comparisons

JFFS and JFFS2 are journalling file systems for Linux. They are quite tightly tied to traditional Unix file system implementation. Woodhouse[3] discusses disadvantages of JFFS that led to the development of JFFS2. For instance, storing all the data in the log forces can cause excessive block movement during scavenging, because the log blocks are swept in order, and blocks early in the log must move before free space later in the log can be reclaimed. JFFS2 uses a more complex set of data structures in flash to reduce that. JFFS2 has been made to work with NAND flash, but was not designed with it in mind.

Perceived problems with JFFS2 when used with NAND flash provoked the design of an overtly NAND-oriented flash file system by Toby Churchill Limited.[5] It is not log-structured; instead the file system is partitioned into file header blocks and data blocks. The former contain the file system metadata. The auxiliary regions in pages in the data blocks identify the file header that contains them. Implementation is not yet complete.

A flash file system for Plan 9, by Bruce Ellis,[4] implements only a subset of the desired file system semantics, but it could of course be extended. Its design has not been documented (at least publicly), although the

source code is available. It is a compact, tidy implementation, but it represents the whole file system contents in a single log. Consequently it shares the disadvantage noted above for JFFS. It has only been used on NOR flash.

References

1. Mendel Rosenblum, John K Ousterhout, ‘‘The Design and Implementation of a Log-Structured File System,’’ in *ACM Transactions on Computer Systems* (February 1992).
2. Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, Carl Staelin, ‘‘An Implementation of a Log-structured File System for UNIX,’’ in *Proceedings of USENIX Winter 1993*, San Diego (1993).
3. David Woodhouse, *JFFS: The Journalling Flash File System*, Red Hat, Inc.
4. Bruce Ellis, *Journalling Flash File System (published code)*, Lucent Technologies Inc (2001).
5. Charles Manning, *Yet Another Flash File System*, Toby Churchill Limited, http://www.aleph1.co.uk/armlinux/projects/yaffs (2001).
6. Dines Björner (editor), *Formal Specification and Software Development*, Prentice-Hall (1982).

Appendix A: adding an extent

It is convenient to manage the data as extents, represented by the following Limbo adt:

```
Extent: adt
{
        min:    int;    # in file
        max:    int;    # in file
        flash:  int;    # logical flash file address
};
```

The algorithm for inserting an `Extent` requires careful case analysis. It is therefore reproduced in its Limbo form here:

```
#
# extents
#
insertextent(a: array of Extent, new: Extent): array of Extent
{
        # initially a's extents are non-empty, disjoint and sorted
        for(i := 0; i < len a; i++){
                old := a[i];
                if(new.max <= old.min)
                        break;
                if(old.min < new.max && new.min < old.max){     # they intersect
                        if(new.min <= old.min){
                                if(new.max < old.max){
                                        # retain tail of old
                                        a[i] = Extent(new.max, old.max, old.flash+(new.max-old.mi
                                        break;
                                }
                                # new.min ≤ old.min < old.max ≤ new.max ⇒ new completely covers
                                a = delete(a, i--);
                        }else{  # new.min > old.min
                                a[i] = Extent(old.min, new.min, old.flash);             # retain
                                if(new.max < old.max){  # new inside old, splitting it
                                        a = insert(a, i+1,
                                                Extent(new.max, old.max, old.flash+(new.max-old.m
                                        i++;    # insert before
                                        break;
                                }
                                # old.max <= new.max ⇒ new covers tail of old
                        }
                }
        }
        # now (a+new)'s extents are disjoint, and i is the position to insert new (in sort)
        return insert(a, i, new);
}

insert(a: array of Extent, i: int, e: Extent): array of Extent
{
        b := array[len a+1] of Extent;
        if(i > 0)
                b[0:] = a[0:i];
        b[i] = e;
        b[i+1:] = a[i:];
        return b;
}

delete(a: array of Extent, i: int): array of Extent
{
        b := array[len a-1] of Extent;
        b[0:] = a[0:i];
        if(i < len b)
                b[i:] = a[i+1:];
        return b;
}
```