Anatomy of Linux flash file systems

Options and architectures

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Consultant Engineer
Emulex Corp.

20 May 2008

You've probably heard of Journaling Flash File System (JFFS) and Yet Another Flash File System (YAFFS), but do you know what it means to have a file system that assumes an underlying flash device? This article introduces you to flash file systems for Linux®, explores how they care for their underlying consumable devices (flash parts) through wear leveling, and identifies the various flash file systems available along with their fundamental designs.

Solid-state drives are all the rage these days, but embedded systems have used solid-state devices for storage for quite some time. You'll find flash file systems used in personal digital assistants (PDAs), cellphones, MP3 players, digital cameras, USB flash drives (UFDs), and even laptop computers. In many cases, the file systems for commercial devices can be custom and proprietary, but they face the same challenges discussed below.

More in Tim's Anatomy of... series on developerWorks

- Anatomy of Security-Enhanced Linux (SELinux)
- Anatomy of real-time Linux architectures
- Anatomy of the Linux SCSI subsystem
- Anatomy of the Linux file system
- Anatomy of the Linux networking stack
- Anatomy of the Linux kernel

- Anatomy of the Linux slab allocator
- Anatomy of Linux synchronization methods
- All of Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

Flash-based file systems come in a variety of forms. This article explores a couple of the read-only file systems and also reviews the various read/write file systems available today and how they work. But first, let's explore the flash devices and the challenges that they introduce.

Flash memory technologies

Flash memory, which can come in several different technologies, is *non-volatile memory,* which means that its contents persist after its source of power is removed. For a great history of flash memory devices, see Resources.

Two of the most common types of flash devices are defined by their respective technologies: NOR and NAND. NOR-based flash is the older technology that supported high read performance at the expense of smaller capacities. NAND flash offers higher capacities with significantly faster write and erase performance. NAND also requires a much more complicated input/output (I/O) interface.

Flash parts are commonly divided into *partitions,* which allows multiple operations to occur simultaneously (erasing one partition while reading from another). Partitions are further divided into *blocks* (commonly 64KB or 128KB in size). Firmware that uses the partitions can further apply unique segmenting to the blocks—for example, 512-byte segments within a block, not including metadata.

Flash devices exhibit a common constraint that requires device management when compared to other storage devices such as RAM disks. The only Write operation permitted on a flash memory device is to change a bit from a one to a zero. If the reverse operation is needed, then the block must be erased (to reset all bits to the one state). This means that other valid data within the block must be moved for it to persist. NOR flash memory can typically be programmed a byte at a time, whereas NAND flash memory must be programmed in multi-byte bursts (typically, 512 bytes).

The process of erasing a block differs between the two memory types. Each requires a special Erase operation that covers an entire block of the flash memory. NOR technology requires a precursor step to clear all values to zero before the Erase operation can begin. An *Erase* is a special operation with the flash device and can be time-consuming. Erasing is an electrical operation that drains the electrons from each cell in an entire block.

NOR flash devices typically require seconds for the Erase operation, whereas a NAND device can erase in milliseconds. A key characteristic of flash devices is the number of Erase operations that can be performed. In a NOR device, each block in the flash memory can be erased up to 100,000 times. NAND flash memories can be erased up to one million times.

Flash memory challenges

In addition to and as a result of the constraints explored in the previous section, managing flash devices presents several challenges. The three most important are garbage collection, managing bad blocks, and wear leveling.

Garbage collection

Garbage collection is the process of reclaiming invalid blocks (those that contain some amount of invalid data). Reclamation involves moving the valid data to a new block, and then erasing the invalid block to make it available. This process is commonly done in the background or as needed, if the file system is low on available space.

Managing bad blocks

Over time, flash devices can develop bad blocks through use and can even ship from the manufacturer with blocks that are bad and cannot be used. You can detect the presence of back blocks from a failed flash operation (such as an Erase) or an invalid Write operation (discovered through an invalid Error Correction Code, or ECC).

After bad blocks have been identified, they are marked within the flash itself in a bad block table. How this is done is device-dependent but can be implemented with a separate set of reserved blocks managed separately from normal data blocks. The process of handling bad blocks—whether they ship with the device or appear over time—is called *bad block management.* In some cases, this functionality is implemented in hardware by an internal microcontroller and is therefore transparent to the upper-level file system.

Wear leveling

Recall that flash devices are consumable parts: You can perform a finite number of Erase cycles on each block before the block becomes bad (and must therefore be tagged by bad block management). To maximize the life of the flash, wear-leveling algorithms are provided. Wear leveling comes in two varieties: *dynamic wear leveling* and *static wear leveling.*

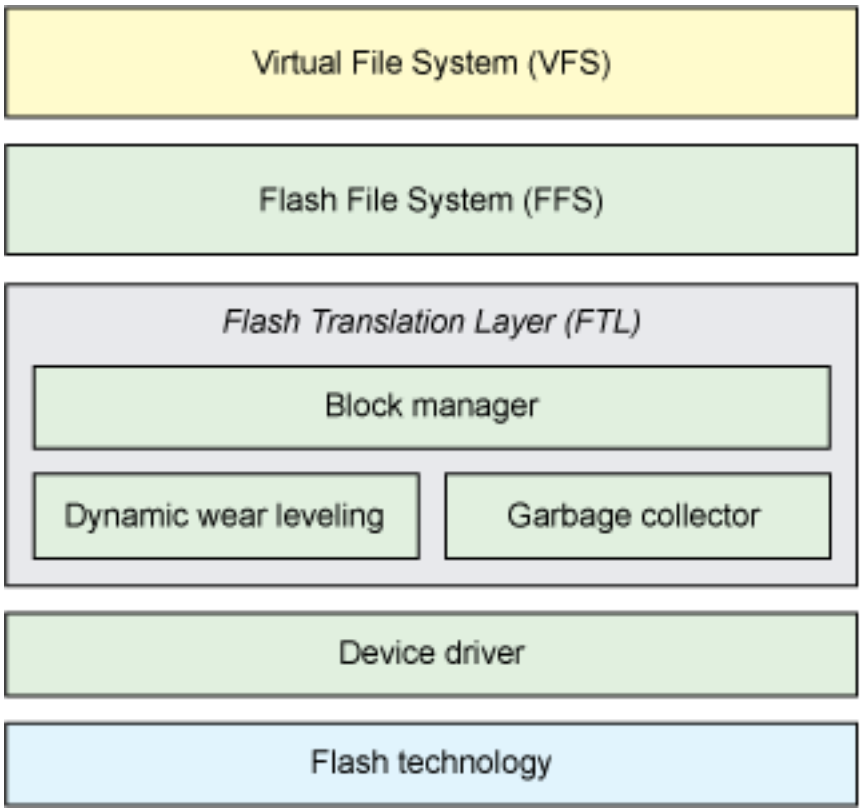
Dynamic wear leveling addresses the problem of a limited number of Erase cycles

for a given block. Rather than randomly using blocks as they are available, dynamic wear-leveling algorithms attempt to evenly distribute the use of blocks so that each gets uniform use. Static wear-leveling algorithms address an even more interesting problem. In addition to a maximum number of Erase cycles, certain flash devices suffer from a maximum number of Read cycles between Erase cycles. This means that if data sits for too long in a block and is read too many times, the data can dissipate and result in data loss. Static wear-leveling algorithms address this by periodically moving stale data to new blocks.

System architecture

So far, I've explored flash devices and their fundamental challenges. Now, look at how these pieces come together as part of a layered architecture (see Figure 1). At the top is the virtual file system (VFS), which presents a common interface to higher-level applications. The VFS is followed by the flash file system, which will be covered in the next section. Next is the Flash Translation Layer (FTL), which provides for overall management of the flash device, including allocation of blocks from the underlying flash device as well as address translation, dynamic wear leveling, and garbage collection. In some flash devices, a portion of the FTL can be implemented in hardware.

Figure 1. Basic architecture of a flash system



The Linux kernel uses the Memory Technology Device (MTD) interface, which is a generic interface for flash devices. The MTD can automatically detect the width of the flash device bus and the number of devices necessary for implementing the bus width.

Flash file systems

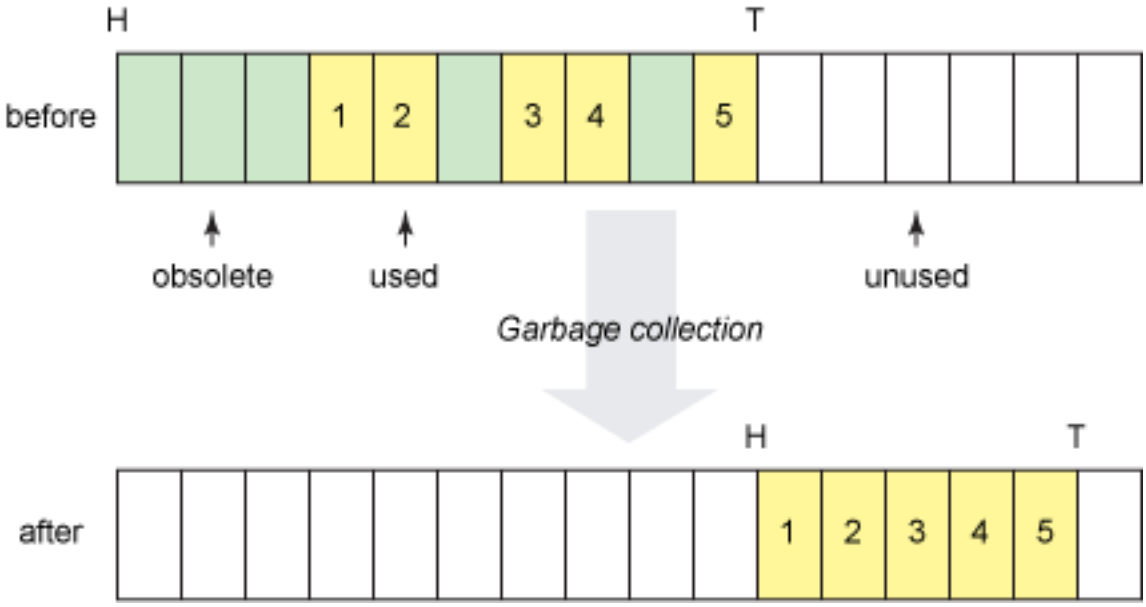
Several flash file systems are available for Linux. The next sections explain the design and advantages of each.

Journaling Flash File System

One of the earliest flash file systems for Linux is called the *Journaling Flash File System.* JFFS is a log-structured file system that was designed for NOR flash devices. It was unique and addressed a variety of problems with flash devices, but it created another.

JFFS viewed the flash device as a circular log of blocks. Data written to the flash is written to the tail, and blocks at the head are reclaimed. The space between the tail and head is free space; when this space becomes low, the garbage collector is executed. The garbage collector moves valid blocks to the tail of the log, skips invalid or obsolete blocks, and erases them (see Figure 2). The result is a file system that is automatically wear leveled both statically and dynamically. The fundamental problem with this architecture is that the flash device is erased too often (instead of an optimal erase strategy), which wears the device out too quickly.

Figure 2. Circular log before and after garbage collection



When a JFFS is mounted, the structural details are read into memory, which can be

slow at mount-time and consume more memory than desired.

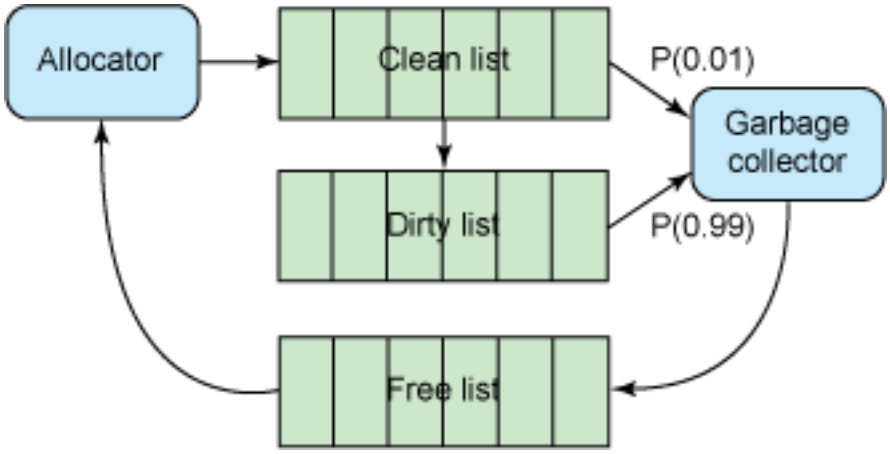
Journaling Flash File System 2

Although JFFS was very useful in its time, its wear-leveling algorithm tended to shorten the life of NOR flash devices. The result was a redesign of the underlying algorithm to remove the circular log. The JFFS2 algorithm was designed for NAND flash devices and also includes improved performance with compression.

In JFFS2, each block in the flash is treated independently. JFFS2 maintains block lists to sufficiently wear-level the device. The clean list represents blocks on the device that are full of valid nodes. The dirty list contains blocks with at least one obsoleted node. Finally, the free list represents the blocks that have been erased and are available for use.

The garbage collection algorithm can then intelligently decide what to reclaim in a reasonable way. Currently, the algorithm probabilistically selects from the clean or dirty list. The dirty list is selected 99 percent of the time to reclaim blocks (moving the valid contents to another block), and the clean list is selected 1 percent of the time (simply moving the contents to a new block). In both cases, the selected block is erased and placed on the free list (see Figure 3). This allows the garbage collector to re-use blocks that are obsoleted (or partially so) but still move data around the flash to support static wear leveling.

Figure 3. Block management and garbage collection in JFFS2



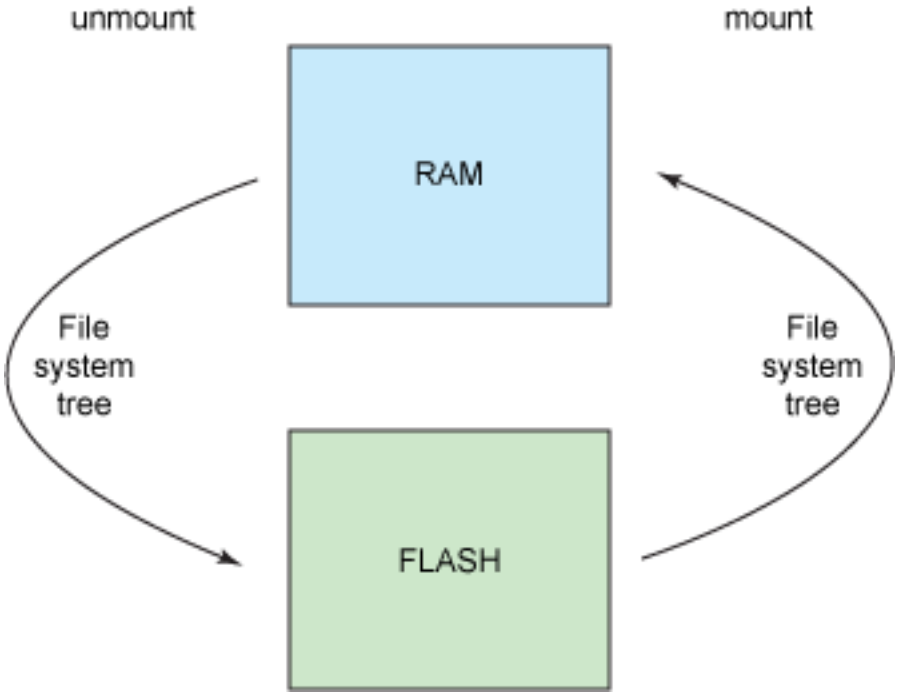
Yet Another Flash File System

YAFFS is another flash file system developed for NAND flash. The initial version (YAFFS) supported flash devices with 512-byte pages, but the newer version (YAFFS2) supports newer devices with larger page sizes and greater Write constraints.

In most flash file systems, obsolete blocks are marked as such, but YAFFS2 additionally marks blocks with monotonically increasing sequence numbers. When

the file system is scanned at mount time, the valid inodes can be quickly identified. YAFFS also maintains trees in RAM to represent the block structure of the flash device, including fast mounting through *checkpointing*—the process of saving the RAM tree structure to the flash device on a normal unmount so that it can be quickly read and restored to RAM at mount time (see Figure 4). Mount-time performance is a great advantage of YAFFS2 over other flash file systems.

Figure 4. Block management and garbage collection in YAFFS2



Read-only compressed file systems

In some embedded systems, there's no need to provide a mutable file system: An immutable one will suffice. Linux supports a variety of read-only file systems, two of the most useful are *cramfs* and *SquashFS.*

Cramfs

The cramfs file system is a compressed read-only Linux file system that can exist within flash devices. The primary characteristics of cramfs are that it is both simple and space-efficient. This file system is used in small-footprint embedded designs.

While cramfs metadata is not compressed, cramfs uses zlib compression on a per-page basis to allow random page access (pages are decompressed upon access).

You can play with cramfs using the `mkcramfs` utility and the loopback device.

SquashFS

SquashFS is another compressed read-only Linux file system that is useful within flash devices. You'll also find SquashFS in numerous Live CD Linux distributions. In addition to supporting zlib for compression, SquashFS uses Lembel-Ziv-Markov chain Algorithm (LZMA) for improved compression and speed.

Like cramfs, you can use SquashFS on a standard Linux system with `mksquashfs` and the loopback device.

Going further

Like most of open source, software continues to evolve, and new flash file systems are under development. An interesting alternative still in development is LogFS, which includes some very novel ideas. For example, LogFS maintains a tree structure on the flash device itself so that the mount times are similar to traditional file systems, such as ext2. It also uses a wandering tree for garbage collection (a form of B+tree). What makes LogFS particularly interesting, however, is that it is very scalable and can support large flash parts.

With the growing popularity of flash file systems, you'll see a considerable amount of research being applied toward them. LogFS is one example, but other options, such as UbiFS, are also growing. Flash file systems are interesting architecturally and will continue to be a source of innovation in the future.

Resources

Learn

- See Wikipedia's overview of flash memory technologies and list of file systems, including disk-based file systems, distributed file systems, and special-purpose file systems (including solid-state media file systems).
- Read more about NAND vs. NOR flash technology.
- Tim's "Anatomy of the Linux file system" (developerWorks, Oct 2007) introduces the VFS, its major structures, and its architecture. It also provides an introduction to file systems and illustrates how Linux can support so many file systems concurrently.
- Read how JFFS and its successor, JFFS2 (PDF file) take different approaches to flash device management.
- Compare YAFFS to other popular flash file systems.
- LogFS and UbiFS are two new flash file systems that solve many problems associated with current flash file systems.
- The MTD is a generic subsystem for memory devices, such as flash parts. It provides a generic interface between lower-level hardware drivers and the upper layers of the file system.
- Cramfs and SquashFS are read-only compressed file systems for Linux. Both were designed for memory-constrained embedded systems, but you'll also find SquashFS used in Live CD distributions. Sometimes, you'll find these used with UnionFS, which implements a union mount for a file system (overlaying one file system on another). SquashFS has the advantage of better compression and performance using LZMA and a set of patches.
- Read all of Tim's *Anatomy of...* articles on developerWorks.
- Read all of Tim's Linux articles on developerWorks.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

M. Tim Jones
M. Tim Jones is an embedded software engineer and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.

Trademarks