

## EMBEDDED IN THIN SLICES



# Embedded File Systems (Part 2)

## File System Integrity

The first part of this article series introduced the topic of Linux embedded file systems. This article discusses file system integrity across power outages and system crashes and describes some integrity assurance solutions.

By Bob Japenga (USA)

In 1975, I was working on a PDP-11 system that used the RT-11 file system. I noticed that the disk access became slower over time. You could issue the `SQUeeze` command to the file system (all 2.5 MB of it), which would resolve this problem. Of course, the command took almost a day to complete.

PCs had a similar problem caused by disk fragmentation that prompted many people to buy a new PC when their old one moved at a sluggish pace. They thought it was worn out. Instead the disk was just fragmented. Disk fragmentation happens any time one file's data is located in noncontiguous physical locations on the disk.

Once when my dentist complained about how slow his PC had become, he asked me for suggestions for a new PC. I suggested he use a defrag program instead of replacing his PC. He was very grateful—but he still drills me now and again.

system (after OS 10.2) nor Linux systems need defragging. Both have file structures that do not require this operation.

I bring this up because most people are familiar with older Windows systems' defragging requirements. Most don't know that this is not inherent in all file systems but is due to a design decision about the file system's structure. The file system's structure can affect things such as robustness over time, excessive disk space usage, and so forth.

This brings us to another feature critical to embedded systems: robustness across unexpected power outages or system crashes. If you are using a file system in an embedded system, you don't want it to ever get corrupted across a power outage or crash! Some file systems prevent this from happening by requiring an orderly shutdown. With embedded systems, you don't want to require an orderly shutdown to be performed before you power down.

One embedded Linux single-board computer (SBC) provider continues to provide misinformation about this on its website. It insists that embedded systems must provide an orderly Linux shutdown to prevent data corruption. Really? This depends on what Linux file system you are using. Thus, it is important for you as the designer to know exactly what your file system can and cannot do. Can it be powered down 10,000 times without corrupting the data? One of the top RTOS developers didn't do this on a system we designed and it required us to retrofit thousands of devices already in the field with a battery backup to prevent unrecoverable data corruption following inadvertent loss of

---

*"Your file system's architecture can affect whether or not a file system's integrity can be guaranteed across every power-down scenario."*

---

### STRUCTURE MATTERS

Both problems were caused by the file system's architecture. File systems have a physical disk structure that defines where the data is stored in the physical media (hard drive, floppy, USB stick, or flash drive). The logical disk structure defines how files are stored on the disk. The way in which the design maps the logical disk structure to the physical disk structure dramatically affects any file system's performance and reliability. For example, neither the Macintosh file

power.

In the same way smart file system architecture can eliminate the need for defragging, your file system's architecture can affect whether or not a file system's integrity can be guaranteed across every power-down scenario. Let's start by looking at the problem.

## THE POWER-DOWN PROBLEM

Early on, when using SD cards and compact flash in cameras and other devices, many users discovered that the inexpensive cards could get corrupted when removing the card while writing to it. When the cards became corrupted, you could lose all of your data on the card.

Even today, my new Nikon camera has strict warnings: *Do not remove the card while the access light is on.* If you ever had an SD card, compact flash, or memory stick become corrupted and lose all of its data (your pictures, songs, etc.), you know how frustrating that can be.

Imagine putting this sticker on the embedded medical device we designed to go inside the human body: *"Don't remove power without first performing an orderly shutdown."* They would probably call for a medical orderly after they shut me down. Embedded systems usually cannot control when power is going to be removed. And every system cannot be held up long enough to provide time to perform an orderly shutdown.

If all the information about a file could be atomically written to one location in the file system, powering down while writing to that one location would be acceptable. Only the most recent data would be lost. Other parts of the file system would not be corrupted.

But real file systems don't keep all of the information about a file in one place. That would make accessing the file very time consuming. Imagine having to look across every block of a terabyte drive trying to find your unique file. As a minimum, file systems usually create other information about the file that they store in another place.

One example of this is directory information, which tells the system where the file is located. This makes finding your file faster. File systems also want to store other information such as who has permission to read, write, delete, and append the file; who owns the file; when the file was last modified; when the file was created; when the file was last accessed; and pointers to the media's physical blocks (hard drive or flash) where the file is kept. This is called metadata. So information about most files is contained in at least two locations: the directory and the file itself.

Therein lies the problem. If you write the data successfully and then start updating the directory when the power goes down, you run the risk of corrupting the directory entries for other files. This is unacceptable for embedded systems. You want a file system that can go through millions of shutdowns during file writes and never corrupt either the existing file being written or any other file in the system.

## THE SOLUTION

Journaling file systems came to the rescue. The idea for such architecture was conceived in the 1980s. IBM released the first journaling file system in 1990, appropriately named JFS (i.e., journaling file system). The basic concept is that the file system creates a journal entry of all changes that are going to be made when a file is written or is changed. Thus, in case of an incomplete operation, it can quickly and easily back out the last change. If the power goes down or something causes the software to crash, the file system will not be corrupted and it can quickly recover. The file system can read the journal to recover the system to the point it was at just before the write was commanded.

Some call journaling file systems log-structured file systems since the difference between keeping a log and keeping a journal is pretty minute. It is beyond the scope of this thin slice, but for argument's sake, let's consider log-structured file systems of the same ilk as journaling file systems.

## FLASH-BASED JFS ON LINUX

An embedded systems designer using Linux has several choices to make when selecting a file system. And the implications can be significant. Explanations of some of the journaling file systems we have used follows:

**JFFS2**—This has been the industry benchmark for flash-based journaling file systems since it was the first one introduced into the mainstream Linux distribution. JFFS2 has compression built into its core. It also has wear leveling built into the structure.

We have worked mostly with JFFS2. It has several quirks we have discovered over time. One is that, because of compression, you cannot tell exactly how much disk space you have left as you run low. For example, if there is 1 MB of raw disk space left and you

---

*"If you write the data successfully and then start updating the directory when the power goes down, you run the risk of corrupting the directory entries for other files. This is unacceptable for embedded systems."*

---

## ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at [rjapenga@microtoolsinc.com](mailto:rjapenga@microtoolsinc.com).

have 2 MB of data to write, can you commit the write? You don't know unless you know the esoteric compression algorithm efficiency on the particular data you want to write. We have worked around this algorithmically but it is nontrivial.

In addition, it can have a fairly long boot time on large (i.e., greater than 32 MB) flash drives. It also uses significantly more RAM than the other options. For us, JFFS2's primary disadvantage is with larger flash drives. We are not convinced it works well on systems above 64 MB.

**YAFFS2**—In 1973, a friend of mine kept going on and on about YACC (yet another compiler compiler). I didn't even know why you needed a compiler to compile compiler code! Around 2001, the YAFFS2 creators reached back into that bit of history to create YAFFS (yet another flash file system).

We are currently using YAFFS2 on one system. Under our tests, it has proven to be very robust across thousands of power failures occurring during disk writes. According to the Yaffs website, it has been crash tested hundreds of thousands of times. Due to its architecture, YAFFS2 boots up much more quickly on large drives than JFFS2. YAFFS2 takes significantly longer to remove a file than the other options. We have very limited experience with quantities of these in the field so talk to me in a few years. We "chose" it because it came with the SBC we were using.

**UBIFS**—Considered the successor to JFFS2 because it contains journaling and compression, we seamlessly switched to UBIFS once on a project when JFFS2 was taking too long to boot. It reduced our boot time by 30 s! In most benchmark tests, it is faster than the other file systems in reads and writes and average on file deletions. We have had no problems using UBIFS. We have hundreds of thousands of JFFS2 systems in the field and comparatively few UBIFS. Again,

talk to me in a few years

**LogFS**—In 2010, yet another flash file system (LogFS) was introduced into the mainstream Linux distribution. One of the problems with JFFS2 is that it keeps the inode tree (think of this as metadata handling where the logical blocks map to the physical blocks) in RAM so it must create this every time you boot. LogFS keeps this on disk. Clearly this creates some performance degradation, but we do not have any experience with this file system.


## WHICH JFS TO CHOOSE?

Some choices are made for you with the hardware. Not all file systems work with all the different kinds of flash. Sometimes you design around a SBC with a flash file system already installed. Yes you could change it—but do you need to?

JFFS2 and UBIFS have built-in compression, which may drive your decision if you are concerned about how little disk space you have. Some are of the school that newer is better. Some like to stick with what they know and has proven robust. We have had just enough problems with JFFS2 that I tend to favor working with a later design (e.g., UBIFS). Benchmarks on UBIFS look very good (see Resources for more information).

## CHOICES, CHOICES

As embedded systems designers, we sometimes wish we could have fewer choices in choosing a processor, a RTOS, a file system, a development tool chain, and so forth. But the fact remains that we do have many choices.

Understanding the implications of the choices will help us create more robust systems. Choosing a flash file system for your embedded Linux system is important. Hopefully I have introduced you to some of the options so that you can take this beyond thin slices. 

## RESOURCES

eLinux.org, "Flash Filesystem Benchmarks 3.1."

Yaffs, open-source file system, [www.yaffs.net](http://www.yaffs.net).



