

Guidelines for Creating Robust Embedded Systems

Part 6 – Handling Out of Disk Space

September 2009

By Bob Japenga

Scope

This is Part 6 of a white paper intended to provide a brief overview of lessons learned in creating robust systems over the past 35 years of embedded systems development (starting with the Intel 4040). This part will address handling out of disk space conditions.

Gracefully Handling Out of Disk Space Conditions

In the mid nineties, we had our first Linux system up and running. George, who joined us when he was in high school 3 years earlier, installed the system. We used it for our email and for our web server. After several months of operation, our email and web server shut down. We discovered that our log files had eaten 90% of the 300 meg hard drive (which we thought was large). A classic case of how not to gracefully handle out of disk conditions because it leads to our first three guidelines:

1. **Don't run out of disk space.** Fully understand anything that cumulatively uses disk space and set limits on it. Linux provides a great feature for doing this with syslogd. However we cut our teeth on non-Linux systems.
2. **Understand every process that generates cumulative data in your system –** Log files aren't the only things that grow. Trash boxes, https certificates, mail servers (outgoing and incoming), virtual memory, and virus definition files are all things we are including in embedded systems that can grow. Understand them and then apply guideline 1.
3. **Have a plan for when the impossible happens.** When you do the impossible (run out of disk space), start shedding less essential disk space to automatically recover.

Okay – so you followed our first three guidelines. You're done, right? Well, only if you trust perfectly your finely honed software design skills. That leads directly to another of MicroTools' axioms – "Imperfection reigns." Our job is to vigilantly root out these imperfections in our perfect designs.

One of the ways to do that during design brings us to our fourth guideline:

4. **Do not assume that every write to disk succeeded.** Check for out of disk space errors.

So – now you designed the system to never run out of disk space. You have always checked for errors coming back from the file system. But you have an error. How do

you gracefully handle the out of disk condition. Write the error to the error log! Whoops – there is no disk space. Like out of memory conditions, you must plan ahead of time, how you will handle an out of disk space condition (Guideline #3). Every system is different. You know what files you can get rid of. Make a priority of what you can dump.

The final guideline I would offer is:

5. **Provide some backup mechanism if all previous efforts have failed** – perhaps a small backup kernel and read only file system that allows the system to recover. Or re-boot the existing kernel in some sort of “safe-mode” that prevents the system from becoming a “brick” and allows some kind of maintenance or diagnostics to be run on the system.

Compressed File Systems

Finally, we need to talk about out of disk space conditions when using a compressed flash file system like JFFS2. First we need a little background on how flash file systems work.¹

How Flash File Systems Work

Because Flash memory can be written to in bytes but only erased in blocks, any flash file system must maintain a list of available blocks (erased) and blocks that were used but no longer needed and not yet erased (dirty). Furthermore, since writing is fairly quick and erasing is comparatively long, erasing is often done in the background (and with some systems only when needed).

So with a file system like JFFS2, when you are out of disk space, the file system can start erasing its dirty blocks and start making them available. This can have a serious effect on performance.

More critically, because of compression, there is no practical way to tell how much usable free space is left on a device since this depends both on how well additional data can be compressed, and the writing sequence.² This can result in robust code that worked on a non-compressed systems to fail because you will check to see if there is enough space before you commit to write the data and you cannot tell. The only hope for you in this situation is rigorous enforcement of guideline 3 (checking every write).

Where Problems Can Occur with Compressed File Systems

Now you have worked through all of these complications and you know that you limit your log file size to 20 meg on a system that has 100 meg free. Your logic is impeccable and has worked great on your network file system (nfs). Strangely you start getting reports from SQA that after stress testing you are running out of disk space and yet your pruning algorithms aren't kicking in. The problem is that your log files are

written out in 200 byte chunks. But each chunk might be taking 1024 bytes because of a number of factors. One is what is called “negative compression rate” where the data takes more space compressed than uncompressed. The second factor happens because of the flash algorithms. If the flash is divided into 1024 byte “sectors,” each 200 byte chunk takes up 1024. When appending to a file (as in a log file), JFFS2 doesn’t uncompress the previously written data, append the new data, recompress and re-write the file. That would be terribly inefficient. Instead it compresses the new data and “journals” it on to the last data by writing to a new “sector.” Your 20 meg log file actually is using 100 meg of physical space.

We have solved this by “caching” our writes (using a flash cache) until they reach an optimal size (depends on the flash disk geometry and the compression algorithm) and then appending the “cache” to the existing log.

Another trick we do with backup log files is that we set a lower limit and keep one or two back versions of the log files. The current log file is inefficient, but the backup log file is copied in a single write which provides the optimal compression and disk storage size.

In [Part 7](#) we will discuss how to create a robust watchdog.

¹ <http://sourceware.org/jffs2/jffs2.pdf> is a good white paper on JFFS2.

² <http://en.wikipedia.org/wiki/JFFS2>