EMBEDDED IN THIN SLICES

Embedded File Systems (Part 4)

COLUMNS

Specialized Linux Tools

Several types of file systems are available for Linux. This article discusses compressed read-only file systems (CRAMFS), RAM file systems, network file systems (NFS), and the Samba networking protocol.

By Bob Japenga (US)

To help embedded systems designers achieve their design goals, Linux offers a variety of file systems that require only a moderate amount of effort to implement. This article introduces several file systems and a networking protocol and describes how and when to use each one.

WHAT IS A CRAMFS?

While pursuing my Master's degree in 1973, a professor made a statement I will never forget: "Memory prices are plunging so fast you will never have to worry about how much memory your designs use from now on." Obviously, this professor did not have a crystal ball and missed something that has been a design challenge in virtually every embedded system I have designed. Our systems demand more and more memory (or file space) and a compressed read-only file system (CRAMFS) can be a useful solution in some instances.

A CRAMFS is an open-source file system available for Linux. I am not sure where the CRAMFS gets its name. Perhaps it gets its name because a CRAMFS is one way to cram your file system into a smaller footprint. The files are compressed one page at a time using the built-in zlib compression to enable random access of all of the files. This makes CRAMFS relatively fast. The file metadata (e.g., information about when the file was created, read and write privileges, etc.) is not compressed but uses a more compact notation than is present in most file systems.

WHEN TO USE A CRAMFS

The primary reason my company has used a CRAMFS is to cut down on the flash memory used by the file system. The first embedded Linux system we worked on had 16 MB of RAM and 32 MB of flash. There was a systems-level requirement to provide a means for the system to recover should the primary partition become corrupt or fail to boot in any way (see Part 3 of this article series "Designing Robust Flash Memory Systems," *Circuit Cellar* 283, 2014, for more detail). We met this requirement by creating a backup partition that used a CRAMFS.

The backup partition's only function was to enable the box to recover from the corrupted primary partition. How we accomplished that is a discussion for another day, but we had very little flash memory to hold the Linux kernel, the file system, the boot-loader, and

the application. We were able to have the two file systems identical in file content, which made it easy to maintain. Using a CRAMFS enabled us to cut our backup file system space requirements in half.

A second feature of a CRAMFS is its read-only nature. Given that it is read-only, it does not require wear leveling. This keeps the overhead for using CRAMFS files very low. Due to the typical data retention of flash memory, this also means that for products that will be deployed for more than 10 years, you will need to rewrite the CRAMFS partition every three to five years.

The fact that the file system is read-only can make it difficult to deploy in all situations. For example, once our CRAMFS began using encryption keys that were uniquely calculated for each system one time at start-up (this took several minutes on a 200-mHz ARM9 processor) the CRAMFS became a problem. We could not store the calculated keys on the CRAMFS and the start-up delay was unacceptable. This forced us to come up with an alternate backup file system.

HOW TO MAKE A CRAMFS

As with the other file systems I have discussed, implementing a CRAMFS with Linux is simple and relatively painless. Linux provides a tool (mkfs) you can use to take any group of directories and files and create a CRAMFS image. Actually, mkfs is a wrapper for a variety of file system building tools. mkfs.cramfs is the tool used to create the CRAMFS image. Once written, the image is placed in its own partition.

WHAT ARE RAM FILE SYSTEMS?

Linux provides two types of RAM file systems: ramfs and tmpfs. Both are full-featured file systems that reside in RAM and are thus very fast and volatile (i.e., the data is not retained across power outages and system restarts).

When the systems are created with the mount command, you specify the ramfs size. However, it can grow in size to exceed that amount of RAM. Thus ramfs will enable you to use your entire RAM and not provide you any with warning that it is doing it. tmpfs does not enable you to write more than the space allocated at mount time. An error is returned when you try to use more than you have allocated. Another difference is that tmpfs uses swap space and can swap seldom used files out to a flash drive. ramfs does not use swapping. This difference is of little value to us since we disable swapping in our embedded systems.

WHEN TO USE RAM FILE SYSTEMS

In 1987, the Department of Defense began requiring the use of Ada in new embedded systems. Shortly thereafter I bought an Ada compiler for my IBM PC at work. Much to my surprise, the compiler came with a board to plug into the PC. The board added a RAM file system to my PC. To improve the speed of compiling from glacial to turtle-like, the compiler designers used a RAM file system to store all intermediate results.

Speed is one of the primary reasons to use a RAM file system. Disk writes are lightning fast when you have a RAM disk. We have used a RAM file system when we are recording a burst of high-speed data. In the background, we write the data out to flash.

A second reason to use a RAM file system is that it reduces the wear and tear on the flash file system, which has a limited write life. We make it a rule that all temporary files should be kept on the RAM disk. We also use it for temporary variables that are needed across threads/processes.

HOW TO MAKE A RAM FILE SYSTEM

Since tmpfs is built into Linux, you can simply mount a RAM drive with a command such as:

```
mount - o size=16G -t tmpfs my_
```

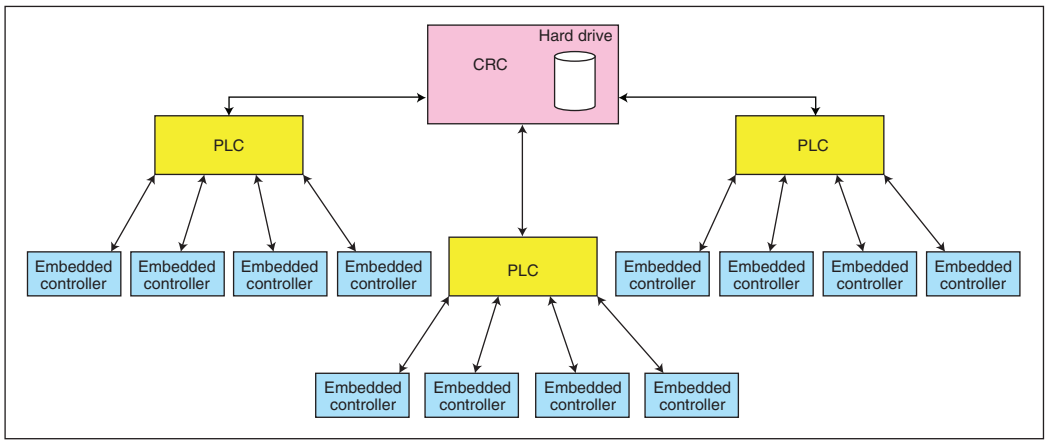

FIGURE 1
This layout shows the memory regions required to bring up one of my company's embedded Linux designs with no redundancy.

COLUMNS

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.

circuitcellar.com/ccmaterials

RESOURCES

FreeNFS, http://freenfs.sourceforge.net.

B. Japenga, "Embedded File Systems (Part 3): Designing Robust Flash Memory Systems," *Circuit Cellar* 283, 2014.

Samba, www.samba.org.

SOURCE

ARM9 Processor
ARM, Ltd. | www.arm.com

```
tmpfs /mnt/tmpfs
```

where the maximum size is 16 GB, the type of file system is `tmpfs`, the name is `my_tmpfs`, and the access point (device name) is `/mnt/tmpfs`. The size can also be specified in terms of percentage of your total physical RAM. tmpfs will only use the RAM that it needs up to the maximum specified. If you don't specify a maximum, it defaults to 50% of your total physical RAM.

ramfs is also built into Linux and is created with a command such as:

```
mount -o size=20m -t ramfs my_ramfs /mnt/ramfs
```

where the type is `ramfs`, the name is `my_ramfs` and the access point (device name) is `/mnt/ramfs`.

WHAT ARE NETWORK FILE SYSTEMS?

In the early 1990s I started working with a company that developed embedded controllers for machine control. These controllers had a user interface that consisted of a PC located on the factory floor. The company called this the production line console (PLC). The factory floor was hot, very dirty, and had a lot of vibration. The company had designed a control room console (CRC) that networked together several PLCs. The CRC was located in a clean and cool environment. The PLC and the CRC were running QNX and the PLC was diskless. The PLC booted from and stored all of its data on the CRC (see **Figure 1**).

This was my first exposure to a network file system (NFS). It was simple and easy to configure and worked flawlessly. The PLCs could only access their "file system." The CRC could access any PLC's files.

QNX was able to do this using the NFS protocol. NFS is a protocol developed initially by Sun Microsystems (which is now owned by Oracle). Early in its lifetime, Sun turned the specification into an open standard, which was quickly implemented in Unix and its derivatives (e.g., Linux and QNX).

WHEN TO USE AN NFS

One obvious usage of an NFS is for environments where a hard drive cannot easily survive, as shown in my earlier example. However, my example was before flash file systems became inexpensive and reliable so that is not a typical use for today.

Another use for an NFS would be to simplify software updates. All of the software could be placed in one central location. Each individual controller would obtain the new software once the central location was updated.

The major area in which we use NFS today is during software development. Even though flash file systems are fast and new versions of your code can be seamlessly written to flash, it can be time consuming. For example, you can use a flash memory stick over USB to update the flash file system on several of our designs. This is simple but can take anywhere from several seconds to minutes.

With an NFS, all of your development tools can be on a PC and you never have to transfer the target code to the target system. You use all of your PC tools to change the file on your PC, and when the embedded device boots up or the application is restarted, those changed files will be used on the device.

HOW TO MAKE AN NFS

An NFS is built into Linux and thus is pretty trivial to set up. The embedded system can mount an external drive using a combination of the Mount command and an entry in the file system table file (usually in /etc/fstab). The external drive requires an NFS driver that is easily available for Windows and is built into all Linux distributions.

WHAT IS SAMBA?

Although we don't like to admit it, many of us still have Windows machines on our desks and on our laptops. And many of us are attached to some good development tools on our Windows machines.

Samba is not exactly a file system but rather a file system/networking protocol that enables you to write to your embedded system's file system from your Windows machine as if it were a Windows file system.

WHEN TO USE SAMBA

Although I primarily see Samba, like an NFS, as a development tool, you could certainly use it in an environment where you needed to talk to your embedded device from a Windows machine. We have never had a need for this, but I can imagine it could be useful in certain scenarios. The Linux community documents a lot of Samba users for embedded Linux.

HOW TO MAKE SAMBA

To make a Samba, two daemons will need to be built for your processor: one (smbd) to handle file and printer sharing and one (nmbd) to handle NETBIOS to IP name services. Virtually all distributions of Linux will have the necessary make files and recipes for you to put Samba on your embedded device.

USEFUL EMBEDDED SYSTEMS TOOLS

You are now armed with some additional tools to create incredible embedded systems. Each of these specialized file systems has its

own niche. But you have to know what you can do to take advantage of them—even if you only know them—in thin slices.

My next article series will explore another dimension of embedded Linux.