Making a More Resilient File System

Raspberry Pi Ltd

2024-06-25: githash: 3e4dad9-clean

Colophon

© 2020-2023 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.)

This documentation is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).

build-date: 2024-06-25
build-version: githash: 3e4dad9-clean

Legal disclaimer notice

Document version history

Release	Date	Description
1.0	22 Dec 2021	Initial release
1.1	22 Jun 2024	Added section on pSLC for CMx

Scope of document

This document applies to the following Raspberry Pi products:

Pi 0			Pi 1		Pi 2		Pi 3	Pi 4	Pi 400	CM1	CM3	CM4	CM 5	Pico
0	W	H	A	B	A	B	B	All	All	All	All	All	All	All
*	*	*	*	*	*	*	*	*	*	*	*	*	*	

Introduction

Raspberry Pi Ltd devices are frequently used as data storage and monitoring devices, often in places where sudden power downs may occur. As with any computing device, power dropouts can cause storage corruption. This whitepaper provides some options on how to prevent data corruption under these and other circumstances by selecting appropriate file systems and setups to ensure data integrity.

This whitepaper assumes that the Raspberry Pi is running the Raspberry Pi (Linux) operating system (OS), and is fully up to date with the latest firmware and kernels.

What is data corruption and why does it occur?

Data corruption refers to unintended changes in computer data that occur during writing, reading, storage, transmission, or processing. In this document we are referring only to storage, rather than transmission or processing. Corruption can occur when a writing process is interrupted before it completes, in a way that prevents the write from being completed, for example if power is lost.

It is worthwhile at this point giving a quick introduction to how the Linux OS (and, by extension, Raspberry Pi OS), writes data to storage. Linux usually uses *write caches* to store data that is to be written to storage. These cache (temporarily store) the data in random access memory (RAM) until a certain predefined limit is reached, at which point all the outstanding writes to the storage medium are made in one transaction. These predefined limits can be time and/or size related. For example, data may be cached and only written to storage every five seconds, or only written out when a certain amount of data has accumulated. These schemes are used to improve performance: writing a large chunk of data in one go is faster than writing lots of small chunks of data.

However, if power is lost between data being stored in the cache and it being written out, that data is lost.

Other possible issues arise further down the writing process, during the physical writing of data to the storage medium. Once a piece of hardware (for example, the Secure Digital (SD) card interface) is told to write data, it still takes a finite time for that data to be physically stored. Again, if power failure happens during that extremely brief period, it’s possible for the data being written to become corrupted.

When shutting down a computer system, including the Raspberry Pi, best practice is to use the `shutdown` option. This will ensure that all cached data is written out, and that the hardware has had time to actually write the data to the storage medium.

The SD cards used by the majority of the Raspberry Pi range of devices are great as cheap hard drive replacements, but are susceptible to failure over time, depending on how they are being used. The flash memory used in SD cards has a limited write cycle lifetime, and as cards approach that limit they can become unreliable. Most SD cards use a procedure called *wear levelling* to make sure they last as long as possible, but in the end they can fail. This can be from months to years, depending on how much data has been written to, or (more importantly) erased from, the card. This lifetime can vary dramatically between cards. SD card failure is usually indicated by random file corruptions as parts of the SD card become unusable.

There are other ways for data to become corrupted, including, but not limited to, defective storage medium, bugs in the storage-writing software (drivers), or bugs in applications themselves.

For the purposes of this whitepaper, any process by which data loss can occur is defined as a `corruption event`.

What can cause a write operation?

Most applications do some sort of writing to storage, for example configuration information, database updates, and the like. Some of these files may even be temporary, i.e. only used while the program is running, and do not require to be maintained over a power cycle; however, they still result in writes to the storage medium.

Even if your application does not actually write any data, in the background Linux will constantly be making writes to the storage, mostly writing logging information.

Hardware solutions

Although not totally within the remit of this whitepaper, it is worth mentioning that preventing unexpected power downs is a commonly used and well-understood mitigation against data loss. Devices such as uninterruptible power supplies (UPSs) ensure that the power supply remains solid and, if power is lost to the UPS, while on battery power it can tell the computer system that power loss is imminent so that shutdown can proceed gracefully before the backup power supply runs out.

Because SD cards have a limited lifetime, it may be useful to have a replacement regime that ensures SD cards are replaced before they have a chance to reach end of life.

Robust file systems

There are various ways that a Raspberry Pi device can be hardened against corruption events. These vary in their ability to prevent corruption, with each action reducing the chance of it occurring.

Reducing writes

Simply reducing the amount of writing that your applications and the Linux OS do can have a beneficial effect. If you are doing lots of logging, then the chances of writes happening during a corruption event are increased. Decreasing logging in your application is down to the end user, but logging in Linux can also be reduced. This is especially relevant if you are using flash-based storage (e.g. eMMC, SD cards) due to their limited write life cycle.

Changing commit times

The commit time for a file system is the amount of time for which it caches data before it copies it all to storage. Increasing this time improves performance by batching up lots of writes, but can lead to data loss if there is a corruption event before the data is written. Reducing the commit time will mean less chance of a corruption event leading to data loss, although it does not prevent it completely.

To change the commit time for the main EXT4 file system on Raspberry Pi OS, you need to edit the `\etc\fstab` file which defines how file systems are mounted on startup.

```
$sudo nano /etc/fstab
```

Add the following to the EXT4 entry for the root file system:

```
commit=<commit time in seconds>
```

So, `fstab` may look something like this, where the commit time has been set to three seconds. The commit time will default to five seconds if not specifically set.

```
proc            /proc           proc    defaults          0       0
PARTUUID=50a7611a-01  /boot           vfat    defaults          0       2
PARTUUID=50a7611a-02  /               ext4    defaults,noatime,commit=3  0       1
```

Temporary file systems

If an application requires temporary file storage, i.e. data only used while the application is running and not required to be saved over a shutdown, then a good option to prevent physical writes to storage is to use a temporary file system, *tmpfs*. Because these file systems are RAM based (actually, in virtual memory), any data written to a tmpfs is never written to physical storage, and therefore doesn’t affect flash lifetimes, and cannot become damaged over a corruption event.

Creating one or more tmpfs locations requires editing the `/etc/fstab` file, which controls all the file systems under Raspberry Pi OS. The following example replaces the storage-based locations `/tmp` and `/var/log` with temporary file system locations. The second example, which replaces the standard logging folder, limits the overall size of the file system to 16MB.

```
tmpfs /tmp tmpfs defaults,noatime 0 0
tmpfs /var/log tmpfs defaults,noatime,size=16m 0 0
```

There is also a third-party script that helps set up logging to RAM, which can be found on GitHub. This has the additional feature of dumping the RAM-based logs to disk at a predefined interval.

Read-only root file systems

The *root file system* (rootfs) is the file system on the disk partition on which the root directory is located, and it is the file system on which all the other file systems are mounted as the system is booted up. On the Raspberry Pi it is `/`, and by default it is located on the SD card as a fully read/write EXT4 partition. There is also a `boot` folder, which is mounted as `/boot` and is a read/write FAT partition.

Making the rootfs *read ONLY* prevents any sort of write accesses to it, making it much more robust to corruption events. However, unless other actions are taken, this means nothing can write to the file system at all, so saving data of any sort from your application to the rootfs is disabled. If you need to store data from your application but want a read-only rootfs, a common technique is to add a USB memory stick or similar that is just for storing user data.

NOTE

If you are using a swap file when using a read-only file system, you will need to move the swap file to a read/write partition.

Overlay file system

An overlay file system (overlayfs) combines two file systems, an *upper* file system and a *lower* file system. When a name exists in both file systems, the object in the upper file system is visible while the object in the lower file system is either hidden or, in the case of directories, merged with the upper object.

Raspberry Pi provide an option in `raspi-config` to enable an overlayfs. This makes the rootfs (lower) read only, and creates a RAM-based upper file system. This gives a very similar result to the read-only file system, with all user changes being lost on reboot.

You can enable an overlayfs using either the command line `raspi-config` or using the desktop `Raspberry Pi Configuration` application on the Preferences menu.

There are also other implementations of overlayfs that can synchronise required changes from the upper to the lower file system at a predetermined schedule. For example, you might copy the contents of a user's home folder from upper to lower every twelve hours. This limits the write process to a very short space of time, meaning corruption is much less likely, but does mean that if power is lost before the synchronisation, any data generated since the last one is lost.

pSLC on Compute modules

The eMMC memory used on Raspberry Pi Compute Module devices is MLC (Multi-Level Cell), where each memory cell represents 2 bits. pSLC, or pseudo-Single Level Cell, is a type of NAND flash memory technology that can be enabled in compatible MLC storage devices, where each cell represents only 1 bit. It is designed to provide a balance between the performance and endurance of SLC flash and the cost-effectiveness and higher capacity of MLC flash.

pSLC has a higher write endurance than MLC because writing data to cells less frequently reduces wear. While MLC might offer around 3,000 to 10,000 write cycles, pSLC can achieve significantly higher numbers, approaching the endurance levels of SLC. This increased endurance translates to a longer lifespan for devices using pSLC technology compared to those using standard MLC.

MLC is more cost-effective than SLC memory, but whilst pSLC offers better performance and endurance than pure MLC, it does so at the expense of capacity. An MLC device configured for pSLC will have half the capacity (or less) it would have as a standard MLC device since each cell is only storing one bit instead of two or more.

Implementation details

pSLC is implemented on eMMC as an Enhanced User Area (also known as Enhanced storage). The actual implementation of the Enhanced User Area is not defined in the MMC standard but is usually pSLC.

- Enhanced User Area is a concept, whereas pSLC is an implementation.
- pSLC is one way of implementing Enhanced User Area.
- At the time of writing, the eMMC used on the Raspberry Pi Compute Modules implements the Enhanced User Area using pSLC.
- There is no need to configure the whole eMMC user area as an Enhanced User Area.
- Programming a memory region to be an Enhanced User Area is a **one-time operation**. That means it **cannot be undone**.

Turning it on

Linux provides a set of commands for manipulating the eMMC partitions in the `mmc-utils` package. Install a standard Linux OS to the CM device, and install the tools as follows:

```
sudo apt install mmc-utils
```

To get information about the eMMC (this command pipes into `less` as there is quite a lot of information to display):

```
sudo mmc extcsd read /dev/mmcblk0 | less
```

WARNING

The following operations are one-time - you can run them once and they cannot be undone. You should also run them before the Compute Module has been used, as they will erase all data. The capacity of the eMMC will be reduced to half the previous value.

The command used to turn on pSLC is `mmc enh_area_set`, which requires several parameters that tell it over how much memory area the pSLC is to be enabled. The following example uses the entire area. Please refer to the `mmc` command help (`man mmc`) for details on how to use a subset of the eMMC.

```
# Find out how big the eMMC is
MAXSIZEKB=$(sudo mmc extcsd read /dev/mmcblk0 | grep MAX_ENH_SIZE_MULT -A 1 | grep
-o '[0-9]\+ ')
# Set the whole area to be pSLC
sudo mmc enh_area set -y 0 $MAXSIZEKB /dev/mmcblk0
reboot -f
```

After the device reboots, you WILL need to reinstall the operating system, as enabling pSLC will erase the contents of the eMMC.

The Raspberry Pi CM Provisioner software has an option to set pSLC during the provisioning process. This can be found on GitHub at https://github.com/raspberrypi/cmprovision.

Off-device file systems / network booting

The Raspberry Pi is able to boot over a network connection, for example using the Network File System (NFS). This means that once the device has completed its first-stage boot, instead of loading its kernel and root file system from the SD card, it is loaded from a network server. Once running, all file operations act on the server and not the local SD card, which takes no further role in the proceedings.

Cloud solutions

Nowadays, many office tasks take place in the browser, with all data being stored online in the cloud. Keeping data storage off the SD card can obviously improve reliability, at the expense of needing an always-on connection to the internet, as well as possible charges from cloud providers.

The user can either use a full-blown Raspberry Pi OS installation, with the Raspberry Pi optimised browser, to access any of the cloud services from suppliers such as Google, Microsoft, Amazon, etc. An alternative is one of the *thin-client* providers, which replace Raspberry Pi OS with an OS/application that runs from resources stored on a central server instead of the SD card. Thin clients work by connecting remotely to a server-based computing environment where most applications, sensitive data, and memory are stored.

Conclusions

When the correct shutdown procedures are followed, the SD card storage of the Raspberry Pi is extremely reliable. This works well in the home or office environment where shutdown can be controlled, but when using Raspberry Pi devices in industrial use cases, or in areas with an unreliable power supply, extra precautions can improve reliability.

In short, the options for improving reliability can be listed as follows:

- Use a well-known, reliable SD card.
- Reduce writes using longer commit times, using temporary file systems, using an overlayfs, or similar.
- Use off-device storage such as network boot or cloud storage.
- Implement a regime to replace SD cards before they reach end of life.
- Use a UPS.

Raspberry Pi