文件系统FAQ

发布版本：1.0

作者邮箱：cmc@rock-chips.com

日期：2018.07

文件密级：公开资料

前言

概述

产品版本

芯片名称	内核版本
全系列	通用

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

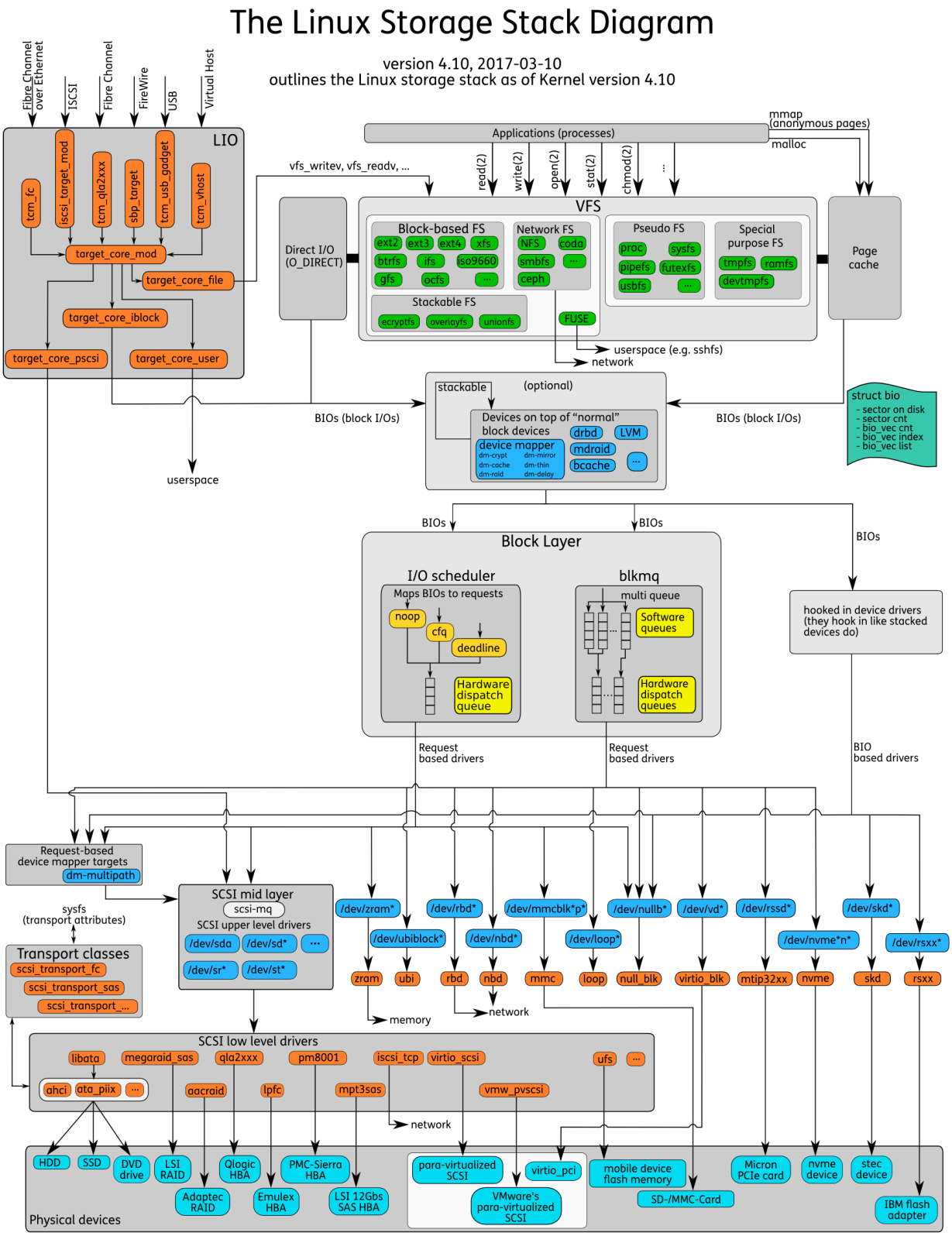
修订记录

日期	版本	作者	修改说明
2017-07-30	V1.0	陈谋春	

---

1 Linux Storage Stack

下图是一张Linux的存储栈的图解，通过它我们可以对Linux的存储子系统有个大致的了解：



我们从用户态发起一个系统调用，一般会进过这样的流程：VFS -> FS(ext4/f2fs) -> Block Layer -> Physical Devices。

2 系统调用与C库函数

fopen和open，read和fread，write和fwrite有什么区别，很多人都会弄混了，而这经常会带来一些问题。所以在这里理清他们的关系是很有必要的。

如上图所示，open/read/write是Linux提供的系统调用，用户态的程序只能通过这些接口来访问文件系统层。而fopen/fread/fwrite是C库提供的文件读写接口，其核心实现是基于open/read/write这一类的系统调用。说到这有些人可能会说，那我在写代码的时候应该用哪一套？其实都是可以的，主要看你的需求，C库函数的目的是为了方便编程，所以C库的文件接口提供了一些额外的处理，例如字符串和文本的处理，比如fpus/fgets，所以如果你在做文本数据的输入输出，无疑C库会是更好的选择。

Note: 绝大部分的C库都为文件接口提供一层缓存，所以你调用fwrite操作的时候，实际上数据是先放到这一层缓存的，在编程的时候必须注意，在下一节会重点说明。

3 Linux 数据回写

这一部分是问的最多的问题，很多对刚接触文件接口（包括前面的系统调用和C库函数）的人都会觉得很奇怪：为什么我fwrite/write函数已经返回了，此时掉电或重启后数据会丢失？问题的根源在于缓存的存在，由于存储设备属于低速设备，直接操作的话会有严重的延迟，所以通常会在DRAM上先缓存一部分数据。而DRAM是易失性存储设备，掉电数据就丢了，所以要确保数据固化就要把数据回写到存储设备上。

数据回写有两种方式：1.主动同步回写；2.异步后台回写；在了解数据如何回写前，有一点必须要注意，你的缓存有可能有好多层，所以你必须从上到下把每一层的缓存都刷出去（即写到下一层）。这一点我们下面会举几个具体的例子，先介绍怎么主动同步回写。

先来看一个系统调用的例子：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd;
ssize_t wr;
char szBuf[] = "hello world";

fd = open("/sdcard/test.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
wr = write(fd, szBuf, strlen(szBuf));
close(fd);
// 在这里掉电，test.txt将会是空文件，close并不能保证数据回写
```

write只是把数据提交到内核的Page Cache（假设没有启用DIO），要想写到存储设备，必须通过fdatasync或fsync系统调用。下面看一下修正后的代码：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd;
ssize_t wr;
char szBuf[] = "hello world";

fd = open("/sdcard/test.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
wr = write(fd, szBuf, strlen(szBuf));
#if 0
fdatasync(fd);  // 只回写数据
#else
fsync(fd);  // 回写数据和元数据（文件大小、最后修改时间等）
#endif
close(fd);
// 这里掉电是安全的
```

看完系统调用的例子，再来看一下C库的例子：

```
#include <stdio.h>

FILE *fp = null;

fp = fopen("/sdcard/test.txt", "w+");
fputs("hello world", fp);
fclose(fp);
// 在这里掉电，test.txt将会是空文件
```

我们知道C库也有一层文件缓存，所以掉电是因为数据还在这一层缓存中，熟悉C库的同学可能知道这里要加一个fflush，其实并不对，fflush只能保证把缓存回写到下一层，即内核的Page Cache，依然需要调用fsync再刷到物理存储设备。正确写法如下：

```
#include <stdio.h>

FILE *fp = null;
int fd;

fp = fopen("/sdcard/test.txt", "w+");
fputs("hello world", fp);
fflush(fp); // c库缓存写回到内核page cache
fd = fileno(fp);
fsync(fd); // page cache回写到物理设备
fclose(fp);
// 这里掉电数据是安全的
```

最后再来看看Java的例子，下面是一个最常见的Java写文件Demo：

```
public static void writeFile(String filePath, String conent) {
        BufferedWriter out = null;
        try {
            out = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(filePath, true)));
            out.write(conent);
            // 这里掉电会丢数据
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

这个例子仍然是缓存没处理好，BufferedWriter从名字来看就知道是有缓存的，通过手册可以看到调用flush可以写回，那是不是加上这个就够了？答案是否定的，这个函数也只能做到把缓存写到Page Cache，同样还要想办法触发fsync系统调用，正确的写法如下：

```
public static void writeFile(String filePath, String conent) {
        BufferedWriter out = null;
        try {
            FileOutputStream fos = new FileOutputStream(filePath, true);
            out = new BufferedWriter(new OutputStreamWriter(fos));
            out.write(conent);
            out.flush();   // 数据写到page cache
            FileUtils.sync(fos); // 数据写到物理设备，方法1
            //FileDescriptor fd = fos.getFD();
            //fd.sync();  // 数据写到物理设备，方法2
            // 这里掉电是安全的
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
    }
```

总结，如果你的程序需要确保数据立即写回，就需要考虑每一层的缓存，确保每一层都按顺序写回。讲到这里，那有些人就会有疑问，如果我不需要数据马上写回到物理设备，但是我需要知道数据什么时候会写回，这就涉及到我前面说的数据异步写回了。

Linux内核会定时触发，把已经提交到Page Cache的脏数据[1]写回到物理设备，需要特别注意如果你的数据还在上层的缓存中，例如Java或C库的缓存中没有刷下来，那自然不会被内核的异步回写机制写回。大致的异步回写机制步骤如下：

- step 1: 内核按dirty_writeback_centisecs的时间间隔唤醒回写线程
- step 2: 回写线程会遍历Page Cache寻找那些被标记为脏的时间超过dirty_expire_centisecs的页面，并全部回写
- step 3: 回写线程接着会判断脏数据总量是否超过dirty_background_ratio（单位是百分比）或dirty_background_bytes，如果超过则回写所有脏数据
- step 4: 回写线程等待下次唤醒周期

  需要注意，在脏数据超过dirty_ratio和dirty_bytes以后如果继续写数据会自动触发同步回写，即这一次的write会把之前和本次的数据都写回到物理设备再返回。

  最后再列一下Linux内核回写机制的几个可调阈值，它们都位于/proc/sys/vm目录：
- dirty_writeback_centisecs: 控制内核回写线程的唤醒周期，单位是10ms，默认值是500，即5s唤醒一次
- dirty_expire_centisecs：脏数据的过期时间，单位是10ms，默认值是3000，从Page被标记为脏算起，超过这个时间会被认为是过期数据，所以默认情况下，脏数据量没有超过阈值时，数据要等30s以上才会回写，实际上要考虑唤醒周期的影响，数据最长要等35s才会写回到存储设备
- dirty_background_ratio & dirty_background_bytes：二者功能一样，前者是百分比（这里的基数不是总内存大小，而是可用内存大小，包括可回收的内存），后者单位是字节，脏数据总量要超过这个阈值才会全部回写，否则只会写过期数据。这两个是互斥的，写其中一个，另一个会自动被清0
- dirty_ratio & dirty_bytes：二者功能一样，前者是百分比（这里的基数不是总内存大小，而是可用内存大小，包括可回收的内存），后者单位是字节，脏数据量超过这个阈值以后会阻塞write，确保数据同步写回到存储设备

  以上配置参数都可以根据自己的实际需求来调整，例如对于那些视频监控的产品，默认的配置可能会导致数据累积到非常多才会回写，给存储设备的压力就非常大了，这时候可以减小dirty_background_ratio，dirty_writebac-k_centisecs，让数据写入更平滑。但是需要注意，考虑到掉电数据安全，这两个值不能无限减小，太小会导致后台一直在做数据回写，无形中增加了掉电丢数据的风险。最后再强调这些改动是全局的，即对所有的存储设备都生效，所以一定要慎重。

  在Linux下可以通过bashrc中加入echo命令来实现，而Android可以再init.rc脚本里搜一下其他写proc目录节点的位置，在附近加一下你的vm参数的调整，例如：

```
write /proc/sys/vm/dirty_background_ratio 5
```

4 Linux 数据预读

在Linux系统中，默认情况下不管是用户态调用read，还是内核态调用vfs_read，都会触发数据预读，即都会多度一部分数据到Page Cache中，这在顺序读的场景下对性能的提升是非常明显的。而高性能的存储设备可以通过加大预读窗口大小来大幅提升顺序读性能。在Android平台有两个方法来修改预读窗口大小：全局控制和文件单独控制。

全局控制是以存储设备为单位的，建议低速设备用128KB，高速可以用2048KB，例如：

```
write /sys/block/mmcblk0/queue/read_ahead_kb 2048
write /sys/block/dm-0/queue/read_ahead_kb 2048
```

Note: 现在很多设备都开启了verity和encrypt，最终物理存储设备会被映射成dm-x这样的逻辑设备，这些逻辑设备的窗口也要一起改掉才会生效

文件单独控制是以文件为单位的，我们可以通过系统调用给文件一个调整预读窗口的Hint，而不是直接设置预读大小，例子如下：

```
#include <fcntl.h>

posix_fadvise(fd, start, len, POSIX_FADV_SEQUENTIAL);  // 暗示内核，上层会在start开始的len范围内顺序访问，内核会在这个范围内加大预读窗口
posix_fadvise(fd, start, len, POSIX_FADV_RANDOM); // 暗示内核，上层会在start开始的len范围内随机访问，内核会在这个范围内禁止预读
posix_fadvise(fd, start, len, POSIX_FADV_NOREUSE); // 暗示内核，上层在指定范围内只会访问一次
posix_fadvise(fd, start, len, POSIX_FADV_WILLNEED); // 暗示内核，上层短时间内会访问这个范围的数据
posix_fadvise(fd, start, len, POSIX_FADV_DONTNEED); // 暗示内核，上层短时间内不会访问某段数据
```

5 文件掉电保护

每个文件系统都有一套自己的掉电保护机制，但是这个机制只保证文件系统本身的完整性，而无法保证文件的完整性。在本节开始前，有必要对这两者做一下解释：

- 文件系统的完整性：文件系统可正常挂载，所有的文件和目录都可正常访问，可以正常完成所有已实现的文件操作
- 文件的完整性：文件可正常读写，并且功能正常，例如媒体文件要能正常播放，XML文件要能正常解析，压缩文件要能正常解压

  举一个具体的例子会更直观一些，假设有一个应用程序在写一个XML文件来保存一些配置信息，而就在写数据的过程中出现掉电，代码如下：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>

int fd, len;
ssize_t wr;
char *szBuf;

// 假设sdcard分区上已经有了A和B两个文件，并且已经写回到物理设备
fd = open("/sdcard/test.xml", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU | S_IRWXG);
{
    // 在这个区间发生随机掉电
    wr = write(fd, szBuf, 8192);
    fsync(fd);
    close(fd);
}
```

文件系统的完整性可以保证，在重新上电后A和B可以正常访问，并且数据是完整的，而test.xml这个文件可能存在也可能不存在（不启用dirsync的情况下），如果存在的话其数据读写是正常的。但是不能保证数据是完整的，即其大小有可能是[0-8192]的任意值，应用程序很可能无法解析这个XML。

导致这个问题的原因是write和fsync这些函数不是原子的，实际上大部分文件系统的系统调用都无法保证原子性，所以文件的完整性需要应用自己来保证。例如Android实现的AtomicFile，其原理是利用rename函数的原子性来解决问题的，关键函数如下：

```
public FileOutputStream startWrite() throws IOException {
        // Rename the current file so it may be used as a backup during the
next read
        if (mBaseName.exists()) {
            if (!mBackupName.exists()) {
                if (!mBaseName.renameTo(mBackupName)) {
                    Log.w("AtomicFile", "Couldn't rename file " + mBaseName
                            + " to backup file " + mBackupName);
                }
            } else {
                mBaseName.delete();
            }
        }
        FileOutputStream str = null;
        try {
            str = new FileOutputStream(mBaseName);
        } catch (FileNotFoundException e) {
            File parent = mBaseName.getParentFile();
            if (!parent.mkdirs()) {
                throw new IOException("Couldn't create directory " +
mBaseName);
            }
            FileUtils.setPermissions(
                parent.getPath(),
                FileUtils.S_IRWXU|FileUtils.S_IRWXG|FileUtils.S_IXOTH,
                -1, -1);
```

```
            try {
                str = new FileOutputStream(mBaseName);
            } catch (FileNotFoundException e2) {
                throw new IOException("Couldn't create " + mBaseName);
            }
        }
        return str;
    }

 public void finishWrite(FileOutputStream str) {
        if (str != null) {
            FileUtils.sync(str);
            try {
                str.close();
                mBackupName.delete();
            } catch (IOException e) {
                Log.w("AtomicFile", "finishWrite: Got exception:", e);
            }
        }
    }

public FileInputStream openRead() throws FileNotFoundException {
        if (mBackupName.exists()) {
            mBaseName.delete();
            mBackupName.renameTo(mBaseName);
        }
        return new FileInputStream(mBaseName);
    }
```

流程很清晰，写数据前先把文件改名成备份文件，再创建一个新文件来写，写完成以后把备份文件删除，读文件前先判断备份文件是否存在，有存在说明发生掉电，则用备份文件来恢复。有兴趣读完整代码可以参考AtomicFile.java。下面是一段AtomicFile的使用Demo：

```
 public static void write(AtomicFile file, IntervalStats stats) throws
IOException {
        FileOutputStream fos = file.startWrite();
        try {
            write(fos, stats);
            file.finishWrite(fos);
            fos = null;
        } finally {
            // When fos is null (successful write), this will no-op
            file.failWrite(fos);
        }
    }
```

C语言也可以参考这里例子，这里就不详解了。

6 IO高性能编程

(略)

1. Page Cache中被修改过的Page会被标记为脏，其中的数据被叫做脏数据