**内核学习笔记-文件系统篇**

如果操作系统的职责“管理和抽象硬件资源并为上层应用提供”来解释文件系统的话，文件系统就是管理并抽象存储资源（磁盘、Flash等），完成从应用层到物理存储介质的IO（申请、读写、擦除、释放等），并针对不同存储介质特点对IO管理起到性能、损耗、安全等优化效果。

# 操作系统概览

## 概念区分

### 操作系统和内核：

操作系统是一个可用的工作环境，包括内核、命令行shell、用户界面和系统工具等。而内核则是支撑所有软件（包括shell、用户界面、系统工具等）的核心，是真正的管理硬件资源并为上层应用提供支持的管理者。其架构如下：[7]



Linux内核设计遵守简单、模块化的准则，一切硬件资源皆用文件抽象（设备、存储资源等），所有针对文件的操作都当被做进程管理，同时配套内存管理作为多进程交互、执行的高速缓存资源。

### 用户态和内核态：

内核态有受保护的内存地址空间和访问所有硬件的权限；用户态仅执行普通的用户程序，涉及底层硬件资源的程序要陷入内核由内核代其完成。所以处理器在任何时刻都处于以下三个状态之一：[8]

1. 运行于用户空间，执行用户进程
2. 运行于内核空间，处于进程上下文，代表某个特定的进程执行
3. 运行于内核空间，处于中断上下文，与任何进程无关，处理某个特定中断

## 内核源码目录树

|  |  |
| --- | --- |
| 目录 | 描述 |
| arch | 特定体系结构的代码 |
| block | 块设备IO层（文件系统相关） |
| crypto | 加密API |
| Documentation | 内核源码文档 |
| drivers | 设备驱动程序（文件系统相关） |
| firmware | 使用某些设备驱动程序而用到的设备固件 |
| fs | VFS和各种文件系统（文件系统相关） |
| include | 内核头文件 |
| init | 内核引导和初始化 |
| ipc | 进程间通信代码 |
| kernel | 核心子系统，包括调度程序 |
| lib | 通用内核函数 |
| mm | 内存管理子系统和VM（文件系统相关） |
| net | 网络子系统（文件系统相关） |
| samples | 示例，示范代码 |
| scripts | 编译内核所用的脚本 |
| security | Linux安全模块（文件系统相关） |
| sound | 语音子系统 |
| usr | 早期用户空间代码 |
| tools | 在Linux开发中有用的工具 |
| virt | 虚拟化基础结构 |

**内核开发特点：**

1. 不能访问标准C库或头文件，为了防止先有鸡还是先有蛋的悖论
2. 必须使用GNU C，一般包含头文件<linux/xxx.h>，例如/include中的文件
3. 缺乏用户态类似的内存空间保护机制，所以越界、溢出可能直接导致系统奔溃
4. 难以执行浮点运算
5. 每个进程只有一个很小的定长堆栈
6. 内核支持异步中断、抢占，所以编程需要时刻考虑同步和并发
7. 考虑可移植性

# 文件系统基础知识

## 文件系统分层

Linux设计哲学是将将一切都看作文件，普通文件（包括文本和二进制）、目录文件、字符设备文件、块设备文件等。Linux文件系统的层次如下所示[1][7]：



各层的主要功能职责为：

|  |  |  |
| --- | --- | --- |
|  | 层名 | 功能职责 |
| 用户态(User Space) | 应用层(Application] | 基于标准库，构建应用程序 |
| C标准库(Standard Lib C) | 基于系统调用接口，构建Linux平台的C标准库，提供上层应用开发的统一编程接口 |
| 内核(Kernel) | 虚拟文件系统层(Virtual File System [VFS]) | 基于不同文件系统的调用接口，抽象不同文件系统的资源管理方式，以inode构建通用文件模型，管理文件数据和超级块的映射方式，提供对用户态提供统一的系统调用接口[7] |
| 文件系统层(File System [FS]) | 基于通用的块层资源访问接口，组织文件和逻辑块层的映射存储关系，将文件读写、新建、删除操作转换成对应的逻辑块读写、申请、释放操作，提供对应文件系统的读写、新建、删除操作接口 |
| 块层(General Block Device Layer) | 基于不同磁盘的驱动接口，将磁盘物理块抽象为逻辑块，并负责从逻辑块到物理块的映射。管理逻辑块资源，为上层文件提供统一的块操作接口 |
| 驱动层(Device Driver) | 基于实际物理资源，不同厂家提供了不同的磁盘操作接口 |
| 物理(Hardware) | 磁盘层(Physical Disk) | 物理磁盘，提供真实的存储资源 |

以Ext2文件系统写入数据到文件为例讲述从上到下的整个流程：

## 区分块、卷、分区、扇区

### 逻辑分区 VS 物理磁盘：



|  |  |
| --- | --- |
| 名称 | 含义 |
| 物理分区(Physical Part [PP]) | 代表一块物理磁盘 |
| 扇区(Sector) | 物理磁盘写入、读取的最小单位，由若干地址连续的存储单元组成，一般为512B |
| 物理块(Physical Extends [PE]) | 由若干连续的扇区组成，一定是块大小的2^N倍 |
| 逻辑块(Logical Extends [LE]) | 和物理块大小相同，可以通过映射得到对应的物理块。操作系统读写磁盘的基本单元是块 |
| 逻辑分区(Logical Part [LP]) | 由若干逻辑块组成（不要求逻辑或物理地址连续），每个分区可拥有各自的文件系统，对整个磁盘进行划分，更合理地使用硬件资源 |

相比于物理磁盘，逻辑分区允许一块磁盘上运行不同的文件系统，使得硬件资源利用率更高。可以发现，逻辑块和物理块大小相同，逻辑块资源总和应该和物理磁盘大小相同，那么为什么要进行一个从逻辑块到物理块的映射？

如果没有逻辑抽象那么所有的写操作则只支持顺序写，逻辑块主要为了支持物理磁盘的随机读写。上层应用完全可能修改某个已经落盘的文件，假设在原有文件末尾追加了新的内容，这时候需要新的物理块支持新增内容，在没有逻辑映射的情况下，如果和原来物理块地址连续的物理块已经有其他文件的内容，则必须先将后面物理块内的数据先搬移至空闲物理块，然后再将新增内容写入后面的物理块。但如果有一层逻辑映射使得上层文件看起来逻辑地址连续但允许分布在地址不连续的物理块上，那就不需要进行数据搬移了，这类似于物理内存随机读写的技术实现。

### 逻辑卷 VS 逻辑分区：



|  |  |
| --- | --- |
| 名称 | 含义 |
| 物理卷(PV [Physical Volume]) | 代表一块物理磁盘 |
| 扇区(Sector) | 物理磁盘写入、读取的最小单位，由若干地址连续的存储单元组成，一般为512B |
| 物理块(Physical Extends [PE]) | 由若干连续的扇区组成，一定是块大小的2^N倍 |
| 容器池(Volume Group [VG]) | 将多块磁盘的物理块组合，并抽象为一个容器池，里面提供很多逻辑块 |
| 逻辑块(Logical Extends [LE]) | 和物理块大小相同，可以通过映射得到对应的物理块。操作系统读写磁盘的基本单元是块 |
| 逻辑卷(Logical Volume [LV]) | 由若干逻辑块组成（不要求逻辑或物理地址连续），每个卷可拥有各自的文件系统，每个卷的逻辑块对应的物理块可分布于不同的物理磁盘上，实现弹性存储。 |

逻辑卷[2]和逻辑分区很相似，都可以对整个磁盘划分得到多个分区并分别运行不同的文件系统，但逻辑卷的强大之处还在于实现弹性存储。即既可以将多个物理磁盘构成一整个地址连续的逻辑卷，也可以将位于不同物理磁盘上空闲的物理块拼接成一个地址连续的逻辑卷。逻辑卷可以随便命名，而一般的分区都是/dev/sda1之类的。

# Virtual File System：虚拟文件系统层

# File System Layer：文件系统层

# General Block Device Layer：块层

## UBI [Unsorted Block Images]

MTD层之上，FS层之下[3]，针对Flash设备，类似于逻辑卷管理(Logical Volume Management [LVM])，负责从逻辑块到物理块的fast map，提供磨损均衡(wear-leveling)和坏块处理(Bad eraseblocks handling)功能。MTD仅仅是适配Flash设备Driver并为FS层提供统一接口，虽然在用户态（FS级）也做了wear-leveling，但没有对坏块处理，可能导致上层在坏块写入时失败。

### 磨损均衡(wear-leveling)：

UBI将volume分为static和dynamic[4]，static是只读卷，主要存储操作系统、boot程序、只读文件等；dynamic是读写卷，一般存放文件内容数据，所以dynamic卷对应的物理块可能擦除比较频繁，一段时间后需要将static卷擦除计数较低的物理块替换掉dynamic擦除计数较高的物理块。首先将对应物理块上的数据交换，然后修改映射关系，将原擦除计数低的物理块对应的逻辑块指向擦除计数高的物理块，将原擦除计数高的物理块对应的逻辑块指向擦除计数低的物理块[5]。



|  |  |
| --- | --- |
| 名称 | 含义 |
| 逻辑可擦除块(Logical eraseblock [LEB]) | 物理块经过UBI fast map后的逻辑块 |
| 物理可擦除块(Physical eraseblock [PEB]) | 物理可擦除块，Flash设备的基本读写单元 |
| 擦除计数(Erase counter [EC]) | 对每个物理块擦除次数的计数 |
| 物理卷(PV [Physical Volume]) | 代表一块物理磁盘 |

为了记录每个逻辑块对应物理块的擦除计数以及和逻辑卷的对应关系，UBI将擦除计数和volume id记录在对应物理擦除块的Header中[4]，使得每个块的可用空间小于实际块大小。但是擦除计数和volume id是分别记录的，如果一个块还没有被分配给一个逻辑卷，那么它的Header只有擦除计数，分配之后会在Header中追加volume id。例如NAND的Header会占用一个page[4]。



### 坏块处理(Bad eraseblocks handling)：

向LEB1对应的物理块写入new data发现物理块损坏，首先将LEB1对应的物理块PEB2上的old data复制到一块空闲的物理块，例如PEB3，然后再将new data写到PEB3，之后修改映射使得LEB1指向PEB3，最后将PEB1标记为坏块[5]。



块拷贝操作分三步，先更对块的version number自增并更新Header CRC，然后复制块数据到新块，最后擦除旧块数据。如果在复制块数据到新块时发生中断，UBI会选择version number高的块，然后发现该块的data CRC不合法，于是执行回滚操作，纠正version number为低的号码，同时擦除不完整的块，并重新执行块拷贝操作；如果在擦除旧块数据时发生中断，UBI会选择version number高的块，然后发现该块的data CRC合法，于是擦除低version number的块，完成块拷贝。

### 预留卷：

UBI最多可以被分为0~65535个卷，其中0~65279供用户使用，65280~65535保留为内部供逻辑块映射、卷信息使用。内部卷名一般以“ubi-”为前缀[4]。

卷65280的名称是ubi-layout，记录了卷的数量、大小和属性。每当一个卷被创建、删除或resize，它就会被更新[4]。

### 备用块：

当磨损均衡触发需要交换两个物理块的数据时，至少需要一个外部块用作交换的临时数据存放点，而此时如果空闲快为0，则无法完成交换，所以至少需要一个备用块支持磨损平衡产生的块数据交换操作[4]。

此外，UBI规定了最大的坏块数量，如果坏块数量超过这个阈值，那么UBI会禁止所有写操作并通知用户手动处理。在进行一次写操作发现新的坏块，此时需要进行坏块处理，如果没有空闲块，那么进行坏块处理时就需要一个备用块来作为旧数据存放点。所以一般会设置备用块数量大于等于最大坏块数量[4]。

综合预留卷和备用块，实际用户可用物理空间为[4]：

|  |  |
| --- | --- |
| 符号 | 含义 |
|  | 可供用户使用的物理块数量 |
|  | Flash设备上的物理块数量 |
|  | UBI保留的物理块数量 |
|  | UBI规定的最大坏块数量 |
|  | UBI预留卷占用的物理块数量 |
|  | 可供用户使用的字节数 |
|  | 每个物理块给用户可用的字节数 |
|  | 每个物理块实际含有的字节数 |
|  | UBI在每个物理块上Header（擦书计数和volume id）占用的字节数。对于NOR Flash是2倍的Header大小；对于NAND Flash是页大小[4] |

Fast map：

# Driver Layer：驱动层

# Hardware Layer：物理层

## Flash设备

Flash[1]，顾名思义，读取速度很快，但写入比较麻烦。写入操作只能把对应位置从1改为0，而不能把0改为1，擦除则是将存储块内容恢复为1，写入、擦除、读取的基本单位是块。擦除次数有限，一般为104~105次。Flash常用于手机，因为经常震动的原因磁盘不能作为手机存储介质。

# GNU C

## 定位[8]

并不完全符合ANSI C标准，内核开发者使用的C涵盖了ISO C99和GNU C扩展特定，而gcc（多种GNU编译器的集合）既可以编译标准C又可以编译GNU C，所以内核常用gcc编译。

## 内联函数[8][9]

没有函数调用和返回带来的压栈、弹栈开销，直接将函数代码放在调用处展开。适合长度短、调用频率低、没有递归的函数，一般作用域在文件内，所以一般内联函数都要在前边加static关键字，例如static inline void f(){ … }。多处调用内联函数会造成代码体积膨胀。

如果inline前面没有static关键字，为了能在文件外部调用，gcc一定会为其生成一份独立的汇编代码；如果在inline前面有extern关键字，该函数只会被内联进去，gcc不会为其生成独立的汇编代码，不会与全局同名函数冲突，作用域只在本文件内生效。

## 打印

使用printk代替printf打印数据到内核日志缓冲区，再使用syslog查看打印内容。

## 优化条件选择分支

likely和unlikely可给编译器做出手动优化，表示某个分支发生的可能性，用于提高性能。如果某个分支发生的可能性几乎总是为真，则可以使用likely对其优化，例如：if (likely(success)) { … }

# 参考文献

1. 文件系统分层. <https://blog.csdn.net/yuexiaxiaoxi27172319/article/details/45241923>.
2. 逻辑卷管理(LVM). <https://blog.csdn.net/cloudeagle_bupt/article/details/41721057>.
3. UBI简介. <https://www.cnblogs.com/keanuyaoo/p/3347806.html>.
4. UBI - Unsorted Block Images. Thomas Gleixner, Frank Haverkamp, Artem Bityutskiy. International Business Machines Corp, 2006.
5. UBI - Unsorted Block Images. Artem Bityutskiy <artem.bityutskiy@nokia.com>. PPT.
6. UBI Fastmap. Thomas Gleixner. Embedded Linux Conference Europe, 2012.
7. 《深入理解Linux内核架构》. Wolfgang Mauerer, 人民邮电出版社, 2017.
8. 《Linux内核设计与实现》, Robert Love. 机械工业出版社, 2011.
9. C内联函数. <https://www.cnblogs.com/xkfz007/articles/2370640.html>.