**Linux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

## 进程和线程

### 进程状态：

TASK\_RUNNNIG：进程可能正在执行或处于调度队列等待执行。

TASK\_INTERRUPTIBLE：进程处于阻塞状态，并等待信号唤醒。

TASK\_UNINTERRUPTIBLE：进程处于等待，不受信号干扰。

****

### 线程和进程的区别：

在内核中，线程和进程具有相同的数据结构和调度算法，线程仅仅是一个和其他进程共享某些资源的进程。进程和线程创建的系统调用接口一样，只是参数不同。

创建线程的系统调用：clone(CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND, 0)，表示父子进程共享地址空间、文件系统资源、文件描述符和信号处理程序，换言之新建的进程和其父进程就是两个线程。

创建进程的系统调用：clone(SIGCHLD, 0)

# 文件系统基础知识

## 文件系统分层

Linux设计哲学是将一切都看作文件，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（基于某个MTD分区），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]。

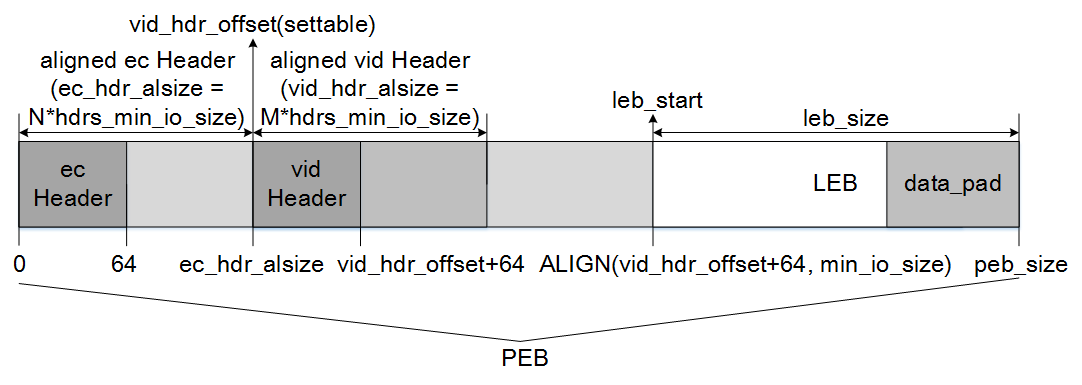


|  |  |
| --- | --- |
| 名称 | 含义 |
| 逻辑可擦除块(Logical eraseblock [LEB]) | 物理块经过UBI fast map后的逻辑块 |
| 物理可擦除块(Physical eraseblock [PEB]) | 物理可擦除块，Flash设备的基本读写单元 |
| 擦除计数(Erase counter [EC]) | 对每个物理块擦除次数的计数 |
| 物理卷(PV [Physical Volume]) | 代表一块物理磁盘 |

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分别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物理块大小。

### UBI PEB：[4] [20]

每个PEB的构成如图所示：



**数据结构：**

### 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。

### 预留块的计算和设置方法：

**每个UBI设备上的预留块数量包括（不论UBI设备基于一个MTD分区还是一个MTD设备）[4][20]：**

|  |  |
| --- | --- |
| 预留块组成 | 源码 |
| 用于磨损均衡的预留块，预留块数量为WL\_RESERVED\_PEBS[1]。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，而此时如果空闲块数量为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 | drivers/mtd/ubi/wl.c-ubi\_wl\_init： |
| 如果定义了fastmap的扫描方式，还需预留2倍的fastmap占用的LEB数量。ubi->fm\_size计算方法见[ubi\_attach\_mtd\_dev](#ubi_ubi_attach_mtd_dev)。 | drivers/mtd/ubi/wl.c-ubi\_wl\_init：    drivers/mtd/ubi/wl.h： |
| 内部卷(目前仅有layout\_volume，未来可能添加日志卷)使用的预留块，UBI最多可以被划分为128+1个卷，可用ID为0~65535，其中0~65279供用户使用，65280~65535保留为内部供逻辑块映射、卷分布信息使用。内部卷名一般以“ubi-”为前缀；卷65280的名称是ubi-layout，记录了用户卷的数量、大小和属性，会预留UBI\_LAYOUT\_VOLUME\_EBS[2]个PEB。layout卷会在attach后自动创建，不管当前UBI设备有没有用户数据。 | drivers/mtd/ubi/vtbl.c-init\_volumes (called by ubi\_attach->ubi\_read\_volume\_table)： |
| 用户卷预留PEB。每当一个卷被创建、删除或resize，存放用户数据的PEB数量即用户卷预留块数量也会被更新，同时还要更新layout\_volume的数据内容。 | drivers/mtd/ubi/vmt.c-ubi\_resize\_volume：调整卷大小可能会增加/减少预留PEB数量    drivers/mtd/ubi/vmt.c-ubi\_remove\_volume：移除卷会减少预留PEB数量    drivers/mtd/ubi/vmt.c-ubi\_create\_volume：创建卷会增加预留PEB数量，预留的PEB实际上就用来存放用户数据。 |
| 坏块处理预留的PEB数量ubi->beb\_rsvd\_level和eba子系统预留的PEB数量EBA\_RESERVED\_PEBS[1]。其中，ubi->beb\_rsvd\_level等于ubi->bad\_peb\_limit(UBI设备允许的最大坏块数量)-ubi->bad\_peb\_count(已经标记为坏块的PEB数量)。而ubi->bad\_peb\_limit实际上就是整个flash设备的PEB数量（注意，不是单个MTD分区）\*max\_peb\_per1024/1024的向上取整。可以通过设置内核参数CONFIG\_MTD\_UBI\_BEB\_LIMIT设定相对值max\_peb\_per1024。ubi->bad\_peb\_count是ubi\_attach扫描结束后为UBI设备得到的已知坏块数量，通过ai->bad\_peb\_count赋值。允许的坏块数量(beb\_rsvd\_level)和为坏块处理预留的PEB数量(beb\_rsvd\_pebs)之间的差值也应当累加到ubi->rsvd\_pebs，负数不算。  注意：坏块处理预留的PEB数量一定在wl、eba、卷预留PEB数量从可用PEB数量扣除后才会扣除。 | drivers/mtd/ubi/eba.c-ubi\_eba\_init：为坏块处理预留的PEB数量    drivers/mtd/ubi/misc.c-ubi\_calculate\_reserved：如何得到ubi->beb\_rsvd\_level    drivers/mtd/ubi/build.c-get\_bad\_peb\_limit：通过相对值max\_peb\_per1024计算UBI设备为坏块处理预留的PEB数量      drivers/mtd/ubi/build.c-ubi\_attach\_mtd\_dev：使用内核参数CONFIG\_MTD\_UBI\_BEB\_LIMIT设置相对值max\_peb\_per1024    drivers/mtd/ubi/attach.c-scan\_peb：attach过程扫描更新已知坏块数量    drivers/mtd/ubi/misc.c- ubi\_update\_reserved：再做一次检查，确保需要为坏块处理预留的PEB数量(ubi->beb\_rsvd\_level)和当前为坏块处理预留的PEB数量(ubi->beb\_rsvd\_pebs)相等 |

**实际用户可用物理空间为：**

|  |  |
| --- | --- |
| 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允许的最大坏块数量 |
|  | UBI预留卷(layout卷)和用户卷占用的物理块数量 |
|  | eba子系统预留的物理块数量 |
|  | 为fastmap预留的物理块数量 |
|  | 可供用户使用的字节数 |
|  | 每个物理块给用户可用的字节数（LEB大小） |
|  | 每个物理块实际含有的字节数 |
|  | UBI在每个物理块上Header（擦除计数和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] |

**ubi\_attach过程各种预留快的块的扣除顺序和影响：**

以下预留块扣除都是建立在ubi->avail\_peb(ubi->peb\_count - ubi->bad\_peb\_count -ubi->corr\_peb\_count)的基础之上，如果坏块和corr PEB直接超过了所有PEB，ubi->avail\_peb就是一个负数，扣除预留块的第一步（用户卷和内部卷预留块）就直接失败。

|  |  |
| --- | --- |
| 预留块扣除先后顺序 | 可用块不够时的影响 |
| 为用户卷和内部卷（包括layout卷）预留的PEB | 返回错误，attach失败 |
| 为磨损均衡、fastmp预留的PEB | 返回错误，attach失败 |
| 为eab预留的PEB | 返回错误，attach失败 |
| 为坏块处理预留的PEB（允许的最大坏块数量-已检测为坏块的数量） | 警告信息，attach成功 |

****

**实验验证（1GB NAND SIM模拟出的Flash，PEB大小为128KB，LEB大小为126KB，页大小为2048B，一共有8192个PEB）：**

1. 用于wear-leveling的预留PEB数量为3，分别是用于WL的预留PEB(WL\_RESERVED\_PEBS[1])和fastmap占用LEB数量(129024/129024=1)的2倍；
2. 用于layout卷预留的PEB数量为2；
3. 为eba子系统预留的PEB数量为1；
4. 为坏块处理预留的PEB数量为8192\* max\_peb\_per1024[20]/1024=160；
5. 总的预留PEB数量=3+2+1+160=166块；
6. 可以发现，UBI设备0一共才240个PEB，却有166个PEB被预留。



**为什么max\_peb\_per1024是针对整个Flash设备，而不是单个MTD分区？**

由于设备厂商原因，UBI基于MTD的WL和坏块处理是基于整个MTD设备的，任何坏掉的PEB可能被映射到不同的MTD分区（UBI设备），所以最坏情况下所有Bad PEB被映射到同一个UBI设备。那么必须要保证所有UBI设备为坏块预留的PEB数量不小于整个Flash的PEB数量\*max\_peb\_per1024/1024向上取整，详见commitID ba4087e956d336488c6df9dfca65d1e70cf480f1。



**（为每个UBI设备）单独设置预留坏块数量的方法：**

1. 【MTD全局相对值法】修改内核配置使得CONFIG\_MTD\_UBI\_BEB\_LIMIT变小，例如为2。但是这样做会统一设置所有UBI设备的坏块处理预留PEB数量，更合理的做法应该是依据UBI大小不同而设置不同的坏块处理预留的PEB数量，测试效果如下：



1. 【单独UBI相对值法-用户态应用程序设置】设置mtd-utils工具的ubiattach -b参数，设置单个UBI设备的max\_peb\_per1024：

$ ubiattach -m 1 -b 2





用户态对/dev/ubi\_ctrl的ioctl会调用到内核中drivers/mtd/ubi/cdev.c-ctrl\_cdev\_ioctl，继而调用ubi\_attach\_mtd\_dev。



mtd-utils工具在878e06ea555ba5dbfb974b3904d1a86a9a0e20f5这个commit之后允许用户态在ubi\_attach过程通过ioctl传参max\_beb\_per1024来设置单独UBI设备的坏块处理预留PEB数量。



内核在edac493dfb48fe46d43fe6afabb8cfb2d1d4c048这个commit之后允许模块和用户态传参设置单个UBI设备的坏块处理预留PEB数量。



1. 【单独UBI相对值法-用户态加载模块设置】以模块的方式加载MTD\_UBI，并设置max\_peb\_per1024相关参数。设置.config文件以模块方式加载UBI\_MTD，使用模块加载ubi\_attach命令传入参数，对应内核源码的参数解析部分drivers/mtd/ubi/build.c-ubi\_mtd\_param\_parse。





1. 【单独UBI绝对值法-用户态应用程序设置-自实现】

**既然所有坏掉的PEB可能被映射到同一个UBI设备，当每个UBI设备设置的坏块处理预留PEB数量不同，如果某个UBI设备映射后的坏块数量超过预留坏块数量该如何处理？**

UBI设备会给出警告，有必要的话会使用可用的PEB，如果没有可用的PEB，UBI设备会被设置为只读模式（详见commit\_id：37f758a036da56c7cff81b68d1d872752079eb6c）。

|  |  |
| --- | --- |
| 过程 | 源码 |
| 坏块增加，调用预留块数量检查方法。 | drivers/mtd/ubi/wl.c-\_\_erase\_worker： |
| 如果ubi不允许坏块存在，则直接将ubi设备设置为只读模式；如果坏块处理预留PEB数量为0：若可用PEB数量为0，则直接设置ubi设备为只读模式，否则将可用PEB数量减1并设置使用可用PEB作为坏块的标识；如果坏块处理预留PEB数量不为0：若设置了使用可用PEB作为坏块的标识，则需要先将可用PEB数量加1恢复，并对坏块处理预留PEB数量减1，否则直接对坏块处理预留PEB数量减1。处于并发考虑，所以要检查两次预留PEB数量够不够，第一次中可能使用可用块代替坏块，第二次如果预留块数量够了还可能要恢复可用块数量。 | drivers/mtd/ubi/wl.c-\_\_erase\_worker： |
| 为坏块处理的预留PEB不够用时给出警告。 | drivers/mtd/ubi/misc.c-ubi\_calculate\_reserved（被\_\_erase\_worker调用）： |

**预留块可以转换成可用块吗？**

当创建新卷或扩充卷大小需要更多地PEB支持时，如果ubi->avail\_peb不够，则返回错误，ubi->rsvd\_pebs中的任何类型预留块（坏块处理、wl、eba等）都不提供支持。

drivers/mtd/ubi/vmt.c-ubi\_create\_volume：



**坏块处理的预留PEB数量发生变化时对attach过程是否有影响？**

1. max\_beb\_per1024变大，预留块数量增加，直接从ubi->avail\_peb中减去相应的增量。如果ubi->available\_peb小于ubi->beb\_rsvd\_level，则修改ubi->beb\_rsvd\_pebs为ubi->available\_peb，并将全部可用PEB作为坏块处理的预留块，UBI attach过程不失败，仅给出warning。坏块处理预留的PEB数量一定在wl、eba、卷预留PEB数量从可用PEB数量扣除后才会扣除。

drivers/mtd/ubi/eba.c-ubi\_eba\_init：



1. max\_beb\_per1024变小，预留块数量减少，此时坏块数量超过了坏块处理预留的PEB数量。ubi\_attach调用返回成功，但给出警告，同时坏块处理的预留PEB数量设置为0。

drivers/mtd/ubi/misc.c-ubi\_calculate\_reserved（被ubi\_eba\_init调用）：



**实验验证：**1GB （8192个PEB）NAND Flash上有一个拥有240个PEB（其中包括15个坏块）的MTD分区，创建了一个总PEB数量为240的UBI设备，设置max\_peb\_per1024=1即允许最大坏块数量为8，初始坏块处理预留的PEB数量为8：



**预留块可以转换成可用块吗？**

当创建新卷或扩充卷大小需要更多地PEB支持时，如果ubi->avail\_peb不够，则返回错误，ubi->rsvd\_pebs中的任何类型预留块（坏块处理、wl、eba等）都不提供支持。

drivers/mtd/ubi/eba.c-ubi\_eba\_init：



**其他预留PEB数量发生变化时对attach过程的影响？**

1. 为eba子系统预留的PEB数量大于可用PEB数量，attach返回失败

drivers/mtd/ubi/eba.c-ubi\_eba\_init：



1. 为wl和fastmap子系统预留的PEB数量大于可用PEB数量，attach返回失败

drivers/mtd/ubi/wl.c-ubi\_wl\_init：



### UBI attach： [19][20]

基于给定的MTD分区，创建并初始化UBI设备，对应函数drivers/mtd/ubi/attach.c:ubi\_attach。完善struct ubi\_device \*ubi中的数据项，struct ubi\_device的实例用于描述一个UBI设备，（部分）主要数据成员如下所示：

|  |
| --- |
| struct ubi\_device {  int ubi\_num; // UBI设备号  char ubi\_name[sizeof(UBI\_NAME\_STR)+5]; // UBI设备名称  int vol\_count; // UBI设备中的卷数量  struct ubi\_volume \*volumes[UBI\_MAX\_VOLUMES+UBI\_INT\_VOL\_COUNT]; // 数组-UBI设备包含的所有卷  int image\_seq; // 同一UBI设备里所有PEB ec Header中的image\_seq相同，除非为0  int rsvd\_pebs; // 保留的PEB数量，UBI保留，不提供给上层应用  int avail\_pebs; // 空闲可用的PEB数量  int beb\_rsvd\_pebs; // 为坏块处理保留的PEB数量  int bad\_peb\_limit; // 允许的最大损坏块数量  int vtbl\_size; // 卷表大小  struct ubi\_vtbl\_record \*vtbl; // 卷表(in-RAM)，扫描结束后调用函数ubi\_read\_volume\_table完善  int max\_ec; // 最大的erase counter（擦除计数）  /\* Note, mean\_ec is not updated run-time - should be fixed \*/  int mean\_ec; // 平均erase counter  /\* Fastmap stuff \*/  int fm\_disabled; // 是否禁用fastmap扫描方法  struct ubi\_fastmap\_layout \*fm; // 描述fastmap的布局信息(in-RAM)，fastmp最终会存放在几个PEB中，fm描述了这几个PEB的ec和pnum。当fastmap被更新后，fastmap需要回写至flash，描述fastmap在flash布局信息的fm就会用到。fastmap的回写是回写整个fastmap，不是只回写更新的部分。如果是以scan\_all扫描了ubi，ubi->fm并不存在；如果fm\_autoconvert为1，fastmap的raw data会在ubi\_write\_fastmap中填充并写入new\_fm指定的flash区域。fastmap会在创建、移除、resize卷、ubi\_detach等过程中更新并回写flash。  struct ubi\_fm\_pool fm\_pool; // fastmap中的pool部分(in-RAM)  struct ubi\_fm\_pool fm\_wl\_pool;// fastmap中用作WL的pool(in-RAM)  void \*fm\_buf; // 存放整个fastmap的raw data  size\_t fm\_size; // fastmap的大小(Byte)  int fast\_attach; // 非0表示UBI attach是通过fastmap方式扫描的  /\* Wear-leveling sub-system's stuff \*/  struct rb\_root used; // 红黑树-使用的PEB，包含没有出现在ai四个list（erase、corr、free、alien）中的非擦洗PEB  struct rb\_root erroneous; // 红黑树-坏掉的PEB  struct rb\_root free; // 红黑树-正常的PEB，ai->free和ai->erase块会汇总在ubi->free中  int free\_count; // 空闲PEB数量  struct rb\_root scrub; // 红黑树-需要擦洗的PEB，包含没有出现在ai四个list（erase、corr、free、alien）中的擦洗PEB  struct ubi\_wl\_entry \*\*lookuptbl; // 用于快速查询任意PEB的ubi\_wl\_entry的表，包括fastmap\_layout和ai->fastmap（fastmap超级块1可能开始的PEB集合）交集的PEB信息集合  /\* I/O sub-system's stuff \*/  long long flash\_size; // 支持UBI设备的MTD分区大小(Byte)  int peb\_count; // UBI设备上的PEB数量  int peb\_size; // PEB大小(Byte)  int bad\_peb\_count; // 坏块数量  int good\_peb\_count; // 既不是坏块也没有损毁的PEB数量  int corr\_peb\_count; // 损毁块数量，MTD保留不提供给UBI  int min\_io\_size; // 底层MTD分区支持的最小IO字节  int hdrs\_min\_io\_size; // 用作计算ec和vid Header对齐后大小的对齐块字节  int ro\_mode; // MTD分区是否read-only  int leb\_size; // LEB大小(Byte)，每个PEB用来存放用户数据的大小  int leb\_start; // LEB在PEB中的开始偏移位置，即用户数据开始位置  int vid\_hdr\_offset; // vid(volume id) Header在PEB中的偏移  unsigned int bad\_allowed:1; // MTD设备是否允许坏块存在  unsigned int nor\_flash:1; // 底层存储介质是否为NOR Flash  struct mtd\_info \*mtd; // MTD设备描述符  }; |

**attach主要流程：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 对MTD分区以scan\_fast或scan\_all方式进行扫描，完善扫描过程的数据结构struct ubi\_attach\_info \*ai。如果内核参数配置了CONFIG\_MTD\_UBI\_FASTMAP、MTD分区上的PEB数量大于UBI\_FM\_MAX\_START(64)并且force\_scan为0则执行[scan\_fast](#ubi_scan_fast)方式的扫描，否则执行[scan\_all](#ubi_scan_all)方式的扫描。同时，scan\_fast扫描失败还要再执行[scan\_all](#ubi_scan_all)方式的扫描。 | struct ubi\_attach\_info {  struct rb\_root volumes; // 红黑树-扫描得到的所有卷  struct list\_head corr; // 链表-损坏块  struct list\_head free; // 链表-空闲块  struct list\_head erase; // 链表-待擦洗块  struct list\_head alien; // 链表-异常块，MTD不会给UBI使用  struct list\_head fastmap; // 链表-存放所有可能是fastmap 超级块1起始PEB的PEB集合  int empty\_peb\_count; // 仅包含0xFF数据的PEB数量  int bad\_peb\_count; // 检测为坏块的PEB数量  int maybe\_bad\_peb\_count; // 还没有被标记为坏块但检测为坏块的数量  int vols\_found; // 扫描找到的卷数量  int highest\_vol\_id; // 最大卷ID  int force\_full\_scan; // 是否强制执行scan\_all扫描方式  int min\_ec; // 最小ec  int max\_ec; // 最大ec  unsigned long long max\_sqnum; // 最大的PEB seq\_num  int mean\_ec; // 平均ec  uint64\_t ec\_sum; // 所有扫描块的ec总和  int ec\_count; // 扫描了多少个PEB  };  drivers/mtd/ubi/attach.c-ubi\_attach：  C:\Users\c00500058\Desktop\20190522111833.png |
| 将扫描填充完成的ai数据项对应赋值给ubi，完善ubi设备描述信息，除了一些基本数据，还包括init\_volumes、wl、eba等子系统的初始化信息。 | drivers/mtd/ubi/attach.c-ubi\_attach： |

**scan\_all扫描方法：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 一共有ubi->peb\_count个PEB，从0号PEB开始对每个PEB执行[scan\_peb](#ubi_scan_peb)扫描 | drivers/mtd/ubi/attach.c-scan\_all： |
| 计算平均ec | drivers/mtd/ubi/attach.c-scan\_all： |
| 调用late\_analysis进行一个UBI全局的检查，例如corrupt PEB数量太多则attach失败。 | drivers/mtd/ubi/attach.c-late\_analysis（被scan\_all调用，UBI全局检查）： |
| 对于所有PEB，除了挂到ai->alien中的PEB，如果PEB的ec为UBI\_UNKNOWN，则将其ec赋值为ai->mean\_ec，在struct ubi\_attach\_info中提到过ai->alien里的PEB不会被UBI使用。 | drivers/mtd/ubi/attach.c-scan\_all： |
| 调用self\_check\_ai检查ai数据内容是否正确，包括每个卷的vol\_id、类型，以及卷中每个PEB的ec、pnum等信息 | drivers/mtd/ubi/attach.c-scan\_all（检查ai数据项）：    drivers/mtd/ubi/attach.c-self\_check\_ai（被scan\_all调用，具体检查内容）： |

**scan\_peb：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 跳过已标记为坏块的PEB | drivers/mtd/ubi/attach.c-scan\_peb： |
| 调用ubi\_io\_read\_ec\_hdr读取ec Header (struct ubi\_ec\_hdr\*)，并在其内检查magic、version和hdr\_crc。根据ubi\_io\_read\_ec\_hdr读取返回结果决定是否将PEB移入ai->erase。  注意：attach过程中PEB的描述数据结构是struct ubi\_ainf\_peb \*aeb；attach过程的volume的描述数据结构是struct ubi\_ainf\_volume \*av。 | drivers/mtd/ubi/io.c-ubi\_io\_read\_ec\_hdr：（被scan\_peb调用，读取ec Header并做检查）    drivers/mtd/ubi/attach.c-scan\_peb（根据ec Header读取返回结果将PEB移入对应链表）：    struct ubi\_ainf\_peb { // attach过程中PEB的数据结构  int ec; // PEB的擦除次数  int pnum; // 物理块号  int vol\_id; // PEB所属卷的ID  int lnum; // 对应的LEB号  unsigned int scrub:1; // 该PEB是否需要被擦除  union {  struct rb\_node rb; // 对应卷中的PEB红黑树  struct list\_head list;  } u;  };  struct ubi\_ainf\_volume { // attach过程中volume的数据结构  int vol\_id; // 卷ID  int leb\_count; // 卷中包含的LEB数量  int vol\_type; // 卷类型  int compat; // 该卷的兼容性标志  struct rb\_node rb; // scan为所有卷构建的红黑树  struct rb\_root root; // 红黑树（根）-该卷包含的PEB  }; |
| 调用ubi\_io\_read\_vid\_hdr读取vid Header (struct ubi\_vid\_hdr \*)，并在其内检查magic、hdr\_crc、vol\_id、data\_crc和vol\_type等。根据ubi\_io\_read\_ec\_hdr读取返回结果决定是否将PEB移入ai->erase、ai->free、ai->corr和ai->alien。 | drivers/mtd/ubi/io.c-ubi\_io\_read\_vid\_hdr（被scan\_peb调用，读取vid Header并做检查）：    drivers/mtd/ubi/attach.c-scan\_peb（根据vid Header读取返回结果将PEB移入对应链表）： |
| 对于没有放入ai->free、ai->corr等链表中的PEB：判断PEB对应volume id是否可能是fastmap的超级块1所在卷，如果是，则调用add\_fastmap将其移入ai->fastmap；否则调用ubi\_add\_to\_av先将PEB对应的volume移入ai-> volumes中的红黑树（如果该volume还未被移入），然后该volume的PEB红黑树会添加该PEB如果该PEB不存在于红黑树，但如果已经存在并且是newer，需要更新对应的PEB。 | drivers/mtd/ubi/attach.c-scan\_peb（将PEB移入fastmap还是volume）：    drivers/mtd/ubi/attach.c-ubi\_add\_to\_av（被scan\_peb调用，添加volume和PEB）：    drivers/mtd/ubi/attach.c-ubi\_add\_to\_av（被scan\_peb调用，RB中已存在PEB，更新PEB）： |
| 更新ai中ec相关数据项 | drivers/mtd/ubi/attach.c-scan\_peb： |

**scan\_fast扫描方法：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 调用[scan\_peb](#ubi_scan_peb)扫描块号在0-UBI\_FM\_MAX\_START(64)的PEB，这次扫描结束后scan\_ai->fastmap数据会被填充。 | drivers/mtd/ubi/attach.c-scan\_fast： |
| 调用[ubi\_scan\_fastmap](#ubi_ubi_scan_fastmap)，执行fastmap扫描方法。 | drivers/mtd/ubi/attach.c-scan\_fast： |

**ubi\_scan\_fastmap：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 调用find\_fm\_anchor在scan\_ai->fastmap找到最大sqnum的PEB块号(fm\_anchor)，该PEB就是fastmap 超级块1的起始块。并拷贝scan\_ai->fastmap到ai->fastmap。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap： |
| 根据fm\_anchor从mtd分区读取struct ubi\_fm\_sb大小的数据，这些数据构成了fastmap的超级块1(REF)，REF描述了整个fastmap用到的所有PEB布局；之后对超级块中的数据项做检查，超级块1 的magic和version要正确，fastmap使用的PEB数量不得超过UBI\_FM\_MAX\_BLOCKS(32)；然后计算fastmap占用的空间为fm\_size。ubi->fm\_size早已在[ubi\_attach\_mtd\_dev](#ubi_ubi_attach_mtd_dev)中ubi\_calc\_fm\_size计算出来，这里还会将其和fm\_size比较一次。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（从MTD以fm\_anchor物理块号开始，偏移sizeof(struct ubi\_fm\_sb)字节读取fastmap超级块1）：    drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（检查超级块1的数据项）：    struct ubi\_fm\_sb { // fastmap的超级块数据结构  \_\_be32 magic; // UBI\_FM\_SB\_MAGIC  \_\_be32 used\_blocks; // fastmap使用的PEB数量  \_\_be32 block\_loc[UBI\_FM\_MAX\_BLOCKS]; // fastmap用到的所有PEB的位置  \_\_be64 sqnum; // fastmap中PEB最大的squence num  } \_\_packed; |
| 对fastmap用到的每个PEB：读取ec Header，检查其image\_seq和ubi->image\_seq是否一致；读取vid Header，获取vid\_hdr->sqnum，并记录最大的sqnum；追加从磁盘上读取的PEB到ubi->fm\_buf (fastmap)中。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（获取fastmap用到的每个PEB num）：    drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（读取ec Header，检查image\_seq一致性）：    drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（读取vid Header，记录最大的sqnum）：    构建fastmap，将PEB放入fm\_buf末尾： |
| 将ubi->fm\_buf (fastmap)的开始struct ubi\_fm\_sb个字节赋值给fastmap超级块2(MAP)，验证CRC正确后令fmsb2->sqnum=sqnum，fm->used\_blocks = used\_blocks，fm是用于描述fastmap布局信息的临时变量。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap： |
| 调用ubi\_attach\_fastmap完善ai信息。首先基于ubi->fm\_buf (fastmap)获取fastmap超级块2，将sqnum赋值给ai->max\_sqnum，并设置偏移量fm\_pos为sizeof(struct ubi\_fm\_sb)。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，从fastmap读取超级块2，并设置偏移量fm\_pos到相应位置）： |
| 接着在fastmap中偏移量为fm\_pos的位置读取fmhdr，并继续累加偏移量fm\_pos到相应fastmap位置。fmhdr描述了fastmap存在的四类PEB集合和卷Header集合，在之后fastmap的偏移中会出现它们。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，从fastmap读取fmhdr，并累加偏移量fm\_pos到相应位置）：    struct ubi\_fm\_hdr { // fastmap数据区描述Header  \_\_be32 magic; // UBI\_FM\_HDR\_MAGIC  \_\_be32 free\_peb\_count;  \_\_be32 used\_peb\_count;  \_\_be32 scrub\_peb\_count;  \_\_be32 bad\_peb\_count;  \_\_be32 erase\_peb\_count;  \_\_be32 vol\_count; // fastmap包含的卷数量  } \_\_packed; |
| 之后，在fastmap中偏移两个sizeof(struct ubi\_fm\_scan\_pool)分别得到fmpl和fmpl\_wl，并使用fmpl和fmpl\_wl对上一步描述fastmap布局信息的局部变量fm的部分数据项赋值。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，从fastmap读取fmpl和fmpl\_wl，并累加偏移量fm\_pos到相应位置）： |
| 继续以sizeof(struct ubi\_fm\_ec)为单位在fastmap中偏移，直到fmhdr的free\_peb\_count、used\_peb\_count、scrub\_peb\_count、erase\_peb\_count的总数和（不会超过ubi->peb\_count），并将每个fmec对应块号的PEB放入ai->free、ai->erase或used (struct list\_head\*)，此过程需要更新ai->ec\_sum、ai->ec\_count、ai->min\_ec、ai->max\_ec。计算ai->mean\_ec。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，在fastmap中以sizeof(struct ubi\_fm\_ec)为偏移单位每次读取一个struct ubi\_fm\_ec\* fmec）：    struct ubi\_fm\_ec { // fastmap中的PEB块描述信息  \_\_be32 pnum; // PEB块号  \_\_be32 ec; // 擦除次数  } \_\_packed; |
| 进入循环（循环执行fmhdr->vol\_count次，不超过UBI\_MAX\_VOLUMES[128]），每次循环执行：  在fastmap中偏移sizeof(struct ubi\_fm\_volhdr)得到一个卷fmvhdr，将卷移入ai-> volumes中的红黑树（如果该卷还未被移入）；在fastmap中偏移sizeof(struct ubi\_fm\_eba)得到PEB和LEB的映射表fm\_eba，对于映射表中的每一项，读取该项的PEB块号，把对应块号的PEB移入（已经移入ai->volumes中对应fmvhdr的）卷的PEB红黑树中，映射表中的PEB都应该在used链表中。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，在fastmap中基于fmhdr对fastmap数据区域的描述信息扫描数据区域PEB和volume）：    struct ubi\_fm\_volhdr { // fastmap中的卷描述信息  \_\_be32 magic; // UBI\_FM\_VHDR\_MAGIC  \_\_be32 vol\_id; // 卷ID  \_\_u8 vol\_type; // 卷类型  \_\_be32 used\_ebs; // 卷中包含的LEB数量  } \_\_packed;  struct ubi\_fm\_eba { // fastmap对eba子系统描述信息  \_\_be32 magic; // UBI\_FM\_EBA\_MAGIC  \_\_be32 reserved\_pebs; // 映射表有多少项  \_\_be32 pnum[0]; // PEB块号，注意初始是0，当reserved\_pebs大于0时，每个PEB块号会在fastmap中偏移sizeof(\_\_be32)个字节  } \_\_packed; |
| 调用scan\_pool扫描fmpl和fmpl\_wl中PEB num对应的PEB（pool中的PEB数量不会超过UBI\_FM\_MAX\_POOL\_SIZE[256]），对于每个PEB检查ec Header和vid Header，如果err是UBI\_IO\_FF或UBI\_IO\_FF\_BITFLIPS则从ai->volumes对应volume的红黑树移除PEB，并将PEB放入free(struct list\_head \*)；如果err是0或UBI\_IO\_BITFLIPS调用process\_pool\_aeb（其过程和scan\_peb第四步“ubi\_add\_to\_av-RB中已存在PEB，更新PEB”类似）。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，扫描fmpl和fmpl\_wl）：    drivers/mtd/ubi/fastmap.c-scan\_pool（被ubi\_attach\_fastmap调用，扫描pool中的PEB）：    struct ubi\_fm\_scan\_pool { // fastmap中的pool  \_\_be32 magic; // UBI\_FM\_POOL\_MAGIC  \_\_be16 size; // pool中PEB数量  \_\_be16 max\_size; // pool的容量，不超过256(UBI\_FM\_MAX\_POOL\_SIZE)  \_\_be32 pebs[UBI\_FM\_MAX\_POOL\_SIZE]; // PEB块号  } \_\_packed; |
| 将free中的PEB添加到ai->free，将used中的PEB添加到ai->erase。 | drivers/mtd/ubi/fastmap.c-ubi\_attach\_fastmap（被ubi\_scan\_fastmap调用，完善ai->free和ai->erase链表）： |
| 遍历fastmap超级块2中的block\_loc和fmsb2->block\_ec完善fm->e。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（完善描述fastmap布局信息的临时变量fm）： |
| 完善ubi中和fastmap相关的描述信息。 | drivers/mtd/ubi/fastmap.c-ubi\_scan\_fastmap（完善ubi中fastmap相关信息）： |

**ubi\_scan\_fastmap扫描过程和fastmap布局图：**



以下几点需要注意：

1. fastmap根据REF寻找MAP(fastmap)，再根据MAP扫描POOL，整个MAP(fastmp)不会超过UBI\_FM\_MAX\_BLOCKS\*sizeof(PEB)个字节。
2. fmhdr(struct ubi\_fm\_hdr\*)中有free、used、scrub、erase四个（struct ubi\_fm\_ec \*）fmec集合的个数。根据每个集合的数量N，在fastmap中fmpl\_wl位置之后分别做N次sizeof(struct ubi\_fm\_ec)的长度偏移就可以得到四个集合中PEB的ec和pnum信息，这四个集合的总数应该等于ubi->peb\_count；fmhdr还描述了volume集合个数，在fastmap中erase位置之后做fmhdr->vol\_count(<=UBI\_MAX\_VOLUMES[128])次偏移，每次偏移内容包括fmvhdr(struct ubi\_fm\_volhdr\*)、fm\_eba(struct ubi\_fm\_eba\*)和fm\_eba-> reserved\_pebs个pnum(\_\_be32)，每个pnum映射的PEB都会被移入fmvhdr->vol\_id对应的volume红黑树。
3. fmpl(struct ubi\_fm\_scan\_pool\*)和fmpl\_wl(struct ubi\_fm\_scan\_pool\*)存放块号的pebs数组中的每个pnum都会被映射到对应的PEB，并且扫描每个PEB的ec Header和vid Header。pool中的pnum数量不超过UBI\_FM\_MAX\_POOL\_SIZE[256]。

**ubi\_attach过程结束后的UBI设备块类型构成：**

|  |  |
| --- | --- |
| 过程 | 源码 |
| 可用PEB数量在attach中scan过程结束后得到，其中ubi->corr\_peb\_count和ubi->bad\_peb\_count通过scan过程得到的ai数据结构赋值。ubi\_attach执行完scan之后的ubi\_read\_volume\_table、ubi\_wl\_init和ubi\_eba\_init还会将rsvd\_peb（卷、WL、eba和坏块处理使用的预留块）从ubi->avail\_pebs中扣除： | drivers/mtd/ubi/vtbl.c-ubi\_read\_volume\_table（被ubi\_attach调用）：    drivers/mtd/ubi/attach.c-ubi\_attach：    drivers/mtd/ubi/vtbl.c-init\_volumes(被ubi\_attach-> ubi\_read\_volume\_table调用)：    drivers/mtd/ubi/wl.c-ubi\_wl\_init：    drivers/mtd/ubi/eba.c-ubi\_eba\_init： |



**ubi\_attach\_mtd\_dev：**

drivers/mtd/ubi/build.c:ubi\_attach\_mtd\_dev调用了[ubi\_attach](#ubi_UBI_attach)，成功则返回UBI设备号ubi\_num。在ubi\_attach\_mtd\_dev中对mtd做了一些检查，对ubi数据结构做了一些初始化操作：

|  |  |
| --- | --- |
| 过程 | 源码 |
| 如果MTD设备是基于UBI volume模拟的，则不支持递归attach；如果底层flash是MLC NAND，也不支持。 | drivers/mtd/ubi/build.c-ubi\_attach\_mtd\_dev（检查MTD设备）： |
| 初始化ubi部分信息，调用[ubi\_attach](#ubi_UBI_attach)。可以发现，ubi->fm\_size(fastmap大小)可以通过ubi\_calc\_fm\_size计算出来。 roundup用于块字节对齐。 | drivers/mtd/ubi/build.c-ubi\_attach\_mtd\_dev（初始化ubi部分信息）：    drivers/mtd/ubi/fastmap.c-ubi\_calc\_fm\_size（被ubi\_attach\_mtd\_dev 调用，计算fastmap大小，可以发现fastmap包含的块和ubi\_attach\_fastmap中fastmap扫描的块大小一致）： |

### 使用磁盘模拟MTD设备，并基于MTD设备创建UBI设备：

**方法一：使用nandsim基于内存模拟NAND Flash设备**

1. 修改内核编译配置.config文件如下：

CONFIG\_MTD\_UBI=y

CONFIG\_MTD\_NAND=y

CONFIG\_UBIFS\_FS=y

CONFIG\_MTD\_NAND\_NANDSIM=m

1. 编译内核和模块，可使用共享文件夹软连接本地/lib/modules到宿主机编译好的/lib/modules。启动qemu执行命令创建ubi设备（默认已安装mtd-util）：[17][18]





$ modprobe nandsim first\_id\_byte=0x20 second\_id\_byte=0xa2 third\_id\_byte=0x00 fourth\_id\_byte=0x15 # 创建一个页大小为2KB，总大小为64MB的MTD设备

$ modprobe nandsim first\_id\_byte=0xec second\_id\_byte=0xd3 third\_id\_byte=0x51 fourth\_id\_byte=0x95 parts=240,800,1600,560 # 创建一个页大小为2KB，总大小为1GB的MTD设备，并将其分为5个分区，分别有240、800、1600、560、总块数[8192]-3200个PEB

$ ubiformat /dev/mtd1

$ ubiattach /dev/ubi\_ctrl -m 1

1. 查看UBI设备



1. 创建卷，在逻辑卷初始化文件系统并挂载

$ ubimkvol /dev/ubi0 -s 100MiB -N vol1 # 卷名为vol1，大小为100MB

$ mkfs.ubifs /dev/ubi0\_0

$ mount /dev/ubi0\_0 ~/tmp

$ umount ~/tmp # 卸载卷设备

**方法二：使用mtdram基于内存模拟NOR Flash设备**

1. 修改内核编译配置.config文件如下：

CONFIG\_MTD\_MTDRAM=y

CONFIG\_MTDRAM\_TOTAL\_SIZE=1048576 #指定mtd设备的大小为1048576KB CONFIG\_MTDRAM\_ERASE\_SIZE=4 # 指定mtd设备的PEB的大小为4KB

1. 编译内核，并启动qemu执行如下命令创建ubi设备（默认已安装mtd-util）：[17]



$ ubiformat /dev/mtd0

$ ubiattach /dev/ubi\_ctrl -m 0

1. 查看UBI设备



# 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。NAND Flash可能存在坏块，在UBI设备中一般有保留物理块用作坏块处理或磨损平衡；NOR Flash是基于RAM的，一般没有保留块。NOR作为运行介质，NAND作为大容量存储介质。

# 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[8]

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)) { … }

## 类型推断

示例：

|  |
| --- |
| struct S{  int u;  };  struct S \*s;  typeof(\*s) ss;  ss.u = 12;  printf("%d\n", ss.u);  **输出：** |

# 常用数据结构[8][20]

## 双向环形链表

### 结构定义：





### 使用：

将list\_head作为某个结构体对象的成员变量，该成员变量的地址就是结构体对象在链表中的标识，以ubi attach过程中的free链表为例：



### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 根据链表标识获取对象地址。ptr是链表地址，type是结构体对象类型，member是结构体对象中的链表标识名。使用举例container\_of(&ai->free, typeof(\*aeb), u.list)，根据ai->free链表中第一个元素找到对应链表标识的结构体对象struct ubi\_ainf\_peb \*aeb。 | scripts/kconfig/list.h：其中offsetof用于计算链表标识在结构体对象中的地址偏移量 |
| 初始化链表。使用举例，INIT\_LIST\_HEAD(&ai->free)初始化ai->free链表。链表一开始将前向后向指针指向自己，链表自身也作为其中的一个元素。 | tools/include/linux/list.h： |
| 向链表添加元素。使用举例，list\_add(&aeb->u.list, &ai->free)，添加aeb到ai->free链表的头部，其中list\_add中的head参数就是链表自身，初始化过程中已指明，头部位置就是head的下一个元素；list\_add\_tail(&aeb->u.list, &ai->free) 添加aeb到ai->free链表的尾部，尾部位置就是head的上一个元素。 | tools/include/linux/list.h：    tools/include/linux/list.h：    tools/include/linux/list.h： |
| 从链表删除节点。使用举例list\_del(&aeb->u.list)，删除aeb。LIST\_POISON1=LIST\_POISON2=NULL。 | tools/usb/usbip/libsrc/list.h：    tools/usb/usbip/libsrc/list.h： |
| 遍历链表。使用举例，list\_for\_each\_entry(aeb, &ai->free, u.list)，遍历ai->free链表中的每个元素。 | scripts/kconfig/list.h： |
| 合并链表，将链表list合并到链表head头部，合并和并将list初始化，合并后的所有元素在head中。使用距离，list\_splice\_init(&free, &ai->free)，把链表free合并到ai->free中，并对free初始化。 | include/linux/list.h： |

### 特点：

1. 灵活，插入元素、删除元素、合并链表时间复杂度为O(1)
2. 无序，只能顺序遍历，查询元素时间复杂度为O(n)

## 红黑树

### 结构定义：



### 使用：

将rb\_node作为某个结构体对象的成员变量，该成员变量的地址就是结构体对象在红黑树中的标识；rb\_root是红黑树标识，里面的rb\_node成员也作为红黑树的一个节点，初始化为NULL。以ubi attach过程中构建的volume红黑树为例：





### 操作：

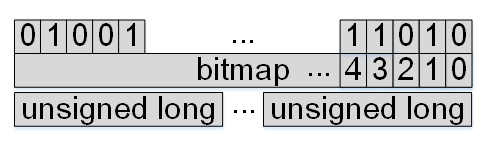
|  |  |
| --- | --- |
| 操作 | 源码 |
| 根据红黑树标识获取对象地址。ptr是红黑树地址，type是结构体对象类型，member是结构体对象中的红黑树标识名。使用举例rb\_entry(ai->volumes.rb\_node, struct ubi\_ainf\_volume, rb)，根据ai->volumes红黑树中根节点rb\_node找到对应红黑树标识的结构体对象struct ubi\_ainf\_volume \* av。 | tools/include/linux/rbtree.h：其中offsetof用于计算链表标识在结构体对象中的地址偏移量 |
| 插入节点。首先在红黑树中遍历，找到适合的节点插入位置；然后调用rb\_link\_node在红黑树中插入节点替换叶子节点；最后调用rb\_insert\_color为父节点着色并根据情况自旋调整二叉树的平衡状态。 | drivers/mtd/ubi/attach.c-find\_or\_add\_av：查找插入位置    tools/include/linux/rbtree.h：插入节点    tools/lib/rbtree.c：自旋调整平衡，着色 |
| 删除节点。使用举例，调用rb\_erase(&av->rb, &ai->volumes)删除卷。 | tools/lib/rbtree.c： |

### 特点：

1. 半平衡二叉搜索树，插入、删除、查询的时间复杂度为O(lgN)，适合大量数据频繁的快速查询操作
2. 每个节点颜色使用unsigned long存储，带来一定的内存开销

## 位图

### 结构定义：



### 使用：

位图和数据对象是分离的，二者之间没有像链表或红黑树一样的内存偏移关系，也即数据结构体对象中并没有和位图相关的成员。位图一般使用unsigned long数组，每个数据对象ID对应到数组在内存中的一个二进制位。

### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 为已知数量的元素申请位图。以unsigned long为对齐单位申请满足元素数量的最小空间，如果unsigned long有64位则可以存放64个元素元素。 | drivers/mtd/ubi/fastmap.c-init\_seen：    toos/include/linux/bitops.h：    toos/include/linux/kernel.h： |
| 置位。在位图中设置元素存在。 | drivers/mtd/ubi/fastmap.c-set\_seen：    tools\include\asm-generic\bitops/atomic.h： |
| 消位。在位图中设置元素不存在。 | tools/include/asm-generic/bitops/atomic.h： |
| 判断。在位图中判断元素是否存在。使用举例，test\_bit(pnum, seen)判断pnum对应的PEB是否在位图seen中存在。 | tools/include/asm-generic/bitops/non-atomic.h：    tools/include/linux/bits.h： |

### 特点：

1. 在已知元素数量而且数量不变的情况下，在O(1)时间判断一个元素否是存在，在O(1)时间从位图中抹去或添加一个元素
2. 一个元素占用一个二进制位，节约大量内存
3. 只能用于判断元素是否存在。位图中的元素之间没有任何关系（例如链表中的元素有prev-next关系，红黑树中的元素有parent-son关系）

# 同步机制

## 原子整数操作

### 结构定义：保证指令以原子的方式（要么全做完，要么都不做，中间不打断）执行，在每种架构上的实现方式也不相同。

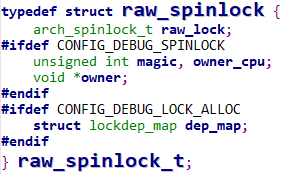


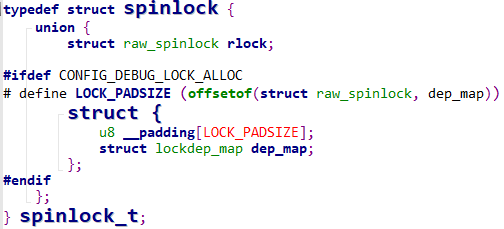
### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 读取整型变量。 | include/asm-generic/atomic-instrumented.h： |
| 设置整型变量的值为i。 | include/asm-generic/atomic-instrumented.h： |
| 原子地给v加i。 | include/asm-generic/atomic-instrumented.h： |
| 原子地给v减i。 | include/asm-generic/atomic-instrumented.h： |
| 原子自增。 | include/asm-generic/atomic-instrumented.h： |
| 原子自减。 | include/asm-generic/atomic-instrumented.h： |

## 自旋锁

### 结构定义：最多可以被一个线程持有，如果发生竞争，未持有的线程会忙等待。一般用于防止多个线程进入临界区。





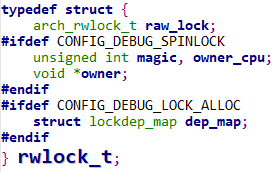
### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 获取自旋锁，其中raw\_spin\_lock的实现方式和架构有关，具体实现和相应体系结构有关。 | include/linux/spinlock.h： |
| 释放自旋锁，其中raw\_spin\_ublock的实现方式和架构有关，具体实现和相应体系结构有关。 | include/linux/spinlock.h： |

## 读写锁

### 结构定义：特殊自旋锁，读锁可以被多个线程同时持有，持有之后不能加写锁；写锁最多被一个线程持有，持有之后不能加读、写锁。一般用于对公共数据有大量并发读写操作时使用，相比于自旋锁，允许多个线程同时读取公共数据可以提高系统性能。



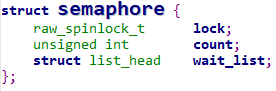


### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 获取读锁，具体实现和相应体系结构有关。 | include/linux/rwlock.h：    include/linux/rwlock\_api\_smp.h： |
| 释放读锁，具体实现和相应体系结构有关。 | include/linux/rwlock.h：    include/linux/rwlock\_api\_smp.h： |
| 获取写锁，具体实现和相应体系结构有关。 | include/linux/rwlock.h：    include/linux/rwlock\_api\_smp.h： |
| 释放写锁，具体实现和相应体系结构有关。 | include/linux/rwlock.h：    include/linux/rwlock\_api\_smp.h： |

## 信号量

### 结构定义：睡眠锁，如果互斥锁不可用线程会忙等待（占用CPU），如果信号量不可用，线程会阻塞（释放CPU），但是信号量会比自旋锁会有更大的调度开销（需要睡眠和唤醒），适合竞争不是很频繁发生，锁被长时间持有的场景。



### 操作：

|  |  |
| --- | --- |
| 操作 | 源码 |
| 初始化信号量，信号量允许多个线程同时持有，设置参数val就是同时最多持有的线程数，如果是1就代表二值信号量。 | include/linux/semaphore.h： |
| 获取信号量，让信号量计数减1。具体实现和相应体系结构有关。 | include/linux/semaphore.c： |
| 释放信号量，让信号量计数加1。具体实现和相应体系结构有关。 | include/linux/semaphore.c： |

## 其他机制

### 禁止抢占：禁止内核进行任务调度

|  |  |
| --- | --- |
| 操作 | 源码 |
| 抢占禁止。 | include/linux/preempt.h： |
| 抢占允许。 | include/linux/preempt.h： |

### 屏障：保证屏障前的所有操作一定发生在屏障后

|  |  |
| --- | --- |
| 操作 | 源码 |
| 设置屏障 | include/asm-generic/barrier.h： |

# Qemu（全系统仿真器）

## 常用参数[10]

-machine [type=]name : 指定特定机型

-smp [cpus=]n[,cores=cores] : 设置CPU数量，核数

-m [size=]megs : 设置内存大小，默认单位MB

-name string1[,process=string2] : 设置客户机名称，string1是客户机窗口名称，string2是客户机在Linux下进程名称

-uuid uuid : 设置客户机UUID

-drive file=file[,if=interface][,format=format][,cache=cache][,id=id][,file.locking=locking] : 设置一个新驱动，要指定file路径（挂载了磁盘的宿主机目录）；if是接口类型一般设置为none；format一般是raw；cache表示磁盘读写缓冲的模式，默认为writeback；id是drive标识符

-device driver[,drive=drive\_id][,id=id] : 设置一个新设备，其接口类型为driver（virtio-scsi-pci, virtio-blk, scsi-hd等），可以基于已经创建的drive id

-virtfs fsdriver[,path=path][,mount\_tag=tag][,security\_model=security\_model][,readonly] : 创建共享文件夹，fsdriver一般为local；path对应宿主机共享文件夹路径；mount\_tag是挂载点标识符，客户机启动后执行mount需要该标识符作为挂载内容；readonly表示只读

-net nic[,macaddr=mac][,model=type] : 创建一个网卡，并指定其MAC；model是网卡类型，可选virtio

-net bridge[,name=name] : 指定客户机网络为桥接模式，并指定其名字

-kernel bzImage : 使用bzImage作为客户机内核镜像

-initrd file : 使用file作为客户机的根文件系统

-append cmdline : 使用cmdline作为内核执行的命令行，可以设置console=ttyS0并配合选项-nographic将客户机串口重定向到宿主机终端

-enable-kvm : 允许使用KVM，使用硬件提供的虚拟化服务进行加速

-nographic : 默认情况qemu运行会开启一个窗口，设置该选项后qemu则只是一个命令行应用

示例：

1. 生成随机MAC：

$ echo $(printf 'DE:AD:BE:EF:%02X:%02X\n' $((RANDOM%256)) $((RANDOM%256)))

DE:AD:BE:EF:9B:5E

1. 在宿主机生成客户机物理磁盘对应的文件[11]

$ truncate -s 10G sda\_disk

$ truncate -s 5G sdb\_disk

$ truncate -s 5G sdc\_disk

$ ls -l

-rw-rw-r-- 1 freg freg 10737418240 May 14 20:48 sda\_disk

-rw-rw-r-- 1 freg freg 5368709120 May 14 20:48 sdb\_disk

-rw-rw-r-- 1 freg freg 5368709120 May 14 20:48 sdc\_disk

1. 制作根文件系统[12]

$ dd if=/ of=/home/freg/qemu/root\_fs/ubuntu.iso bs=4096 count=1024

1. 创建虚拟网卡，使用设定的网段作为客户机IP[13][14]

$ br=qemu\_czh

$ dev=eth0

$ ip=192.168.88.1

$ sudo brctl addbr $br

$ sudo brctl stp $br on

$ sudo ip addr add $ip dev $br

$ sudo brctl addif $br $dev

$ sudo ip link set $br up

寻找虚拟网卡对应的allow配置文件：[15]

$ sudo strace -f -e openat,open /fs\_harden/sw/x86/bin/qemu-system-x86\_64 -enable-kvm -smp 8 -m 4096 -kernel /home/freg/qemu/kernel\_img/bzImage -net nic,model=virtio,macaddr=DE:AD:BE:EF:9B:5E -net bridge,br=qemu\_czh -device virtio-scsi-pci -drive file=/home/freg/qemu/root\_fs/rootfs.gz,if=none,cache=none,id=root -device virtio-blk,drive=root,id=d\_root -drive file=/home/freg/qemu/disk/sda\_disk,if=none,format=raw,id=dd\_1 -device scsi-hd,drive=dd\_1,id=disk\_1 -drive file=/home/freg/qemu/disk/sdb\_disk,if=none,format=raw,id=dd\_2 -device scsi-hd,drive=dd\_2,id=disk\_2 -drive file=/home/freg/qemu/disk/sdc\_disk,if=none,format=raw,id=dd\_3 -device scsi-hd,drive=dd\_3,id=disk\_3 -virtfs local,path=/home/freg/qemu/share,mount\_tag=share\_dir,security\_model=none -append "console=ttyS0 IP=192.168.88.66 root=/dev/vda1 rw" -nographic 2>&1 |grep bridge.con

[pid 42878] open("/fs\_harden/sw/x86/etc/qemu/bridge.conf", O\_RDONLY) = 3

在/fs\_harden/sw/x86/etc/qemu/bridge.conf文件添加allow qemu\_czh

查看虚拟网卡：

$ ls /sys/devices/virtual/net/

qemu\_ci qemu\_czh qemu\_fs

删除虚拟网卡：

$ sudo ip link set $br down

$ sudo brctl delbr $br

1. 编译内核源码，得到内核镜像[16][8]

$ make menuconfig

$ make -j100

生成的内核镜像在kernel/arch/x86/boot/bzImage

编译内核模块：

$ make modules -j144

$ make modules\_install $DEST\_DIR –j144

1. 创建共享文件夹

$ mkdir /home/freg/qemu/share

1. 使用qemu启动虚拟机

/fs\_harden/sw/x86/bin/qemu-system-x86\_64 -enable-kvm -smp 8 -m 4096 -kernel /home/freg/qemu/kernel\_img/bzImage -net nic,model=virtio,macaddr=DE:AD:BE:EF:9B:5E -net bridge,br=qemu\_czh -device virtio-scsi-pci -drive file=/home/freg/qemu/root\_fs/rootfs.gz,if=none,cache=none,id=root -device virtio-blk,drive=root,id=d\_root -drive file=/home/freg/qemu/disk/sda\_disk,if=none,format=raw,id=dd\_1 -device scsi-hd,drive=dd\_1,id=disk\_1 -drive file=/home/freg/qemu/disk/sdb\_disk,if=none,format=raw,id=dd\_2 -device scsi-hd,drive=dd\_2,id=disk\_2 -drive file=/home/freg/qemu/disk/sdc\_disk,if=none,format=raw,id=dd\_3 -device scsi-hd,drive=dd\_3,id=disk\_3 -virtfs local,path=/home/freg/qemu/share,mount\_tag=share\_dir,security\_model=none -append "console=ttyS0 IP=192.168.88.66 root=/dev/vda1 rw" -nographic

可以把步骤E中的DEST\_DIR作为宿主机的共享文件夹-virtfs local,path=$DEST\_DIR,mount\_tag=share\_dir,readonly,security\_model=none，客户机开机后使用mount -t 9p -o trans=virtio share\_dir /tmp挂载共享文件夹到/tmp

# 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>.
10. Qemu-3.1.50用户手册. <https://qemu.weilnetz.de/doc/qemu-doc.html>.
11. Linux truncate 命令. <https://blog.csdn.net/u010709783/article/details/77896017>.
12. 根文件系统制作. <https://blog.csdn.net/master5512/article/details/69055662>.
13. 创建虚拟网卡. <https://blog.csdn.net/hzhsan/article/details/44677867>.
14. 允许使用虚拟网桥. <https://blog.christophersmart.com/2016/08/31/configuring-qemu-bridge-helper-after-access-denied-by-acl-file-error/>.
15. 使用strace跟踪系统调用. <https://www.linuxidc.com/Linux/2018-01/150654.htm>.
16. 内核编译配置menuconfig. <https://blog.csdn.net/tiger15605353603/article/details/81323848>.
17. 使用mtd-util基于MTD创建UBI. <https://www.cnblogs.com/pengdonglin137/p/3415663.html>.
18. 使用nandsim虚拟MTD设备. <http://www.linux-mtd.infradead.org/faq/nand.html>.
19. UBI介绍. <https://www.cnblogs.com/wahaha02/p/4814698.html>.
20. Linux内核源码. <https://www.kernel.org/>.