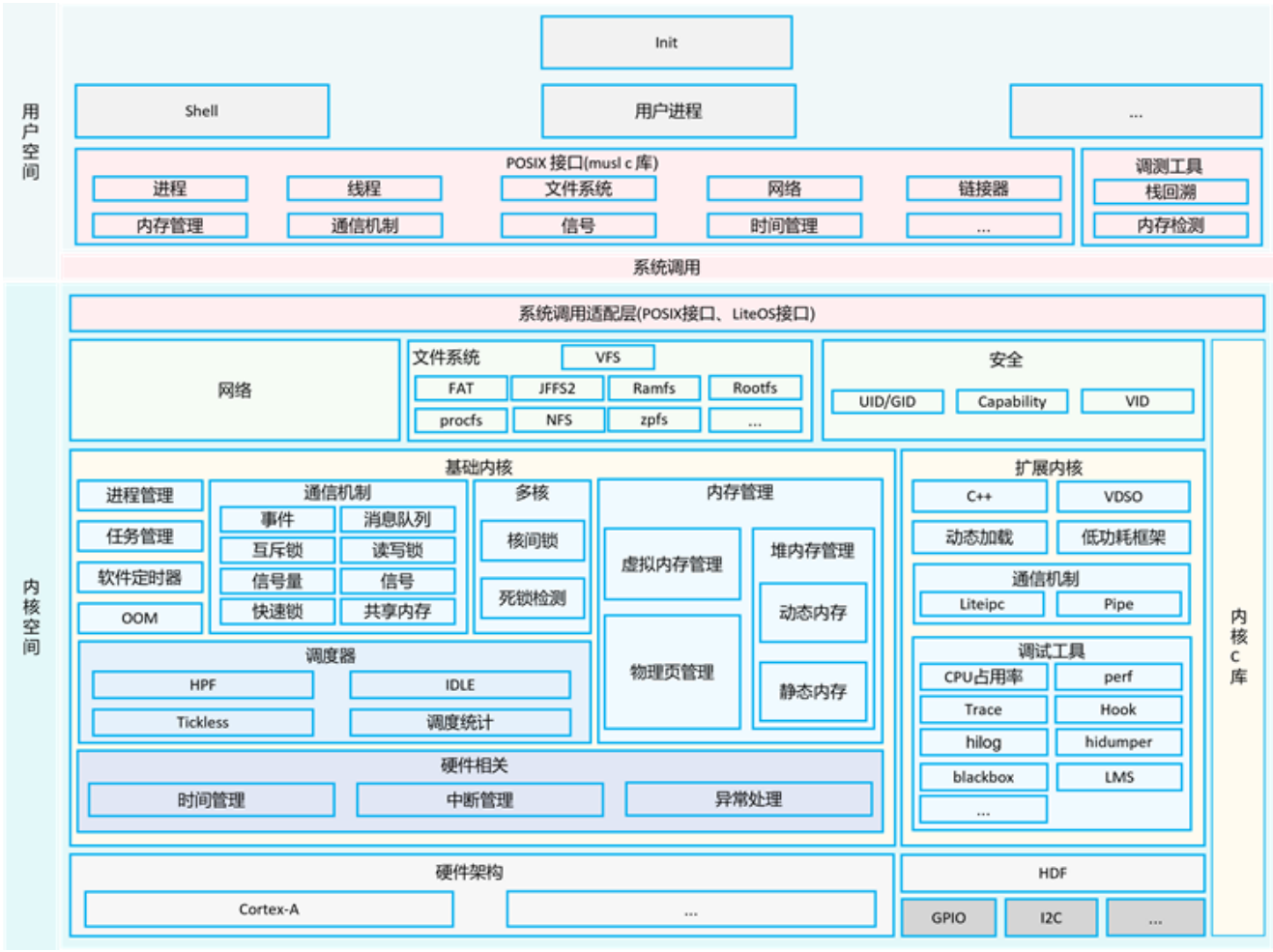
鸿蒙文件系统报告



请根据所提供的源代码，完成以下两个任务：

（1）文件系统结构分析：梳理文件系统的主要模块与组件，如文件系统的挂载与卸载、目录管理、文件存储管理等。分析各模块之间的交互关系与数据流，绘制模块交互图或数据流图。

（2）选取某一个文件系统进行关键代码剖析：分析出关键代码段，如文件打开、关闭、读写、目录遍历等功能的实现代码。对选取的代码段进行模块化分析，解释其功能、使用的数据结构、算法原理及异常处理机制。

文件系统结构分析

简介

文件系统是操作系统中负责管理持久数据的子系统，基本数据单位是文件，它的目的是对磁盘上的文件进行组织管理。

为了满足设备在应用程序之间的共享，磁盘需要支持数据的持久化，将数据能独立于进程之外被长期存储引入了文件。

文件是存储设备 (字节序列) 的虚拟化，是一个可以读/写的动态字节序列。文件系统就是负责管理和维护所有文件的系统。

文件系统按类型可分成以下四种:

- **磁盘文件系统：**是一种设计用来利用数据存储设备来保存计算机文件的文件系统，最常用的数据存储设备是磁盘驱动器，可以直接或者间接地连接到计算机上。例如：`FAT`、`exFAT`、`NTFS`、`HFS`、`HFS+`、`ext2`、`ext3`、`ext4`、`ODS-5`、`btrfs`、`XFS`、`UFS`、`ZFS`。有些文件系统是行程文件系统（也有译作日志文件系统）或者追踪文件系统。

- **闪存文件系统：**尽管磁盘文件系统也能在闪存上使用，但闪存文件系统是闪存设备的首选，理由如下：
  - 擦除区块
  - 随机访问
  - 写入平衡（Wear levelling）：闪存中经常写入的区块往往容易损坏。闪存文件系统的设计可以使数据均匀地写到整个设备。 日志文件系统具有闪存文件系统需要的特性，这类文件系统包括`JFFS2`和`YAFFS`。也有为了避免日志频繁写入而导致闪存寿命衰减的非日志文件系统，如`exFAT`。

    `JFFS2`（全称： Journalling Flash File System Version2），是`Redhat`公司开发的闪存文件系统，其前身是`JFFS`，最早只支持`NOR Flash`， 自2.6版以后开始支持`NAND Flash`，适合使用于嵌入式系统。
- **伪文件系统：**启动时动态生成的文件系统，包含有关当前正在运行的内核的许多信息、配置和日志，由于它们放置在易失性存储器中，因此它们仅在运行时可用，而在关闭时消失。这些伪文件常**挂载**到以下目录: `sysfs (/sys)`， `procfs (/proc)`， `debugfs (/sys/kernel/debug)`， `configfs (/sys/kernel/config)`， `tracefs (/sys/kernel/tracing)`， `tmppfs`（`/dev/shm`， `/run`， `/sys/fs/cgroup`， `/tmp/`， `/var/volatile`， `/run/user/<id>`)， `devtmpfs`（`/dev`）
  - `procfs` 是 进程文件系统 (file system) 的缩写，用于通过内核访问进程信息。这个文件系统通常被挂载到 `/proc` 目录。由于 `/proc` 不是一个真正的文件系统，它也就不占用存储空间，只是占用有限的内存。
  - `tmpfs` （temporary file system） 是类Unix系统上暂存档存储空间的常见名称，通常以挂载文件系统方式实现，并将资料存储在易失性存储器而非永久存储设备中。所有在tmpfs上存储的资料在理论上都是暂时借放的，那也表示说，文件不会创建在硬盘上面。一旦重启，所有在tmpfs里面的资料都会消失不见。
  - `Sysfs` 是Linux 2。6所提供的一种虚拟文件系统。这个文件系统不仅可以把设备（`devices`）和驱动程序（`drivers`）的信息从内核输出到用户空间，也可以用来对设备和驱动程序做设置。 `sysfs`的目的是把一些原本在`procfs`中的，关于设备的部分，独立出来，以‘设备层次结构架构’（device tree）的形式呈现。
  - `devtmpfs` 是在 `Linux`核心启动早期建立一个初步的 `/dev`，令一般启动程序不用等待 `udev`，缩短`GNU/Linux`的开机时间。将设备也看成为文件，突出了`Linux`文件系统的特点：一切皆文件或目录。
- **网络文件系统:** `NFS`，(Network File System）是一种将远程主机上的分区（目录）经网络挂载到本地系统的一种机制，是一种分布式文件系统，力求客户端主机可以访问服务器端文件，并且其过程与访问本地存储时一样，它由Sun公司开发，于1984年发布。它的特点是将网络也看出了文件，再次体现一切皆文件的说法。

文件系统的主要模块与组件

文件系统的挂载与卸载

将一个文件系统与系统的一个特定目录（称为挂载点）关联起来，从而使该文件系统的内容可以通过该目录访问。挂载过程使得存储设备（如硬盘分区、USB驱动器、网络共享等）上的数据能够被操作系统和用户如同访问本地目录一样进行读写操作盘。

在Linux和其他类Unix系统中，`mount` 命令用于挂载文件系统。它需要指定要挂载的设备文件（如 `/dev/sda1`）和挂载点（如 `/mnt/usb`）。此外，还可以指定文件系统的类型（如 `ext4`，`ntfs`，`vfat`）和挂载选项（如读写权限、权限掩码等）。

系统启动时，许多操作系统（包括Linux和macOS）会自动挂载一些预定义的文件系统，这些信息通常保存在配置文件中，如Linux的 `/etc/fstab` 或systemd的 `/etc/systemd/system/*.mount` 单元文件。Windows系统也有类似机制，通过注册表或组策略设定开机自动挂载的驱动器。

与挂载相对应，`umount` 命令用于卸载文件系统，即将文件系统从挂载点分离。这通常在不再需要访问该文件系统时执行，以释放资源或允许安全移除存储设备。

目录与存储磁盘块管理

FAT

- 磁盘块管理：
  - **文件分配表**：FAT是核心组件，它是一个数组，每个元素代表一个磁盘簇（连续的扇区集合），用来记录该簇是否被分配以及它之后的簇是否属于同一个文件。
  - **链式分配**：文件存储在磁盘上时，通过一系列簇链接起来。文件的第一个簇号记录在目录项中，后续簇通过FAT表中的指针链接。
  - **碎片问题**：随着文件的频繁增删改，容易产生磁盘碎片，因为文件可能被分散存储在多个不连续的簇中，影响读写效率。
- 目录:
  - **简单层级结构**：FAT文件系统采用简单的树形目录结构，每个目录项包含文件名、文件属性、起始簇号等信息。目录项按照顺序排列，没有硬链接或符号链接的概念。
  - **短文件名限制**：传统FAT文件系统（如FAT16和FAT32）对文件名长度有限制，通常是8.3格式（即8个字符的文件名加上3个字符的扩展名）。

ext2/UNIX 文件系统

按对象方式集中存储文件/目录元数据

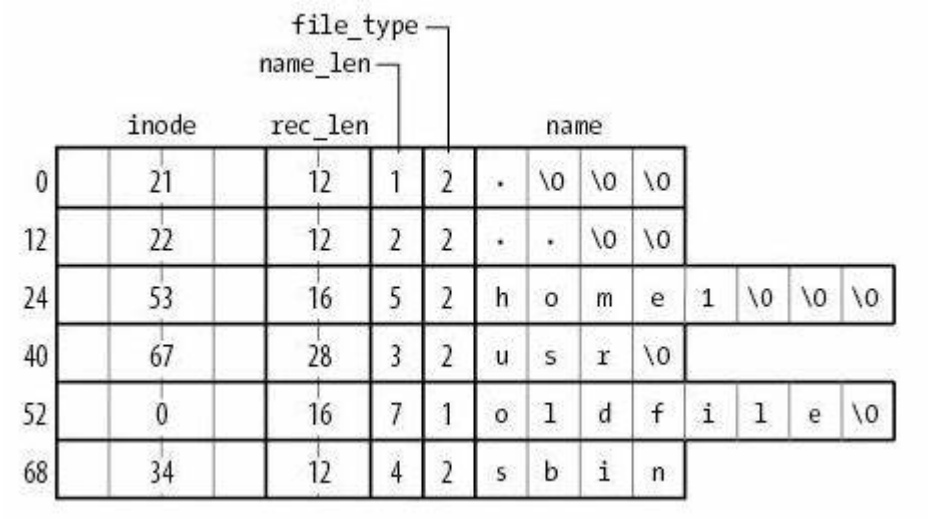
- 增强局部性 (更易于缓存)
- 支持链接

inode：ext 2 使用 inode（索引节点）来存储文件的元数据，包括权限、大小、创建修改时间、链接数以及数据块指针。每个文件或目录都有一个唯一的 inode 号。

ext2 目录文件：

与 FAT 本质相同：在文件上建立目录的数据结构

- 注意到 inode 统一存储
  - 目录文件中存储文件名到 inode 编号的 key-value 映射



- 磁盘块管理:
  - **块组**：为提高管理效率，ext2将磁盘空间划分为多个块组，每个块组包含自己的inode表、数据块、超级块副本等，使得文件系统可以独立地管理不同区域。
  - **inode和数据块分配**：inode中直接或间接地存储了数据块的地址，支持多级间接寻址，以适应不同大小的文件。小文件可以直接在inode中记录数据。
  - **超级块和日志**：超级块包含文件系统的整体信息，如块大小、inode数量等。虽然原始ext2没有日志功能，但后续的ext3和ext4通过日志记录增强了崩溃恢复能力。

错误处理与恢复

崩溃一致性（Crash Consistency）和崩溃恢复（Crash Recovery）是文件系统设计中的两个关键概念，旨在确保即使在系统突然崩溃或电源故障的情况下，文件系统也能维持一种已知的、一致的状态，并且能够在重启后恢复到一个正确的运行状态。

> 崩溃一致性 Crash Consistency:
>
> Move the file system from one consistent state (e.g., before the file got appended to) to another
>
> atomically (e.g., after the inode, bitmap, and new data block have been written to disk).

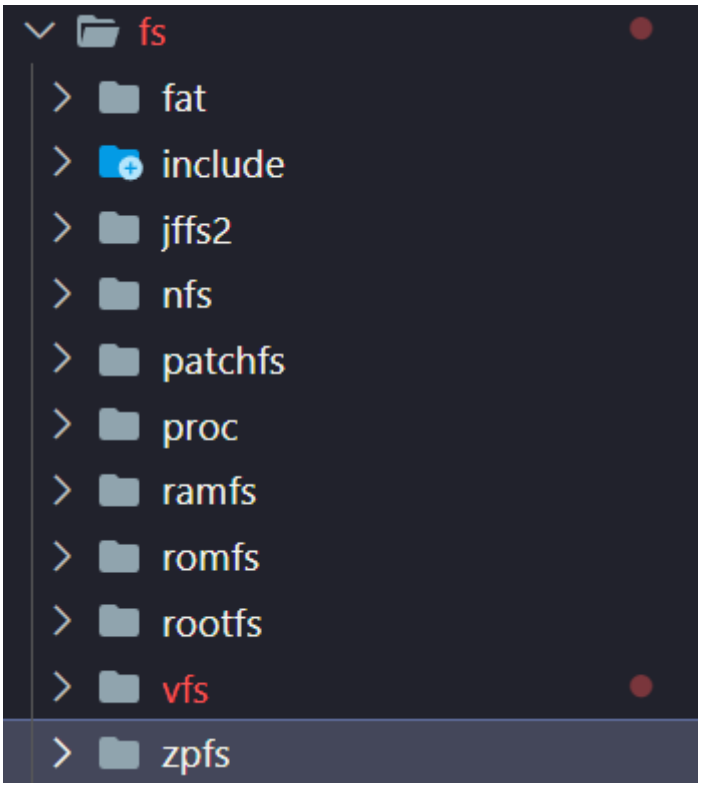
崩溃一致性关注的是系统在遭遇意外停止（如电源故障、硬件错误或软件崩溃）时，文件系统数据是否仍能保持一种逻辑上连贯和可预测的状态。为了实现这一点，文件系统需要确保正在进行的写操作要么完全完成（原子提交），要么根本不开始（原子放弃）。这意味着：

- **事务日志**：许多文件系统采用事务日志来记录所有更改的预览或更改本身，这样在系统恢复时可以根据日志内容重做或撤销未完成的操作。
- **写前日志（Journaling）**：是一种常见的技术，它要求所有的修改操作首先被记录到一个特殊的日志区域，之后才实际更新文件或元数据。这样，在系统崩溃后，可以通过重放日志来恢复文件系统的状态。
- **同步策略**：通过控制何时将缓存中的数据刷写到持久存储上，可以平衡性能与一致性需求。例如，同步写可以确保数据即时写入磁盘，但会降低写操作的速度。

*崩溃恢复*是指在系统经历崩溃事件后，通过一系列步骤使文件系统回到一个一致且可用的状态的过程。这个过程通常涉及：

1. **检查文件系统一致性**：系统启动时，会运行文件系统检查工具（如 `fsck`）来扫描并修复任何不一致的地方，比如损坏的inode、丢失的链接或不匹配的文件大小信息。
2. **重放事务日志**：如果文件系统采用了日志技术，系统会读取日志文件，重新执行那些在崩溃时尚未完成的事务，确保所有变更都被正确应用。
3. **缓存与内存状态同步**：恢复过程中可能还需要将内存中的文件系统缓存内容与磁盘上的实际状态进行同步，避免数据丢失或不一致。
4. **标记已恢复**：一旦恢复过程完成，系统会设置一个标志，表明文件系统已经过检查和修复，可以安全地进行下一步操作。

LiteOS-A 文件系统代码剖析



LiteOS-A 内核原生支持的文件系统有FAT、JFFS 2、NFS、RAMFS、ROOTFS、ZPFS 等

LiteOS-A中使用**VFS**作为各个文件系统的粘合层，而VFS在OpenHarmony内核中采用树结构实现，树中的每一个节点都是Vnode结构体。*VFS提供统一的抽象接口用于屏蔽文件系统之间的差异，其提供三大操作接口用于统一不同文件系统调用不同接口的现状。*

VFS提供的三大操作接口：

- `VnodeOps`
- `MountOps`
- `file_operations_vfs`

VFS 基本模块

Vnode 管理

`VnodeOps` 用于控制 `Vnode` 节点， `MountOps` 控制挂载点， `file_operations_vfs` 提供常用的文件接口。

`vnode` 是内核内存中的一个对象，它使用 UNIX 风格的文件接口（打开、读取、写入、关闭、readdir 等）。 `Vnode` 可以代表文件、目录、管道、套接字、块设备、字符设备。 `vnode` 是 `BSD`

的叫法，鸿蒙沿用了 `BSD` 的称呼， `Linux` 的叫法是 `inode`。

在 `fs/vfs/include/vnode.h` 我们可以看到 LiteOS-A 中 `vnode` 的结构体定义：

```
struct Vnode {
    enum VnodeType type;                /* vnode type | 节点类型 (文件|目录|链接...)*/
    int useCount;                       /* ref count of users | 节点引用(链接)数，即有多少文件名指向这个vnode,即上层理解的硬链接数*/
    uint32_t hash;                      /* vnode hash | 节点哈希值*/
    uint uid;                           /* uid for dac | 文件拥有者的User ID*/
    uint gid;                           /* gid for dac | 文件的Group ID*/
    mode_t mode;                        /* mode for dac | chmod 文件的读、写、执行权限*/
    LIST_HEAD parentPathCaches;         /* pathCaches point to parents | 指向父级路径缓存,上面的都是当了爸爸节点*/
    LIST_HEAD childPathCaches;          /* pathCaches point to children | 指向子级路径缓存,上面都是当了别人儿子的节点*/
    struct Vnode *parent;               /* parent vnode | 父节点*/
    struct VnodeOps *vop;               /* vnode operations | 相当于指定操作Vnode方式 (接口实现|驱动程序)*/
    struct file_operations_vfs *fop;    /* file operations | 相当于指定文件系统*/
    void *data;                         /* private data | 文件数据block的位置,指向每种具体设备私有的成员，例如 ( drv_data | nfsnode | ....)*/
    uint32_t flag;                      /* vnode flag | 节点标签*/
    LIST_ENTRY hashEntry;               /* list entry for bucket in hash table | 通过它挂入哈希表 g_vnodeHashEntrys[i], i:[0,g_vnodeHashMask]*/
    LIST_ENTRY actFreeEntry;            /* vnode active/free list entry | 通过本节点挂到空闲链表和使用链表上*/
    struct Mount *originMount;          /* fs info about this vnode | 自己所在的文件系统挂载信息*/
    struct Mount *newMount;             /* fs info about who mount on this vnode | 其他挂载在这个节点上文件系统信息*/
    char *filePath;                     /* file path of the vnode */
    struct page_mapping mapping;        /* page mapping of the vnode */
#ifdef LOSCFG_MNT_CONTAINER
    int mntCount;                       /* ref count of mounts */
#endif
};
```

其中 `VnodeOps *vop` 这是对 `vnode` 的操作， `vnode` 本身也是数据，存储在索引表中，记录了用户，用户组，权限，时间等信息，这部分信息是可以修改的，就需要接口来维护，便是 `VnodeOps`。

`VnodeOps` 的定义如下：

```
struct VnodeOps {
    int (*Create)(struct Vnode *parent,  const char *name,  int mode,  struct Vnode **vnode);//创建节点
  int (*Lookup)(struct Vnode *parent,  const char *name,  int len,  struct Vnode **vnode);//查询节点
  //Lookup向底层文件系统查找获取inode信息
  int (*Open)(struct Vnode *vnode,  int fd,  int mode,  int flags);//打开节点
  int (*Close)(struct Vnode *vnode);//关闭节点
  int (*Reclaim)(struct Vnode *vnode);//回收节点
  int (*Unlink)(struct Vnode *parent,  struct Vnode *vnode,  const char *fileName);//取消硬链
```

```
接
  int (*Rmdir)(struct Vnode *parent,  struct Vnode *vnode,  const char *dirName);//删除目录节
点
  int (*Mkdir)(struct Vnode *parent,  const char *dirName,  mode_t mode,  struct Vnode
**vnode);//创建目录节点
  int (*Readdir)(struct Vnode *vnode,  struct fs_dirent_s *dir);//读目录节点
  int (*Opendir)(struct Vnode *vnode,  struct fs_dirent_s *dir);//打开目录节点
  int (*Rewinddir)(struct Vnode *vnode,  struct fs_dirent_s *dir);//定位目录节点
  int (*Closedir)(struct Vnode *vnode,  struct fs_dirent_s *dir);//关闭目录节点
  int (*Getattr)(struct Vnode *vnode,  struct stat *st);//获取节点属性
  int (*Setattr)(struct Vnode *vnode,  struct stat *st);//设置节点属性
  int (*Chattr)(struct Vnode *vnode,  struct IATTR *attr);//改变节点属性(change attr)
  int (*Rename)(struct Vnode *src,  struct Vnode *dstParent,  const char *srcName,  const char
*dstName);//重命名
  ....
}
```

全部由函数指针组成，所有函数基本都是对索引节点 (索引页) 的增删改查操作。所有文件系统都要实现对应的接口。

我们可以查看LiteOS-A 中FAT 文件系统的实现

```
// proc 对 VnodeOps 接口实现,暂没有实现创建节点的功能.
static struct VnodeOps g_procfsVops = {
    .Lookup = VfsProcfsLookup,
    .Getattr = VfsProcfsStat,
    .Readdir = VfsProcfsReaddir,
    .Opendir = VfsProcfsOpendir,
    .Closedir = VfsProcfsClosedir,
    .Truncate = VfsProcfsTruncate,
    .Readlink = VfsProcfsReadlink,
#ifdef LOSCFG_KERNEL_PLIMITS
    .Mkdir = VfsProcfsMkdir,
    .Rmdir = VfsProcfsRmdir,
#endif
};
```

将其中成员指向 `fatfs.c` 中的具体函数。

同理 `file_operations_vfs` 记录了对单一文件/设备的操作接口如 `open` ，`close`，`read`，`write` 等等。

Mount挂载管理

当前OpenHarmony内核中，对系统中所有挂载点通过链表进行统一管理。挂载点结构体中，记录了该挂载分区内的所有Vnode。当分区卸载时，会释放分区内的所有Vnode。

```
static LIST_HEAD *g_mountList = NULL;//挂载链表，上面挂的是系统所有挂载点
struct Mount {
    LIST_ENTRY mountList;               /* mount list */    //通过本节点将Mount挂到全局Mount链
表上
    const struct MountOps *ops;         /* operations of mount */ //挂载操作函数
```

```
    struct Vnode *vnodeBeCovered;       /* vnode we mounted on */ //要被挂载的节点 即
/bin1/vs/sd 对应的 vnode节点
    struct Vnode *vnodeCovered;         /* syncer vnode */   //要挂载的节点 即/dev/mmcblk0p0 对
应的 vnode节点
    struct Vnode *vnodeDev;             /* dev vnode */
    LIST_HEAD vnodeList;                /* list of vnodes */  //链表表头
    int vnodeSize;                      /* size of vnode list */ //节点数量
    LIST_HEAD activeVnodeList;          /* list of active vnodes */ //激活的节点链表
    int activeVnodeSize;                /* szie of active vnodes list *///激活的节点数量
    void *data;                         /* private data */ //私有数据，可使用这个成员作为一个指
向它们自己内部数据的指针
    uint32_t hashseed;                  /* Random seed for vfs hash */ //vfs 哈希随机种子
    unsigned long mountFlags;           /* Flags for mount */ //挂载标签
    char pathName[PATH_MAX];            /* path name of mount point */ //挂载点路径名称
/bin1/vs/sd
    char devName[PATH_MAX];             /* path name of dev point */  //设备名称 /dev/mmcblk0p0
};
//分配一个挂载点
struct Mount* MountAlloc(struct Vnode* vnodeBeCovered,  struct MountOps* fsop)
{
    struct Mount* mnt = (struct Mount*)zalloc(sizeof(struct Mount));//申请一个mount结构体内
存，小内存分配用 zalloc
    if (mnt == NULL) {
        PRINT_ERR("MountAlloc failed no memory!\n");
        return NULL;
    }

    LOS_ListInit(&mnt->activeVnodeList);//初始化激活索引节点链表
    LOS_ListInit(&mnt->vnodeList);//初始化索引节点链表

    mnt->vnodeBeCovered = vnodeBeCovered;//设备将装载到vnodeBeCovered节点上
    vnodeBeCovered->newMount = mnt;//该节点不再是虚拟节点，而作为 设备结点
#ifdef LOSCFG_DRIVERS_RANDOM //随机值 驱动模块
    HiRandomHwInit();//随机值初始化
    (VOID)HiRandomHwGetInteger(&mnt->hashseed);//用于生成哈希种子
    HiRandomHwDeinit();//随机值反初始化
#else
    mnt->hashseed = (uint32_t)random(); //随机生成哈子种子
#endif
    return mnt;
}
```

PathCache

`PathCache` 是路径缓存，它通过哈希表存储，利用父节点 `Vnode` 的地址和子节点的文件名，可以从 `PathCache` 中快速查找到子节点对应的 `Vnode`。当前 `PageCache` 仅支持缓存二进制文件，在初次访问文件时通过 `mmap` 映射到内存中，下次再访问时，直接从 `PageCache` 中读取，可以提升对同一个文件的读写速度。另外基于 `PageCache` 可实现以文件为基底的进程间通信。下图展示了文件/目录的查找流程。

Procfs 实现

procfs文件系统中每个目录或文件都是一个Vnode，也可以理解为一个entry。ProcDirEntry中的subdir指向的目录中的一个子项，其本质是一个单向链表的形式，并且采用头插法的形式进行节点的插入。

Procfs的注册过程

1. 向系统注册文件系统入口函数：

`LOS_MODULE_INIT(ProcFsInit, LOS_INIT_LEVEL_KMOD_EXTENDED);`

2. 向VFS文件系统表注册系统名以及实现的接口等：(VnodeOps 前面已经列出)

```
/// proc 对 MountOps 接口实现
const struct MountOps procfs_operations = {
    .Mount = VfsProcfsMount,//装载
    .Unmount = NULL,
    .Statfs = VfsProcfsStatfs,//统计信息
};
// proc 对 file_operations_vfs 接口实现
static struct file_operations_vfs g_procfsFops = {
    .read = VfsProcfsRead,  // 最终调用 ProcFileOperations -> read
    .write = VfsProcfsWrite,// 最终调用 ProcFileOperations -> write
    .open = VfsProcfsOpen,  // 最终调用 ProcFileOperations -> open
    .close = VfsProcfsClose // 最终调用 ProcFileOperations -> release
};

//文件系统注册入口
FSMAP_ENTRY(procfs_fsmap, "procfs", procfs_operations, FALSE, FALSE);
#endif
```

Procfs挂载

`mount -R -t procfs [Dir_Path]`

mount的系统调用间接调用procfs的mount接口。

用户输入挂载命令后，引发系统调用 `SysMount` 开始逐层调用：

1. `SysMount(``const char *source, const char *target, const char *filesystemtype, unsigned long mountflags,``const void *data)` 将路径，文件系统等转化之后调用mount 函数
2. `mount(sourceRet, targetRet, (filesystemtype ? fstypeRet : NULL), mountflags, dataRet)` 函数内会找到指定的文件系统

- `fsmap = mount_findfs(filesystemtype)`
- `mops = fsmap->fs_mops` 为mount节点指定mount的接口函数
- `VnodeLookup(target, &mountpt_vnode, 0)` 找到挂载目录对应的Vnode并且设置文件系统相关信息
  - `VnodeLookupAt(path, vnode, flags, NULL)` 对目录变成绝对路径并且从全局Vnode链表中开始找
  - `PreProcess(path, &startVnode, &normalizedPath)`
    - `vfs_normalize_path (NULL, originPath, &absolutePath)`
- `mnt = MountAlloc(mountpt_vnode, (``struct MountOps*)mops)`

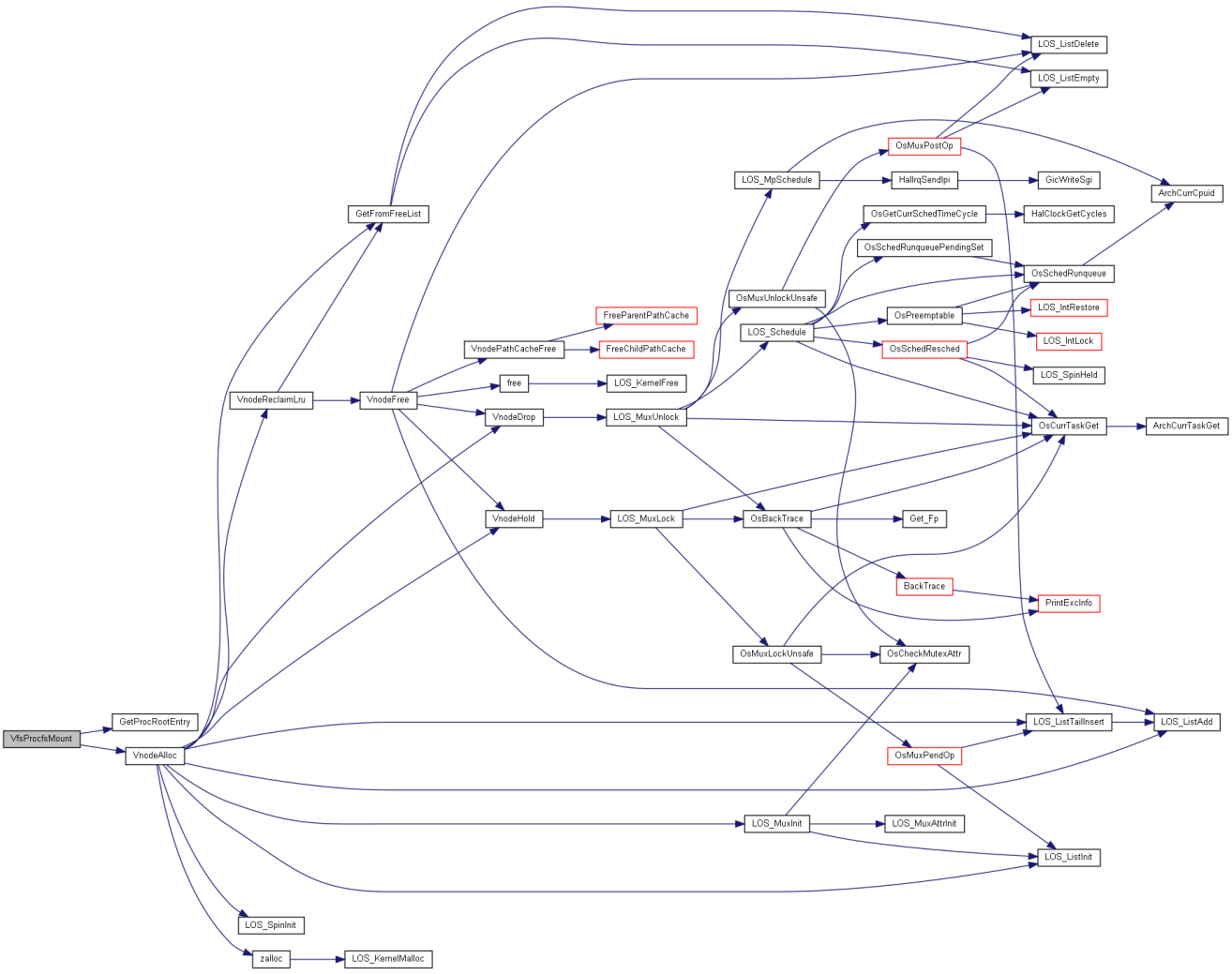
- `mops->Mount(mnt, device, data)` 调用 mops 接口的Mount 函数
  - `VfsProcfsMount(``struct Mount *mnt, struct Vnode *device, const void *data)` 进入具体的 **procfs 文件系统的 mount 函数**
    - `VnodeAlloc(&g_procfsVops, &vp);` 生成一个Vnode用于挂载mount节点和procfs文件系统的root节点
    - `root = GetProcRootEntry();` 获取procfs文件系统的根节点
    - `vp->data = root; vp->originMount = mnt;` 将vp挂载在挂载目录所对应的mount节点上
    - `mnt->vnodeCovered = vp;` mount节点挂载的Vnode是该文件系统，方便后续在mount链表中找挂载点
    - ...

Procfs 的文件系统挂载函数 `VfsProcfsMount`

```
int VfsProcfsMount(struct Mount *mnt, struct Vnode *device, const void *data)
{
    struct Vnode *vp = NULL;
    int ret;
    spin_lock_init(&procfsLock);
    procfsInit = true;      //已初始化 /proc 模块
    ret = VnodeAlloc(&g_procfsVops, &vp);//分配一个节点
    if (ret != 0) {
        return -ENOMEM;
    }
    struct ProcDirEntry *root = GetProcRootEntry();
    vp->data = root;
    vp->originMount = mnt;//绑定mount
    vp->fop = &g_procfsFops;//指定文件系统
    mnt->data = NULL;
    mnt->vnodeCovered = vp;
    vp->type = root->type;
    if (vp->type == VNODE_TYPE_DIR) {//目录节点
        vp->mode = S_IFDIR | PROCFS_DEFAULT_MODE;//贴上目录标签
    } else {
        vp->mode = S_IFREG | PROCFS_DEFAULT_MODE;//贴上文件标签
    }
    return LOS_OK;
}
```

此函数用于挂载procfs文件系统。关键步骤如下：

- 初始化procfs锁(`spin_lock_init`)以确保并发访问的安全性。
- 设置procfs模块已初始化标志。
- 分配一个新的Vnode（虚拟节点）结构体，这是文件系统中的基本对象，代表一个文件、目录或其他对象。
- 为新Vnode分配并设置`ProcDirEntry`（代表/proc目录下的条目）作为其数据指针，同时绑定到指定的挂载点（`mnt`）、指定文件操作(`g_procfsFops`)和类型（基于`ProcDirEntry`的类型决定是目录还是文件）。
- 最后，更新挂载点的信息，完成挂载过程。



VfsProcfsStat

```
int VfsProcfsStat(struct Vnode *node, struct stat *buf)
{
    VnodeHold();
    struct ProcDirEntry *entry = VnodeToEntry(node);
    if (entry == NULL) {
        VnodeDrop();
        return -EPERM;
    }
    (void)memset_s(buf, sizeof(struct stat), 0, sizeof(struct stat));
    buf->st_mode = entry->mode;
    VnodeDrop();
    return LOS_OK;
}
```

此函数用于获取procfs下某一节点（文件或目录）的状态信息。其流程包括：

- 首先保持Vnode引用计数（防止Vnode被提前释放）。
- 将Vnode转换为对应的`ProcDirEntry`结构体，失败则返回错误。
- 清零并初始化`stat`结构体，然后根据`ProcDirEntry`中的模式信息填充`st_mode`字段，最后释放Vnode引用计数并返回成功。

这些函数共同构成了LiteOS-A中procfs模块的基础，允许系统以文件形式暴露内核和进程信息，便于监控和调试。

VnodeOps

VfsProcfsLookup

函数的作用是在/proc文件系统中查找指定名称的文件或目录。下面是详细的步骤解释：

1. **参数校验**：首先检查传入的参数是否有效，包括父Vnode（`parent`）、文件或目录名称（`name`）的指针及长度（`len`），以及输出Vnode指针的指针（`vpp`）是否为NULL，任何一项无效则返回 `-EINVAL` 错误码。
2. **转换父节点**：将父Vnode转换为对应的 `ProcDirEntry` 结构体（`entry`），如果转换失败（即 `parent` 不是有效的procfs目录节点），则返回 `-ENODATA`。
3. **遍历子目录**：从 `parent` 所代表的目录的子目录(`subdir`)开始，进入一个循环遍历链表。每次迭代检查当前 `entry` 是否匹配传入的 `name` 和 `len`。匹配逻辑由 `EntryMatch` 函数实现，通常会比较名称字符串。
4. **匹配与返回**：
   - 如果找到匹配的 `entry`，则通过 `EntryToVnode` 将其转换为Vnode结构体，并赋值给输出参数 `*vpp`。如果转换失败（即内存分配失败），返回 `-ENOMEM`。
   - 成功转换后，还需设置新Vnode的 `originMount` 为父Vnode的挂载点，以及设置其父节点为当前遍历的父目录，确保文件系统层次结构的完整性。
   - 最终，如果一切顺利，函数返回 `LOS_OK` 表示查找成功。

VfsProcfsReaddir

此函数用于读取指定目录下的条目（目录项）到用户提供的缓冲区中。关键步骤如下：

- **参数校验**：确保输入的 `dir` 非空，且Vnode类型为目录。
- **持有Vnode**：调用 `VnodeHold` 确保Vnode不会被释放。
- **转换PDE**：将Vnode转换为 `ProcDirEntry` 结构体。
- **循环读取**：对每个待读取的目录项，分配缓冲区 `buffer`，调用 `ReadProcFile` 读取目录项名称到缓冲区。然后，将缓冲区内容安全地复制到 `dir->fd_dir[i]` 中，并更新目录项的偏移量、长度等信息。每次处理完一个条目后释放缓冲区。
- **释放Vnode**：完成所有读取后，调用 `VnodeDrop` 释放Vnode持有。
- **返回结果**：成功读取的条目数量。

VfsProcfsOpendir

这个函数用于打开一个目录，以便后续的 `readdir` 操作。其主要操作包括：

- **参数校验与持有Vnode**：确保输入的Vnode有效，然后持有它以防止被释放。
- **初始化目录遍历指针**：设置当前目录条目的遍历指针 `pdirCurrent` 为第一个子目录，并确保文件操作结构 `pf` 有效。
- **设置读取位置**：将文件操作结构中的读取位置(`fPos`)初始化为0，准备开始读取。

- **释放Vnode**：完成初始化后，释放Vnode的持有。