

天津大學



OSKernel2024-X1

操作系统设计与分析文档

RuanaOS

学校：天津大学

作者：李福鹏

时间：2024 年 7 月 29 日

目录

第一章 文件系统.....	6
1.1 ext4 文件系统概述.....	6
1.2 ext4 文件系统介绍.....	7
1.2.1 ext4 磁盘布局	7
1.2.2 superblock 超级块.....	9
1.2.3 Group descriptors	15
1.2.4 Block bitmap 块位图.....	17
1.2.5 Inode bitmap 索引节点位图	19

1.2.6 Inode table 索引节点表	20
1.2.6 根目录.....	22
1.2.7 目录项.....	22
1.2.8 寻找文件.....	23
1.2.9 特殊文件.....	24
1.2.10 exec 读取 elf 文件	24
1.2.11 创建文件.....	25
1.2.12 删除文件.....	25
1.2.13 文件系统相关系统调用	25
第二章 内存管理.....	26
2.1 内存相关异常处理	26
2.1.1 13、15 号异常.....	26
2.1.2 7 号异常	29
2.2 free 内存	29
2.3 用户空间管理	30
2.4 内存管理相关系统调用	32
第三章 进程管理.....	34
3.1 父子进程之间共享内存	34
3.2 线程.....	34
第四章 信号机制.....	36
4.1 信号处理机制概述	36
4.2 信号蹦床 (Trampoline) 实现.....	36
4.3 信号处理函数的执行	37
4.4 信号嵌套处理	37
4.5 同种信号的不可重入性	37
第五章 遇到的问题以及解决方法.....	38
5.1 读取磁盘镜像	38
5.2 exec 改进	38

5.3 elf 文件读取	38
5.4 wait 相关接口	39
5.5 getdents64 系统调用修改	39
5.6 对于浮点数的支持	39
第六章 第二阶段任务	40
6.1 net 操作	40
6.2 文件系统继续完善	41
6.3 文件结构整理	41

本项目借鉴了以下第三方库：

1. ****xv6-riscv****

- 链接：<https://github.com/mit-pdos/xv6-riscv>

2. ****oskernel2023-avx****

- 用途：优化文件 IO 操作
- 链接：

<https://gitlab.eduxiji.net/202310487101114/oskernel2023-avx>

3. **循序渐进，学习开发一个 RISC-V 上的操作系统 - 汪辰 -

2021 春**

- 链接：

https://www.bilibili.com/video/BV1Q5411w7z5/?seid=13099150746000866207&vd_source=e9be3150af06b69873c1f18bef15b2af

第一章 文件系统

1.1 ext4 文件系统概述

Ext4 (Fourth Extended File System) , 即第四代扩展文件系统 , 是 Linux 系统下的日志文件系统 , 是 ext2 和 ext3 文件系统的后继版本。Ext4 由 Ext3 的维护者 Theodore Tso 领导的开发团队实现 , 并引入到 Linux 2.6.19 内核中。Ext4 在 Ext3 的基础上进行了大量改进和增强 , 以提高文件系统的性能和可靠性。

1.2 ext4 文件系统介绍

1.2.1 ext4 磁盘布局

直接从我们评测的 ext4 磁盘镜像分析，一个 2GB 大小的空间，ext4 文件系统将它分隔成了 0~15 的 16 个 Group。

ext4 的总体磁盘布局如下：

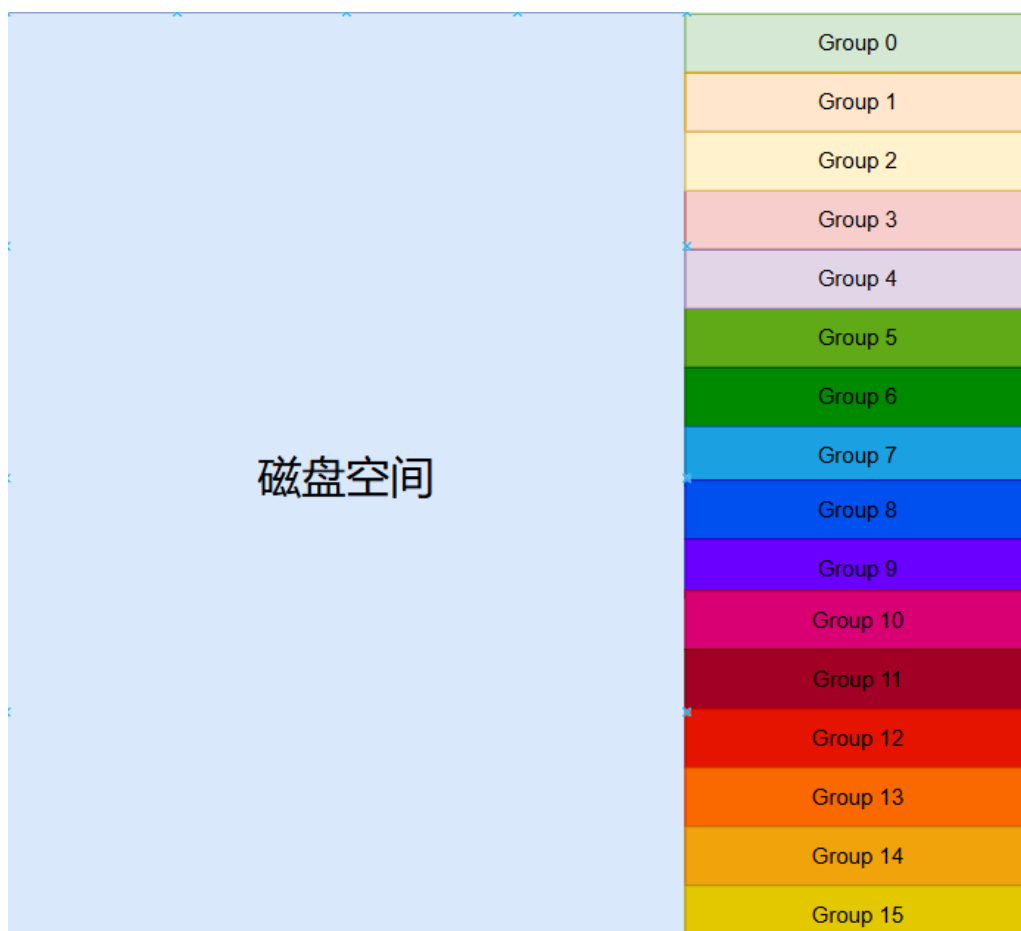


图 1.1 ext4 总体布局

其中 每个Group中又有superblock、Group descriptors、bitmap、Inode table、usrer data、还有一些保留空间，细分之后的空间布局如下：



图 1.2 ext4 group 布局

从图示中可以明确观察到几个关键点：

1.Backup Superblock、Group Descriptors、Reserved GDT 的特定分布 :这些关键的系统元数据信息被精心分配在文件系统的特定块组中，具体是 1、3、5、7 号块组，而 2、4、6 号块组则不包含这些关键数据，体现了数据备份与恢复的冗余设计策略。

2.Block Bitmap、Inode Bitmap、Inode Table 的差异化布局：在每一个块组内，Block Bitmap（块位图）、Inode Bitmap（索引节点位图）以及 Inode Table（索引节点表）等关键元文件的物理位置并非固定不变，而是相对于各自块组的起始位置，呈现出一种有规律的偏移，具体表现为相邻块组间的这些元文件位置相差一个 Block 的空间（尽管图示中因显示比例限制未直观展现此细节）。这种设计优化了文件系统的数据布局，提高了访问效率。

此外，值得注意的是，每个包含 superblock 的块组（即 1、3、5、7 号块组）中的 superblock 内容是完全一致的，它们实际上是文件系统主 superblock（位于 group 0）的冗余备份。这种设计增强了文件系统的健壮性，即使主 superblock 因故损坏，系统也能通过这些备份 superblock 快速恢复关键的系统元数据信息，从而保障数据的完整性和可访问性。

1.2.2 superblock 超级块

```
1.  0000000 0000 0000 0000 0000 0000 0000 0000 0000
2.  *
3.  0000400 0000 0002 0000 0008 6666 0000 cd90 0000
4.  0000410 f2f5 0001 0000 0000 0002 0000 0002 0000
5.  0000420 8000 0000 8000 0000 2000 0000 fdc2 66a4
6.  0000430 fdc2 66a4 001d ffff ef53 0001 0001 0000
7.  0000440 42cd 665d 0000 0000 0000 0000 0001 0000
8.  0000450 0000 0000 000b 0000 0100 0000 003c 0000
9.  0000460 02c6 0000 046b 0000 8432 e80b 2f52 9b45
10. 0000470 7489 ce05 39da b8ef 0000 0000 0000 0000
11. 0000480 0000 0000 0000 0000 6d2f 746e 652f 2e2f
12. 0000490 4141 4141 4141 4141 e641 94af b5e8 2f9b
13. 00004a0 6e6d 0074 0000 0000 0000 0000 0000 0000
14. 00004b0 0000 0000 0000 0000 0000 0000 0000 0000
```

15. 00004c0 0000 0000 0000 0000 0000 0000 00ff

16. 00004d0 0000 0000 0000 0000 0000 0000 0000

17. 00004e0 0008 0000 0000 0000 0000 0000 3473 70a8

18. 00004f0 140a dc41 f5a0 c5e8 03b2 2702 0101 0040

19. 0000500 000c 0000 0000 0000 42cd 665d f30a 0001

20. 0000510 0004 0000 0000 0000 0000 0000 4000 0000

21. 0000520 0000 0004 0000 0000 0000 0000 0000 0000

22. 0000530 0000 0000 0000 0000 0000 0000 0000 0000

23. 0000540 0000 0000 0000 0000 0000 0000 0000 0400

24. 0000550 0000 0000 0000 0000 0000 0000 0020 0020

25. 0000560 0001 0000 0000 0000 0000 0000 0000 0000

26. 0000570 0000 0000 0104 0000 4601 001b 0000 0000

27. 0000580 0000 0000 0000 0000 0000 0000 0000 0000

28. *

29. 0000640 0000 0000 0000 0000 6626 0000 0000 0000

30. 0000650 0000 0000 0000 0000 0000 0000 0000 0000

31. *

32. 00007f0 0000 0000 0000 0000 0000 0000 dbca 1ecf

33. 0000800 0000 0000 0000 0000 0000 0000 0000 0000

34. *

35. 0001000

对超级块的部分数据进行解析 ,我们设计了 ext4 superblock 的相关

数据结构：

```
struct ext4_sblock {
    //char   pad[1024];
    /* 0 */ uint32_t inodes_count;      /* I-nodes count */
    /* 4 */ uint32_t blocks_count_lo;   /* Blocks count */
    /* 8 */ uint32_t reserved_blocks_count_lo; /* Reserved blocks count */
    /* c */ uint32_t free_blocks_count_lo; /* Free blocks count */
    /*10 */ uint32_t free_inodes_count; /* Free inodes count */
    /*14 */ uint32_t first_data_block; /* First Data Block */
    /*18 */ uint32_t log_block_size;    /* Block size */
    /*1c */ uint32_t log_cluster_size; /* Obsoleted fragment size */
    /*20 */ uint32_t blocks_per_group; /* Number of blocks per group */
    /*24 */ uint32_t frags_per_group; /* Obsoleted fragments per group */
    /*28 */ uint32_t inodes_per_group; /* Number of inodes per group */
    /*2c */ uint32_t mount_time;        /* Mount time */
    /*30 */ uint32_t write_time;        /* Write time */
    /* 0 */ uint16_t mount_count;       /* Mount count */
    /* 0 */ uint16_t max_mount_count;   /* Maximal mount count */
    /* 0 */ uint16_t magic;             /* Magic signature */
    /* 0 */ uint16_t state;             /* File system state */
    /* 0 */ uint16_t errors;            /* Behavior when detecting errors */
    /* 0 */ uint16_t minor_rev_level;   /* Minor revision level */
    /* 0 */ uint32_t last_check_time;   /* Time of last check */
    /* 0 */ uint32_t check_interval;    /* Maximum time between checks */
    /* 0 */ uint32_t creator_os;        /* Creator OS */
    /* 0 */ uint32_t rev_level;         /* Revision level */
    /* 0 */ uint16_t def_resuid;        /* Default uid for reserved blocks */
    /* 0 */ uint16_t def_resgid;        /* Default gid for reserved blocks */

    /* Fields for EXT4_DYNAMIC_REV superblocks only. */
    /* 0 */ uint32_t first_inode; /* First non-reserved inode */
    /* 0 */ uint16_t inode_size; /* Size of inode structure */
    /* 0 */ uint16_t block_group_index; /* Block group index of this superblock */
    /* 0 */ uint32_t features_compatible; /* Compatible feature set */
    /* 0 */ uint32_t features_incompatible; /* Incompatible feature set */
    /* 0 */ uint32_t features_read_only; /* Readonly-compatible feature set */
    /* 0 */ uint8_t uuid[UUID_SIZE]; /* 128-bit uuid for volume */
    /* 0 */ char volume_name[16]; /* Volume name */
    /* 0 */ char last_mounted[64]; /* Directory where last mounted */
    /* 0 */ uint32_t algorithm_usage_bitmap; /* For compression */

    /*
```

```

    * Performance hints. Directory preallocation should only
    * happen if the EXT4_FEATURE_COMPAT_DIR_PREALLOC flag is on.
    */

/* 0 */ uint8_t s_prealloc_blocks; /* Number of blocks to try to preallocate */
/* 0 */ uint8_t s_prealloc_dir_blocks; /* Number to preallocate for dirs */
/* 0 */ uint16_t s_reserved_gdt_blocks; /* Per group desc for online growth */

/*
 * Journaling support valid if EXT4_FEATURE_COMPAT_HAS_JOURNAL set.
 */
/* 0 */ uint8_t journal_uuid[UUID_SIZE]; /* UUID of journal superblock */
/* 0 */ uint32_t journal_inode_number; /* Inode number of journal file */
/* 0 */ uint32_t journal_dev; /* Device number of journal file */
/* 0 */ uint32_t last_orphan; /* Head of list of inodes to delete */
/* 0 */ uint32_t hash_seed[4]; /* HTREE hash seed */
/* 0 */ uint8_t default_hash_version; /* Default hash version to use */
/* 0 */ uint8_t journal_backup_type;
/* 0 */ uint16_t desc_size; /* Size of group descriptor */
/* 0 */ uint32_t default_mount_opts; /* Default mount options */
/* 0 */ uint32_t first_meta_bg; /* First metablock block group */
/* 0 */ uint32_t mkfs_time; /* When the filesystem was created */
/* 0 */ uint32_t journal_blocks[17]; /* Backup of the journal inode */

/* 64bit support valid if EXT4_FEATURE_COMPAT_64BIT */
/* 0 */ uint32_t blocks_count_hi; /* Blocks count */
/* 0 */ uint32_t reserved_blocks_count_hi; /* Reserved blocks count */
/* 0 */ uint32_t free_blocks_count_hi; /* Free blocks count */
/* 0 */ uint16_t min_extra_isize; /* All inodes have at least # bytes */
/* 0 */ uint16_t want_extra_isize; /* New inodes should reserve # bytes */
/* 0 */ uint32_t flags; /* Miscellaneous flags */
/* 0 */ uint16_t raid_stride; /* RAID stride */
/* 0 */ uint16_t mmp_interval; /* # seconds to wait in MMP checking */
/* 0 */ uint64_t mmp_block; /* Block for multi-mount protection */
/* 0 */ uint32_t raid_stripe_width; /* Blocks on all data disks (N * stride) */
/* 0 */ uint8_t log_groups_per_flex; /* FLEX_BG group size */
/* 0 */ uint8_t checksum_type;
/* 0 */ uint16_t reserved_pad;
/* 0 */ uint64_t kbytes_written; /* Number of lifetime kilobytes written */
/* 0 */ uint32_t snapshot_inum; /* I-node number of active snapshot */
/* 0 */ uint32_t snapshot_id; /* Sequential ID of active snapshot */
/* 0 */ uint64_t snapshot_r_blocks_count; /* Reserved blocks for active snapshot's future use */
/* 0 */ uint32_t snapshot_list; /* I-node number of the head of the on-disk snapshot list */

```

```

/* 0 */ uint32_t error_count; /* Number of file system errors */
/* 0 */ uint32_t first_error_time; /* First time an error happened */
/* 0 */ uint32_t first_error_ino; /* I-node involved in first error */
/* 0 */ uint64_t first_error_block; /* Block involved of first error */
/* 0 */ uint8_t first_error_func[32]; /* Function where the error happened */
/* 0 */ uint32_t first_error_line; /* Line number where error happened */
/* 0 */ uint32_t last_error_time; /* Most recent time of an error */
/* 0 */ uint32_t last_error_ino; /* I-node involved in last error */
/* 0 */ uint32_t last_error_line; /* Line number where error happened */
/* 0 */ uint64_t last_error_block; /* Block involved of last error */
/* 0 */ uint8_t last_error_func[32]; /* Function where the error happened */
/* 0 */ uint8_t mount_opts[64];
/* 0 */ uint32_t usr_quota_inum; /* inode for tracking user quota */
/* 0 */ uint32_t grp_quota_inum; /* inode for tracking group quota */
/* 0 */ uint32_t overhead_clusters; /* overhead blocks/clusters in fs */
/* 0 */ uint32_t backup_bgs[2]; /* groups with sparse_super2 SBs */
/* 0 */ uint8_t encrypt_algos[4]; /* Encryption algorithms in use */
/* 0 */ uint8_t encrypt_pw_salt[16]; /* Salt used for string2key algorithm */
/* 0 */ uint32_t lpf_ino; /* Location of the lost+found inode */
/* 0 */ uint32_t padding[100]; /* Padding to the end of the block */
/* 0 */ uint32_t checksum; /* crc32c(superblock) */

```

我们选择上面比较重要的字段进行解释：

- **inodes_count** (uint32_t): 文件中节点(inodes)的总数。inodes 是 Linux 文件系统中用于存储文件元数据（如权限、所有者、大小、时间戳等）的数据结构。
- **blocks_count_lo** (uint32_t): 文件系统中块的总数(低 32 位)。ext4 文件系统使用块来存储数据。
- **reserved_blocks_count_lo** (uint32_t): 保留块的数量（低 32 位）。这些块是为系统或特定用途预留的，普通用户无法访问。
- **free_blocks_count_lo** (uint32_t): 空闲块的数量（低 32 位）。
- **free_inodes_count** (uint32_t): 空闲 inodes 的数量。

- **first_data_block** (uint32_t): 第一个数据块的编号。在 ext4 中，超级块和组描述符等信息占据了文件系统的前几个块，first_data_block 指出了第一个数据块的起始位置。
- **log_block_size** (uint32_t): 块大小的对数值(以 1024 字节为单位)。例如，如果 log_block_size 为 10，则块大小为 $1024 * 1024 = 1\text{MB}$ 。
- **log_cluster_size** (uint32_t): 已废弃的碎片大小的对数值。在 ext4 中，这个字段已经不再使用。
- **blocks_per_group** (uint32_t): 每个块组中的块数。ext4 文件系统将磁盘划分为多个块组，以提高大文件系统的效率。
- **frags_per_group** (uint32_t): 已废弃的每个块组中碎片的数量。
- **inodes_per_group** (uint32_t): 每个块组中 inodes 的数量。
- **first_inode** (uint32_t): 第一个非保留 inode 的编号。在 ext4 中，一些特定的 inode 被保留用于特殊目的，如根目录、节点表等。
- **inode_size** (uint16_t): inode 结构的大小。这个大小可以根据文件系统的配置而变化（在我们此处是 256 字节）。
- **block_group_index** (uint16_t): 当前超级块所属的块组索引。
- **features_compatible** (uint32_t): 兼容特性集。这个字段用于指示文件系统支持哪些与兼容性相关的特性。

1.2.3 Group descriptors

我们设计的 group descriptors 结构体如下：

```
struct ext4_bgroup {
    uint32_t block_bitmap_lo;    /* Blocks bitmap block */
    uint32_t inode_bitmap_lo;    /* Inodes bitmap block */
    uint32_t inode_table_first_block_lo; /* Inodes table block */
    uint16_t free_blocks_count_lo; /* Free blocks count */
    uint16_t free_inodes_count_lo; /* Free inodes count */
    uint16_t used_dirs_count_lo; /* Directories count */
    uint16_t flags;              /* EXT4_BG_flags (INODE_UNINIT, etc) */
    uint32_t exclude_bitmap_lo; /* Exclude bitmap for snapshots */
    uint16_t block_bitmap_csum_lo; /* crc32c(s_uuid+grp_num+bbitmap) LE */
    uint16_t inode_bitmap_csum_lo; /* crc32c(s_uuid+grp_num+ibitmap) LE */
    uint16_t itable_unused_lo; /* Unused inodes count */
    uint16_t checksum;          /* crc16(sb_uuid+group+desc) */

    uint32_t block_bitmap_hi;    /* Blocks bitmap block MSB */
    uint32_t inode_bitmap_hi;    /* I-nodes bitmap block MSB */
    uint32_t inode_table_first_block_hi; /* I-nodes table block MSB */
    uint16_t free_blocks_count_hi; /* Free blocks count MSB */
    uint16_t free_inodes_count_hi; /* Free i-nodes count MSB */
    uint16_t used_dirs_count_hi; /* Directories count MSB */
    uint16_t itable_unused_hi; /* Unused inodes count MSB */
    uint32_t exclude_bitmap_hi; /* Exclude bitmap block MSB */
    uint16_t block_bitmap_csum_hi; /* crc32c(s_uuid+grp_num+bbitmap) BE */
    uint16_t inode_bitmap_csum_hi; /* crc32c(s_uuid+grp_num+ibitmap) BE */
    uint32_t reserved;          /* Padding */
};
```

上面字段的解释如下：

- **block_bitmap_lo**: 块位图的低 32 位地址。块位图是一个位数组，用于跟踪块组中哪些块是已使用的，哪些块是空闲的。这个字段表示块位图在文件系统中的起始块号（低 32 位）。
- **inode_bitmap_lo**: inode 位图的低 32 位地址。inode 位图与块位图类似，但它跟踪的是 inodes 的使用情况。这个字段表示 inode 位图在文件系统中的起始块号（低 32 位）。

- **inode_table_first_block_lo**: inode 表首块的低 32 位地址。
inode 表包含了块组中所有 inode 的详细信息。这个字段表示 inode 表在文件系统中的起始块号（低 32 位）。
- **free_blocks_count_lo**: 空闲块数的低 16 位。这个字段记录了块组中当前空闲的块数（低 16 位）。
- **free_inodes_count_lo**: 空闲 inode 数的低 16 位。这个字段记录了块组中当前空闲的 inode 数（低 16 位）。
- **used_dirs_count_lo**: 已使用目录数的低 16 位。这个字段可能用于跟踪块组中已使用的目录数量（低 16 位），但请注意，这个字段的确切用途可能因文件系统版本和实现而异。
- **flags**: 块组标志。这个字段包含了一系列标志位，用于指示块组的特定属性或状态，如 inode 是否未初始化等。
- **exclude_bitmap_lo**: 排除位图的低 32 位地址。在支持快照功能的 ext4 文件系统中，排除位图用于标识哪些块在快照过程中应该被排除在外。
- **block_bitmap_csum_lo** 和 **inode_bitmap_csum_lo**: 块位图和 inode 位图的 CRC32 校验和的低 16 位。这些字段用于验证块位图和 inode 位图的完整性。
- **itable_unused_lo**: 未使用 inode 数的低 16 位。这个字段可能记录了块组中未使用的 inode 数量（低 16 位），但这取决于文件系统的具体实现。

- **checksum**: CRC16 校验和。这个字段是整个块组描述符的 CRC16 校验和，用于验证描述符的完整性。
- 接下来的字段（带有_hi 后缀）是上述字段的高 32 位或高 16 位部分，用于支持大于 4GB 的文件系统，其中块号和 inode 号可能超过 32 位的范围。
- **block_bitmap_hi, inode_bitmap_hi, inode_table_first_block_hi**: 分别是块位图、inode 位图和 inode 表首块的高 32 位地址。
- **free_blocks_count_hi, free_inodes_count_hi, used_dirs_count_hi, itable_unused_hi**: 分别是空闲块数、空闲 inode 数、已使用目录数和未使用 inode 数的高 16 位。
- **exclude_bitmap_hi**: 排除位图的高 32 位地址。
- **block_bitmap_csum_hi** 和 **inode_bitmap_csum_hi**: 块位图和 inode 位图的 CRC32 校验和的高 16 位(注意这里通常不会用到，因为 CRC32 是 32 位的，但可能是为了与低 16 位对齐或保留未来扩展)。
- **reserved**: 保留字段，用于将来的扩展或对齐。

1.2.4 Block bitmap 块位图

Block Bitmap 是一种在块组（Block Group）层面上用于高效管理数据块使用状态的数据结构。它通过位图的形式，精确地记录了块组

中每一个数据块是处于已使用状态还是空闲状态。在这种机制中，每个位（bit）都对应块组中的一个数据块，位值为 1 时表示该数据块已被占用（即已使用），位值为 0 则表示该数据块当前空闲，可供分配。

当我们在 ext4(或类似支持块位图管理的文件系统) 上创建文件时，文件系统会执行以下操作来利用 Block Bitmap：

- **查找空闲块**：首先，文件系统会查看 Block Bitmap，寻找未被使用的数据块（即位值为 0 的位）。这一步骤可能涉及遍历整个位图或利用更高效的算法来快速定位空闲块。
- **分配块**：一旦找到足够的空闲块，文件系统会将这些块分配给新创建的文件，并在 Block Bitmap 中将对应位设置为 1，以标记这些块为已使用状态。
- **更新文件元数据**：除了修改 Block Bitmap 外，文件系统还会更新文件的 inode（索引节点），记录文件的块分配信息，如文件占用了哪些数据块。
- 相反，在删除文件时，文件系统会执行相反的操作来释放资源：
- **回收块**：首先，文件系统会从文件的 inode 中读取块分配信息，确定哪些数据块被该文件占用。
- **更新 Block Bitmap**：然后，文件系统会在 Block Bitmap 中将这些数据块对应的位从 1 重置为 0，标记它们为空闲状态，以便后续的文件操作可以重新使用这些块。

- **可选的清理操作**：在某些情况下，为了优化文件系统的性能或空间使用效率，删除文件后可能还会执行额外的清理或合并操作，如回收连续空闲块以减少碎片。
- 通过这些步骤，Block Bitmap 不仅确保了文件系统中数据块的有效管理，还支持了文件的高效创建和删除操作。

1.2.5 Inode bitmap 索引节点位图

与 Block Bitmap 的工作原理相类似，Inode Bitmap 在 ext4 文件系统中扮演着管理块组中 inode 使用状态的关键角色。Inode Bitmap 是一个位图，其中每一位都对应块组中的一个 inode，用于明确地标识哪些 inode 当前已被分配并正在使用中，以及哪些 inode 仍处于空闲状态，可供新文件或目录的创建所使用。

在创建文件或目录时，文件系统需要为这些新实体分配一个或多个 inode。此时，文件系统会首先检查 Inode Bitmap，寻找一个或多个标记为空闲的 inode 位。一旦找到空闲的 inode，文件系统就会将这些位从空闲状态更改为已使用状态，并将相应的 inode 数据结构初始化为新文件或目录的属性。这样，新创建的文件或目录就有了自己的 inode 编号，并可以存储到文件系统的 inode 表中。

相反，在删除文件或目录时，文件系统会执行相反的操作。它会首先标记与文件或目录相关联的 inode 为空闲状态，在 Inode Bitmap

中将相应的位从已使用状态更改为空闲状态。这样，这些 inode 就重新变得可用，可以在未来的文件或目录创建过程中被重新分配。

综上所述，Inode Bitmap 是 ext4 文件系统中一个至关重要的组成部分，它有效地管理着块组中 inode 的分配和回收，确保了文件系统的稳定性和效率。在文件或目录的创建和删除过程中，对 Inode Bitmap 的更新操作是必不可少的步骤。

1.2.6 Inode table 索引节点表

对于 ext4 的 inode，他们在索引节点表 (inode table) 中的大小都是唯一的，根据系统而定，在此处为 256 字节，我们根据每个字节的含义设计了 inode (物理) 数据结构：

```
struct ext4_inode {
    uint16_t mode;           /* File mode */
    uint16_t uid;           /* Low 16 bits of owner uid */
    uint32_t size_lo;       /* Size in bytes */
    uint32_t access_time;   /* Access time */
    uint32_t change_inode_time; /* I-node change time */
    uint32_t modification_time; /* Modification time */
    uint32_t deletion_time; /* Deletion time */
    uint16_t gid;           /* Low 16 bits of group id */
    uint16_t links_count;   /* Links count */
    uint32_t blocks_count_lo; /* Blocks count */
    uint32_t flags;         /* File flags */
    uint32_t unused_osd1;    /* OS dependent - not used in HelenOS */
    struct i_block i_block; /* Pointers to blocks */
    uint32_t generation;     /* File version (for NFS) */
    uint32_t file_acl_lo;    /* File ACL */
    uint32_t size_hi;
    uint32_t obso_faddr; /* Obsoleted fragment address */

    union {
        struct {
            uint16_t blocks_high;

```

```

        uint16_t file_acl_high;
        uint16_t uid_high;
        uint16_t gid_high;
        uint16_t checksum_lo; /* crc32c(uuid+inum+inode) LE */
        uint16_t reserved2;
    } linux2;
    struct {
        uint16_t reserved1;
        uint16_t mode_high;
        uint16_t uid_high;
        uint16_t gid_high;
        uint32_t author;
    } hurd2;
} osd2;

uint16_t extra_isize;
uint16_t checksum_hi; /* crc32c(uuid+inum+inode) BE */
uint32_t ctime_extra; /* Extra change time (nsec << 2 | epoch) */
uint32_t mtime_extra; /* Extra Modification time (nsec << 2 | epoch) */
uint32_t atime_extra; /* Extra Access time (nsec << 2 | epoch) */
uint32_t crtime; /* File creation time */
uint32_t crtime_extra; /* Extra file creation time (nsec << 2 | epoch) */
uint32_t version_hi; /* High 32 bits for 64-bit version */
};

```

其余的字段比较好理解，但是其中比较重要的是 `i_block` 字段，这指向了文件所在的 user data 位置。对于不同大小的文件，`i_block` 的含义也不同，因此我们设计了 `i_block` 结构：

```

struct ext4_extent{
    uint32_t ee_block;
    uint16_t ee_len;
    uint16_t ee_start_hi;
    uint32_t ee_start_lo;
};

struct ext4_extent_idx{
    uint32_t ei_block;
    uint32_t ei_leaf_lo;
    uint16_t ei_leaf_hi;
    uint16_t unused;
};

```

```

struct i_block {
    struct {
        uint16_t magic;
        uint16_t entries;
        uint16_t max;
        uint16_t depth;
        uint32_t generation;
    } extent_header;
    union {
        struct ext4_extent ext4_extent[4];
        struct ext4_extent_idx ext4_extent_idx[4];
    };
};

```

其中一个是指向 user data 所在位置，另一个是通过树的结构，来指向文件的各个块（block，大小为 4096）。

但是在我们评测 sdcard.img 中，都是第一种方式，因此文件 data 也是连续存储在磁盘上的。

通过 `ext4_extent` 的 `ee_start_hi` 和 `ee_start_lo` 我们可以定位到 user data block 中。

1.2.6 根目录

在 ext4 中，0~11 的 inode 号中，只有 2 是根目录所用，其余都是保留。但是根目录在 inode table 中的索引为 1，因此其余的 inode 的索引都是 inode 号减一。

1.2.7 目录项

在 ext4 中，目录项的字节数并不像 fat32 一样固定，这也显得操作人不需要耗费精力在长短目录项上，但是读取目录项是不固定的，需要根据文件的名称来确定长度，而且需要在文件名字后加一个终结符。

因此我们设计了 ext4 目录项结构：

```
struct dirent {  
    uint32_t inum;  
    uint16_t len;  
    uint8_t name_len;  
    uint8_t file_type;  
    char name[0x20];  
};
```

这里为了方便，我们将文件名字长度设置为了 32 字节，但是在读取目录项时，同样要将 name_len 处字节设置为 '\0'，这是因为我们反复地删除和创建文件，可能会有字节超出了我们的预料，为了安全，我们选择这么做。

Ext4 目录项还有一个特点，就是最后一个目录项，他的 len 字段的大小不等于 $4+2+1+1+name_len$ ，而是等于从目录项开始到 block 结束的字节数。

1.2.8 寻找文件

寻找目录和文件或者是设备文件的方法都是一样的，根据 linux 的标准，文件路径如果是 '/' 开头，则从根目录开始查找，否则从进程的“当前目录”进行查找。

我们根据父目录的 inode 号找到父目录的 data block，然后逐个读取目录项：如果遇到上面所述 len 字段比 $8+name_len$ （8 是除了 name 字段的字节数）大，或者是 len 为 0，则遇到了目录的最后一个目录项，寻找文件结束，未找到文件；否则读取目录项信息，获得 inode 号，寻址到 data block，读取目标文件内容。

1.2.9 特殊文件

对于/proc 或者/syslog 这种文件,都是不用存在在 disk 上的特殊文件, 我们需要为他们在内核中创建信息。

1.2.10 exec 读取 elf 文件

在处理评测点时,我们遇到了一些问题:elf 文件虚拟地址不与 block size (4096) 对齐, 我们使用 readelf 发现 elf 文件多出了很多其他段, 于是我们要修改读取 elf 文件的逻辑:

```
if((sz1 = uvmmalloc(pagetable, sz, ph.vaddr + v_off - 1, flags2perm(ph.flags))) == 0) {
    printf("uvmmalloc failed\n");
    goto bad;
}

//printf("exec: vaddr: %p %p\n", sz, ph.vaddr);
if(loadseg(pagetable, ph.vaddr, ip, ph.off, v_off) < 0) {
    printf("loadseg failed\n");
    goto bad;
}

//printf("v_off: %d\n", v_off);
sz = sz1;

ph.vaddr = ph.vaddr + v_off;
if((sz1 = uvmmalloc(pagetable, sz, ph.vaddr + ph.memsz - v_off, flags2perm(ph.flags))) == 0)
{
    printf("uvmmalloc failed\n");
    goto bad;
}
sz = sz1;
//printf("ph.vaddr: %p %p %p\n", ph.vaddr, ph.vaddr + ph.memsz - v_off, sz);
if(loadseg(pagetable, ph.vaddr, ip, ph.off + v_off, ph.filesz - v_off) < 0) {
    printf("loadseg failed\n");
    goto bad;
}
} else {
    /**/
}
```



```

//printf("sz: %d %p %p %p %p\n", sz, ph.vaddr, ph.memsz, ph.filesz, ph.off);
if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flags))) == 0) {
    printf("uvmalloc failed\n");
    goto bad;
}
//printf("exec: %p\n", ph.vaddr + ph.memsz);
sz = sz1;
if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0) {
    printf("loadseg error\n");
    goto bad;
}
}
}

```

这时 ,我们就不能按照 pghdr 的偏移是否对齐来判断 elf 文件是否合法 ,因此 ,loadseg 也需要稍微调整 ,只是需要迎合未对齐部分即可。但是由于我们是分段 loadseg ,先读取未对齐部分 ,loadseg 如果 load 的 size 等于想要 load 的 size 就直接 return ,否则会出现 bug。

1.2.11 创建文件

创建文件需要在父目录的 data 中创建此文件的目录项 然后再 block map 和 inode map 中申请一位 ,这就是申请了 block 和 inode ,接着在 inode table 中填入相关信息 ,如果目标文件是目录的话 ,我们需要在文件的 data 中首先添加 “.” 和 “..” 目录项。

1.2.12 删除文件

删除文件与创建文件的操作正好相反 ,但是在接下来 ,我们要优化相关测操作 ,做到节省磁盘空间。

1.2.13 文件系统相关系统调用

第二章 内存管理

2.1 内存相关异常处理

2.1.1 13、15 号异常

在 riscv 中 ,13 号异常 load 异常 ,15 号是 store 异常 ,如下图所示 :

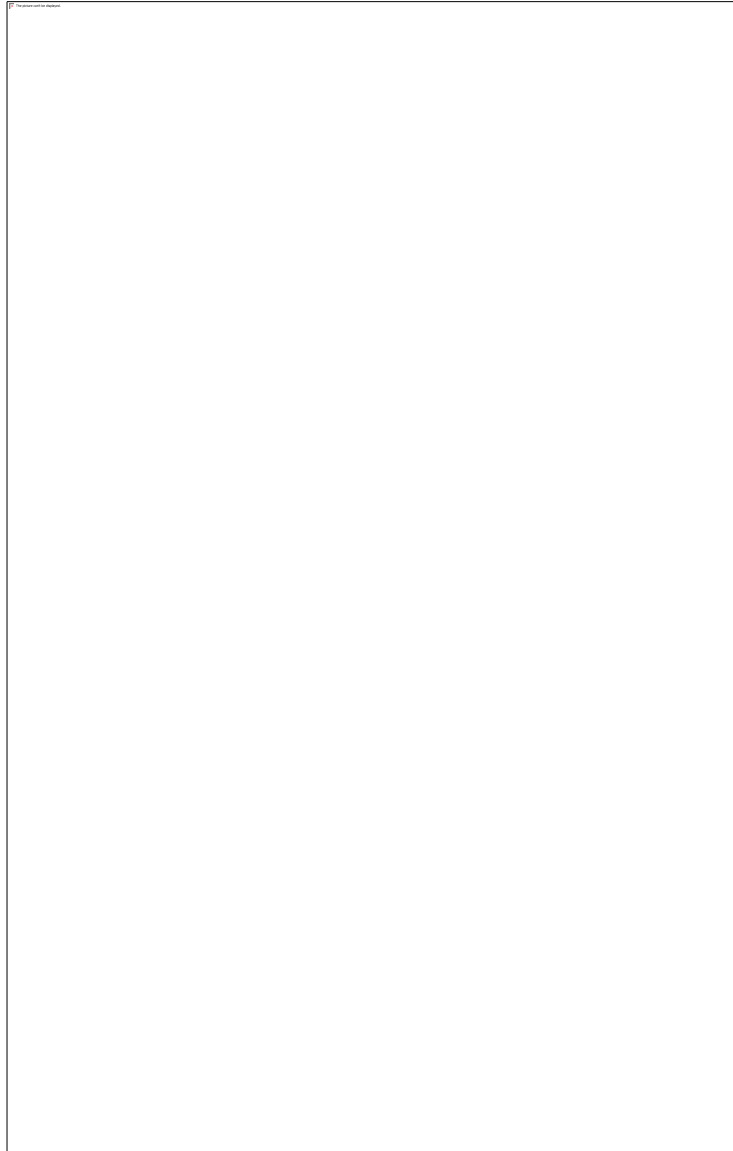


图 2.1 riscv Exception Code

因此，我们需要为缺页的虚拟地址 map 一个物理内存。对于缺页处理，我们有两种处理方式，一种是只处理缺页的虚拟内存，另一种是从当前进程的 size 处开始处理，一直到处理完缺页的位置即可。

另外，缺页的另一个原因是页面已经存在，但是权限不对，因此，对于小于当前内存 size 的虚拟地址，我们需要修改权限：

```
if(va < PGROUNDUP(myproc()->sz) || va > MAXUVA) {  
    pte_t *pte = walk(myproc()->pagetable, va, 0);  
    if(pte == 0) {  
        char *mem = kalloc();
```

```

        //printf("pte==0: %p %d\n", va, myproc()->pid);
        if(mappages(myproc()->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64_t)mem,
PTE_W|PTE_R|PTE_U) != 0){
            printf("mappages failed\n");
            kfree(mem);
            p->killed = 1;
        }
    } else {
        *pte |= PTE_U | PTE_R | PTE_W;
    }
} else {
    // allocate one physical page
    if(!namecmp(myproc()->name, "iozone")) {
        char *mem = kalloc();
        if(mem == 0)
            panic("usertrap: no more physical mem!");
        // zero the physical page
        memset(mem, 0, PGSIZE);
        // add the mappings
        //printf("mappages: va: %p pa: %p\n", va, mem);
        if(mappages(myproc()->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64_t)mem,
PTE_W|PTE_R|PTE_U) != 0){
            printf("mappages failed\n");
            kfree(mem);
            p->killed = 1;
        }
        myproc()->sz = PGROUNDDOWN(va) + PGSIZE;
    } else {
        uint64_t a;
        for(a = p->sz; a <= va; a+=PGSIZE) {
            char *mem = kalloc();
            if(mem == 0)
                panic("usertrap: no more physical mem!");
            // zero the physical page
            memset(mem, 0, PGSIZE);
            // add the mappings
            //printf("mappages: va: %p pa: %p\n", va, mem);
            if(mappages(p->pagetable, a, PGSIZE, (uint64_t)mem, PTE_W|PTE_R|PTE_U) != 0){
                printf("mappages failed\n");
                kfree(mem);
                p->killed = 1;
            }
            myproc()->sz = PGROUNDDOWN(a) + PGSIZE;
        }
    }
}

```

```

    }
}

```

之所以出现两种缺页处理的方式，这是因为我们在处理 iotest 测试点的时候，测试点需要 fork 进程，由于进程 size 过大，所以出现了物理内存不够用的情况，因此我们只能 fork text 和 data 段（elf 文件中的），同理我们对 iotest 进程只处理出现异常的地址。

2.1.2 7 号异常

由上图可以看出，7 号异常是权限出现错误，因此我们需要为页表项重新添加权限：

```

uint64_t va = r_stval();
pte_t *pte;
pte = walk(myproc()->pagetable, va, 0);
char *mem = kalloc();
*pte |= PTE_U | PTE_R | PTE_W | PA2PTE(mem);
if(myproc()->sz < PGROUNDDOWN(va + PGSIZE)){
    myproc()->sz = PGROUNDDOWN(va + PGSIZE);
}

```

2.2 free 内存

对于 fork 出来的进程，父进程通常会使用 wait 或者 wait64 来等待子进程（线程）退出，因此，父进程通常会释放子进程（线程）的资源，包括内存，但是我们进程可以是不连续的（如果有缺页异常的话，并且选择 lazy 处理），因此我们对于未与物理内存 map 的虚拟内存，我们选择“忽视”。

注：uvmunmap 是将虚拟内存与物理内存取消 map 的函数，此处重要的是 **continue**。

```

void uvmunmap(pagetable_t pagetable, uint64_t va, uint64_t npages, int do_free){

```

```

uint64_t a;
pte_t *pte;
if((va % PGSIZE) != 0){
    panic("uvmunmap : not aligned");
}
for(a = va; a < va + npages * PGSIZE; a += PGSIZE){
    if ((pte = walk(pagetable, a, 0)) == 0) {
        continue;
        panic("vmunmap: walk");
    }

    if ((*pte & PTE_V) == 0) {
        //printf("uvmunmap: %p %p %p\n", a, pte, *pte);
        continue;
    }

    if (PTE_FLAGS(*pte) == PTE_V)
        panic("vmunmap: not a leaf");
    if(do_free){
        uint64_t pa = PTE2PA(*pte);
        kfree((void*)pa);
    }
    *pte = 0;
}
}

```

2.3 用户空间管理

下图是我们初赛时用户空间的内存管理：

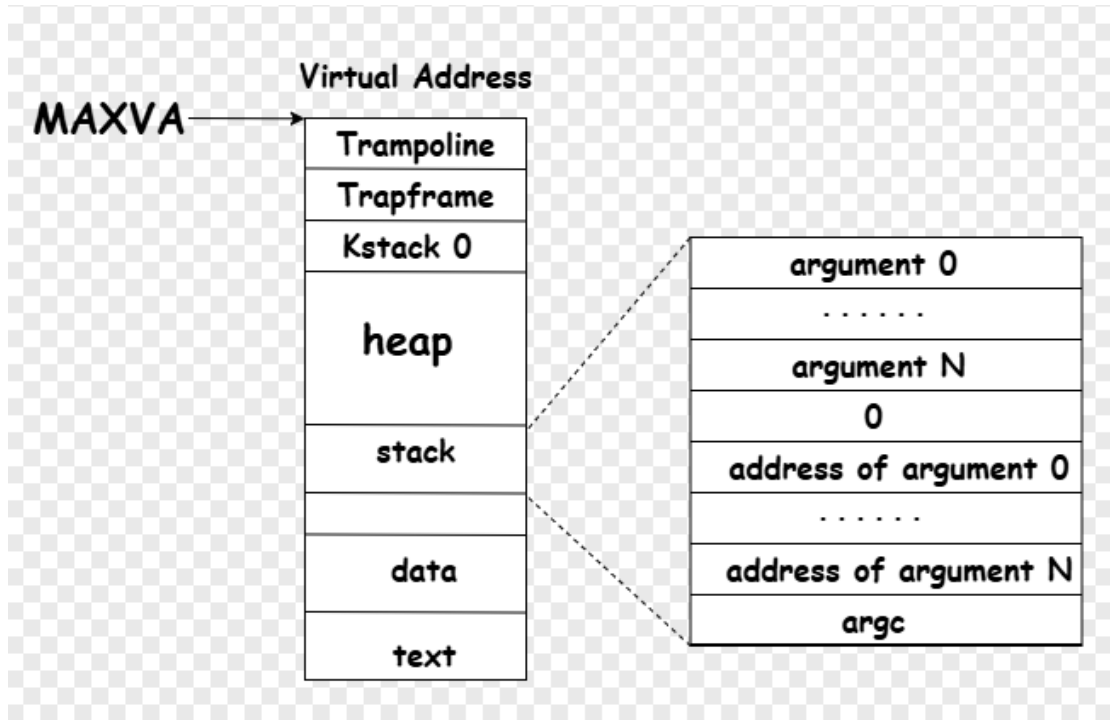


图 2.2 用户空间内存管理

但是在我们运行 `libc-bench` 测试点时，我们发现内核尝试修改 `text` 段，但是 `text` 并不可以写，内核却常常向此处写入数据，因此我们必须将 `stack` 再次提高，而不是 `data` 与 `stack` 只有一个 `pgsize` (4096 字节) 大小。

并且，我们要向栈中填入一些辅助信息：

```
uint64_t aux[MAXARG * 2 + 3] = {0, 0, 0};
alloc_aux(aux, AT_HWCAP, 0);
alloc_aux(aux, AT_PAGESZ, PGSIZE);
alloc_aux(aux, AT_PHDR, ph.vaddr);
alloc_aux(aux, AT_PHENT, elf.phentsize);
alloc_aux(aux, AT_PHNUM, elf.phnum);
alloc_aux(aux, AT_BASE, 0/*interp_start_addr*/);
alloc_aux(aux, AT_ENTRY, elf.entry);
alloc_aux(aux, AT_UID, 0);
alloc_aux(aux, AT_EUID, 0);
alloc_aux(aux, AT_GID, 0);
alloc_aux(aux, AT_EGID, 0);
alloc_aux(aux, AT_SECURE, 0);
alloc_aux(aux, AT_RANDOM, sp);
```

2.4 内存管理相关系统调用

第三章 进程管理

3.1 父子进程之间共享内存

对于 iotest 测试点，用到了 shmget、shmat 等等系统调用，这就需要父进程在 fork 子进程时，需要将共享内存区同样在子进程中映射到相同的物理内存区，这就需要我们修改 fork 的相关逻辑。

```
for(int id = 0; id < 4096; id++) {  
    if(shm.pid[id] == p->pid && p->pid != 0) {  
        //printf("fork: id: %d\n", id);  
        for(int i = 0; i < shm.shm_length[id]; i+=PGSIZE) {  
            //printf("fork: %p %p\n", shm.addr[id] + i, shm.pa[id][i / PGSIZE]);  
            mappages(np->pagetable, shm.addr[id] + i, PGSIZE, shm.pa[id][i / PGSIZE], PTE_R | PTE_U  
| PTE_V | PTE_W | PTE_X);  
        }  
    }  
}
```

3.2 线程

线程是通过 clone 系统调用，然后通过 thread_clone 来创建的，我们将线程视为“与父进程完全共享资源的子进程”，因此 thread_clone 逻辑与 fork 相似，只是将父进程的用户空间所映射的

物理地址完完全全映射到子进程的物理地址。

```
int thread_clone(uint64_t addr)
{
    int i, pid;
    struct proc *np;
    struct proc *p = myproc();

    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // Copy user memory from parent to child.
    if(uvmcopy1(p->pagetable, np->pagetable, p->sz) < 0){
        freeproc(np);
        release(&np->lock);
        return -1;
    }
    np->sz = p->sz;

    // copy saved user registers.
    *(np->trapframe) = *(p->trapframe);
    //printf("%0x\n", addr);
    np->trapframe->sp = addr;
    //np->trapframe->tp = addr;

    // Cause fork to return 0 in the child.
    np->trapframe->a0 = 0;
    //printf("%0x\n", p->trapframe->epc);
    // increment reference counts on open file descriptors.
    for(i = 0; i < NOFILE; i++)
        if(p->ofile[i])
            np->ofile[i] = filedup(p->ofile[i]);
    np->cwd = idup(p->cwd);

    safestrcpy(np->name, p->name, sizeof(p->name));

    pid = np->pid;

    release(&np->lock);

    acquire(&wait_lock);
    np->parent = p;
```

```
release(&wait_lock);  
  
acquire(&np->lock);  
np->state = RUNNABLE;  
release(&np->lock);  
return pid;  
}
```

第四章 信号机制

4.1 信号处理机制概述

在 Linux 系统中，信号是一种软件中断，用于通知进程某个事件的发生。当进程接收到信号时，它可以选择忽略该信号、捕获该信号并执行特定的信号处理函数，或者执行默认操作（如终止进程）。信号处理的核心在于如何在保证系统稳定性和安全性的前提下，高效且准确地传递和处理信号。

4.2 信号蹦床（Trampoline）实现

信号蹦床是一种特殊的代码段，其作用是作为信号处理函数执行完毕后的跳转点，确保控制流能够安全地返回到用户态。在您的示例中，蹦床段通过调用 `sigreturn` 系统调用来实现这一功能。不过，需要注意的是，`sigreturn` 已经逐渐被 `rt_sigreturn`（用于实时信号）所取代，因为后者提供了更好的支持，包括信号屏蔽字的恢复等。

```
asm
.section .signalTrampoline
    .globl signalTrampoline
    .align 12
signalTrampoline:
    # 假设使用 rt_sigreturn 系统调用号
    li a7, [RT_SIGRETURN_SYSCALL_NUMBER]
    ecall # 执行系统调用
```

4.3 信号处理函数的执行

当进程捕获到信号时，内核会暂停该进程的执行，并根据信号类型查找相应的信号处理函数。如果设置了信号处理函数，内核会保存当前的用户态上下文（包括寄存器状态、程序计数器等），然后切换到内核态执行信号处理函数。在信号处理函数执行前，内核会将返回地址设置为蹦床段的地址，确保处理完毕后能正确返回到用户态。

4.4 信号嵌套处理

信号嵌套处理指的是在信号处理函数执行过程中，进程可能再次接收到信号的情况。Linux 内核通过信号屏蔽字（signal mask）来管理信号的嵌套处理。当信号处理函数开始执行时，内核会自动将当前处理的信号添加到进程的信号屏蔽字中，防止该信号在处理过程中被再次处理。直到信号处理函数执行完毕并返回到用户态时，该信号才会从屏蔽字中移除，允许其再次被处理。

4.5 同种信号的不可重入性

同种信号的不可重入性指的是，在信号处理函数执行期间，如果进程

再次接收到同种信号，则默认情况下，该信号会被忽略，直到当前信号处理函数执行完毕。这通过修改信号的处理方式（如设置为 `SA_RESETHAND`）或显式地管理信号屏蔽字来实现。通过设置 `SA_RESETHAND` 标志，信号处理函数执行完毕后，该信号的处理方式会被重置为默认（通常是终止进程），从而防止了同种信号的再次立即处理。

第五章 遇到的问题以及解决方法

5.1 读取磁盘镜像

Ext4 block 的大小为 4096 字节，但是初赛设置的读取字节大小为 512 字节，因此我们要修改相关驱动逻辑。

5.2 exec 改进

我们在运行一些测试点时，发现运行结果并没有像预期一样，于是我们修改了 `exec` 相关逻辑，设计了辅助信息。

5.3 elf 文件读取

我们在进行 elf 文件读取，进行测试点运行时，发现总是出现缺页或者非法指令，而且缺页的地址已经超过虚拟内存的最大值了，这说明我们有可能 elf 文件读取出现了问题，我们的解决的方法是反复读取 elf 文件，进行打印，并且使用二进制工具读取 sdcard.img，观察读取磁盘的位置是否正确。

5.4 wait 相关接口

我们在进行 wait 系统调用时，第一个问题是区分不清楚 sys_wait 和 sys_wait64，我们根据系统调用号区分这两个系统调用；第二个就是 sys_wait64 的 pid 有可能是-1，这样我们原本的逻辑并不支持，尽管子进程退出，父进程仍然死循环，因此我们修改了相关逻辑。这个问题使我们通过阅读 pthread_join 的源码发现的，并且修改正确。

5.5 getdents64 系统调用修改

我们发现进行 getdents64 系统调用时，尤其是 busybox ls 时，结果并不是很正确，于是我们修改逻辑，sys_getdents64 需要返回下一个目录项在父目录中的偏移。

5.6 对于浮点数的支持

对于浮点数，我们需要对 mstatus 进行修改，修改后的逻辑如下：

```
li t5, -6145
```

```
and t6, t5, t6  
  
li t5, 1024 * 2  
  
or t6, t5, t6  
  
li t5, 0x00006000  
  
or t6, t6, t5  
  
csrw mstatus, t6
```

之所以这么修改，是因为运行 time-test 测试点时，总是出现非法指令，打印 elf 之后也发现没有问题，我通过阅读 opensbi 源码时，发现需要相关设计，让内核支持 riscv 的 D 和 F 扩展。

第六章 第二阶段任务

6.1 net 操作

我们设计了 net 相关接口以及数据结构，但是并没有进行相关测试点测试，接下来我讲完善 net 操作。

6.2 文件系统继续完善

在我们设计完文件系统相关操作，比如删除和创建文件，sdcard 就不可以正常的 mount 到 linux 的文件夹，这说明我们的设计还是有纰漏，我们将继续设计，和 linux 接轨。

6.3 文件结构整理

我们现在的文件结构将 kernel 放到主目录中，没有层次，接下来将规范代码。