簡易ジャーナリング機能付き
ファイルシステムの自作

演習 III 第 2 期 — 最終発表

内容

演習 III で取り組んだこと

- 簡易的ジャーナリング付きファイルシステムの自作
  - CTF の Linux カーネル問で頻出のカーネルモジュールを触ってみたかった。
  - ファイルのバイナリ構造を見るのが好き
    - 今まで解読したことがあるもの：BMP, GIF, EXT4(少し)

ファイルシステムとは

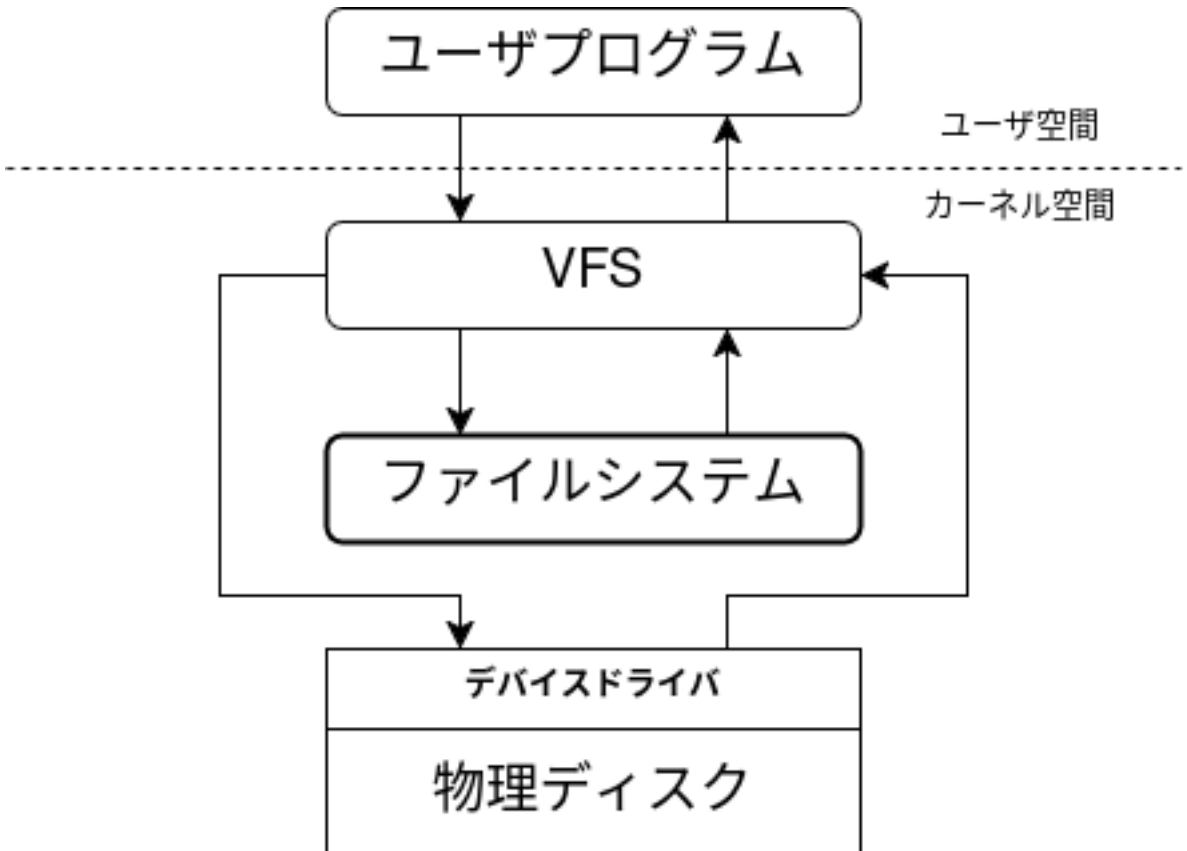
ファイルシステムの役割

- ストレージデバイス上のデータレイアウトを管理する。
  - ディレクトリの木構造とストレージの直列構造を効率良く変換する
  - デバイスへの書き込み途中にエラーが発生しても、データの整合性を保つ（→ ジャーナリング）
  - 例）EXT4, NTFS, BTRFS, FAT32, …

ファイルシステムの実現方法

- 主要なファイルシステムは OS に組み込まれている。
  - `$ cat /proc/filesystems` で使用可能なファイルシステム一覧を確認できる。
  - Linux ではカーネルモジュールとして動的に追加することもできる。

ファイルシステムの階層



何を作ったか

ブロックデバイス上に構築する、Unix 風の小さなファイルシステム LLFS を実装した。

実装した機能

- inode での管理をベースとしたファイル/ディレクトリ構造
  - `loookup`, `create`, `read/write` など簡単なもののみ
- ビットマップによる空きブロック・空き inode の管理
- ジャーナリング（2 段階書き込み）
- クラッシュ後の自動回復

> 目標は、書き込みの途中で電源が落ちても **ファイルシステムが壊れない** こと。 これをジャーナル（先行書込みログ）で保証する。

ファイルシステムの処理

ディスクレイアウト

ディスク先頭から、メタ情報・ジャーナル・データの順に領域を固定配置する。

予約領域	Super block	inode table	inode bitmap	block bitmap	Journal	data blocks
blk 0	blk 1	blk 2	blk 3	blk 4	blk 5-36	blk 37–⋯

通常の書き込みは、まず Journal 領域に記録してから本来の位置へ反映する。 ここが今回の肝になる。

スーパーブロック：構造体の定義

ファイルシステム全体の情報が格納される領域。

C

```c
struct llfs_super_block {
    __le32 magic;              // Magic number (="LLFS")
    __le32 block_size;         // Size of one block
    __le32 itable_block;       // Block number of inode table
    __le32 inode_bitmap_block; // Block number of inode bitmap
    __le32 block_bitmap_block; // Block number of block bitmap
    __le16 inode_size;         // Inode size (64 in llfs)
    __le16 pad;
    __le32 journal_block;      // [JOURNAL] Top block number of journal area
    __le32 journal_blocks;     // [JOURNAL] Number of blocks of journal area
};
```

inode：ファイル 1 個のメタデータ

ファイルやディレクトリのメタデータを格納する。本演習では直接ブロックのみ対応した。

C

```c
#define LLFS_N_BLOCKS 13

struct llfs_inode {
    __le16 mode;                // File mode
    __le16 uid;                 // Owner Uid
    __le32 size;                // File size in bytes
    __le32 blocks;              // Blocks count
    __le32 block[LLFS_N_BLOCKS]; // Pointers to blocks
};

struct llfs_itable {
    struct llfs_inode table[];
};
```

dir_entry

ディレクトリの中身。子ファイルや子ディレクトリの一覧を保持

```c
struct llfs_dir_entry {
    __le32 inode;   // Inode number
    __le16 rec_len; // Entry length
    __u8 name_len;  // Filename length
    __u8 file_type; // File type
    char name[];    // File name
};
```

```
// $ hexdump -C vdisk.img
// ルートディレクトリのデータ
...
00007000  01 00 00 00 0c 00 01 02  2e 00 00 00 01 00 00 00  |................|
00007010  0c 00 02 02 2e 2e 00 00  02 00 00 00 14 00 0a 01  |................|
00007020  65 6d 70 74 79 2d 66 69  6c 65 00 00 03 00 00 00  |empty-file......|
00007030  10 00 05 01 68 65 6c 6c  6f 00 00 00 00 00 00 00  |....hello.......|
00007040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
...
```

mount 時の処理

- モジュールロード時
  - `register_filesystem()`: 自身をファイルシステムとして登録
- mount 時
  1. ルート i ノードの情報を VFS に渡す
  2. ジャーナルログ領域を見て必要があれば回復

lookup 時の処理

VFS からパスが与えられて、ファイルが存在するかしないか、存在するならその情報を返す。

0. 親ディレクトリの i ノードと、探したいファイル名が渡される
1. 親ディレクトリの dir_entry を見に行って、探索
   - 探索アルゴリズムが重要(LLFS では線形探索)
2. ファイル名に合致するエントリが見つかったら、対応する i ノードを見に行って情報を VFS に返す

create 時の処理

0. 親ディレクトリと、作りたいファイルのファイル名やその他の情報(ファイル権限など)が渡される
1. 新規ファイルに割り当てるための空き i ノード番号を取得する
2. i ノードテーブルに書き込み
3. 親ディレクトリの dirent を更新

(open)/read/write 時の処理

iomap という API を使って処理：i ノード情報と論理ブロック番号をもらって、物理ブロック番号を返すような関数を設定する。

1. i ノードテーブルのブロックテーブルを見に行く
2. すでにブロックが割り当てられていたら、その物理ブロック番号を返す
3. ブロックが割り当てられていなかったら、
   - read の場合は穴(HOLE)として報告する
   - write の場合は、空きブロックを確保して、それを返す。同時に i ノードブロックも更新

iomap のうれしいところ：
論理ブロックや物理ブロックの対応関係を VFS に渡してあげれば、読み書きは VFS が勝手に制御してくれる。

ジャーナリングの仕組み

なぜジャーナルが要るのか

1 回のファイル操作は、複数ブロックの更新に分解される。

> **更新の途中でクラッシュすると…**
> ブロックは確保してある(ブロックビットマップを更新済み)なのに、iノードテーブルが更新されていないため、使われないブロックが生まれる可能性

解決策：Write-Ahead Logging（WAL）
本来の場所を書き換える前に、「これからやる更新一式」をジャーナルへ先に書いておき、 途中で落ちても再起動時にジャーナルを見ればやり直せる。

トランザクションの処理

1. ログ領域に書き込み
2. 最後に書き込み完了フラグを立てる
3. 本物の領域に書き込み
4. 書き込みが終わったらログを無効にする

> **境界は「コミット」**
> 書き込み完了フラグが書けた瞬間に、その更新を「確定」とみなす。 2.未満で落ちた更新は捨て、2.以上で落ちた更新は次回の mount 時に 3.をやり直す。

コミット処理(create 編)

変更ログを保持しておき、最後に書き込みをする。

```c
    /* A. 旧 commit ブロックを無効化(seq 再利用による torn-commit 誤検出を防ぐ)。
     *    これ以降クラッシュしても commit magic は無効なので「未コミット」と判定される。 */
    bh = sb_bread(sb, commit_off);
    if (!bh)
        goto fail;
    memset(bh->b_data, 0, sb->s_blocksize);
    jwrite_sync(bh);
    brelse(bh);
    jbarrier(sb);
```

```c
    /* B. descriptor を書く(magic, seq, n, target[]) */
    bh = sb_bread(sb, jb);
    if (!bh) goto fail;
    memset(bh->b_data, 0, sb->s_blocksize);
    desc = (struct llfs_jrnl_desc *)bh->b_data;
    desc->magic = cpu_to_le32(LLFS_JRNL_DESC_MAGIC);
    desc->sequence = cpu_to_le32(seq);
    desc->n_blocks = cpu_to_le32(n);
    for (i = 0; i < n; i++)
        desc->target[i] = cpu_to_le32(t->target[i]);
    jwrite_sync(bh);
    brelse(bh);

    /* C. data ブロック(変更後のメタを丸ごとジャーナルへコピー) */
    for (i = 0; i < n; i++) {
        struct buffer_head *jbh = sb_bread(sb, jb + 1 + i);
        if (!jbh)
            goto fail;
        memcpy(jbh->b_data, t->bh[i]->b_data, sb->s_blocksize);
        jwrite_sync(jbh);
        brelse(jbh);
    }
```

```c
    /* D. ── BARRIER ── descriptor + data を確実に永続化 */
    jbarrier(sb);

    /* E. commit ブロックを書く ← ★コミット点★ */
    bh = sb_bread(sb, commit_off);
    if (!bh)
        goto fail;
    cmt = (struct llfs_jrnl_commit *)bh->b_data;
    cmt->magic = cpu_to_le32(LLFS_JRNL_COMMIT_MAGIC);
    cmt->sequence = cpu_to_le32(seq);
    jwrite_sync(bh);
    brelse(bh);
    jbarrier(sb);

    /* F. チェックポイント: ここで初めて最終位置(in-place)へ反映する */
    for (i = 0; i < n; i++) {
        mark_buffer_dirty(t->bh[i]);
        sync_dirty_buffer(t->bh[i]);
    }
    jbarrier(sb);
```

```c
    /* G. ジャーナルをクリア(次回マウントで replay されないように) */
    bh = sb_bread(sb, jb);
    if (!bh)
        goto fail;
    desc = (struct llfs_jrnl_desc *)bh->b_data;
    desc->magic = 0;
    jwrite_sync(bh);
    brelse(bh);
    jbarrier(sb);

    /* H. pin を解放してトランザクションを破棄 */
    for (i = 0; i < n; i++)
        brelse(t->bh[i]);
    sbi->cur_txn = NULL;
    kfree(t);
    return 0;
```

クラッシュからの回復

mount 時にジャーナルを走査し、フラグが立っているトランザクションだけを再適用する。

C

```c
// マウント時に必ず呼ぶ
int llfs_journal_recover(struct super_block *sb) {
    ...
    n = le32_to_cpu(desc->n_blocks);
    seq = le32_to_cpu(desc->sequence);

    if (le32_to_cpu(cmt->magic) != LLFS_JRNL_COMMIT_MAGIC ||
        le32_to_cpu(cmt->sequence) != seq) {
        /* 未コミット(torn write)→ 破棄。in-place は手付かずなので旧状態で一貫。 */
        ...
        return 0;
    }
```

C

```c
    /* コミット済み → redo(同一内容の上書きなので冪等) */
    for (i = 0; i < n; i++) {
        u32 target = le32_to_cpu(desc->target[i]);
        struct buffer_head *src = sb_bread(sb, jb + 1 + i);
        struct buffer_head *dst;

        ...

        memcpy(dst->b_data, src->b_data, sb->s_blocksize);
        mark_buffer_dirty(dst);
        sync_dirty_buffer(dst);
        brelse(dst);
        brelse(src);
    }
    jbarrier(sb);
```

デモ

まとめ

できたこと

- inode ベースの小さなファイルシステム
- 簡易的なジャーナリングシステム

今後の課題

- ls, remove, mkdir などへの対応
- B 木ベースのファイルシステム
  - Copy on Write をやりたかった
- FUSE (Filesystem in Userspace)
  - ファイルシステムが落ちるとカーネル全体が落ちるのは危ない