Operating Systems Project: Topic 10

Kernel-Level Buffer Cache Instrumentation

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.27

Outline

Outline

The Mission: Topic 10 Requirements

Objective: Analyze and optimize the Kernel Buffer Cache mechanism.

Requirement 1: Instrumentation

- **Target:** Modify `fs/buffer.c` (or related memory paths).
- **Task:** Log Cache Hits vs. Cache Misses.
- **Output:** Expose statistics via a new file: `/proc/cache_stats`.

Requirement 2: Custom Policy

- **Task:** Implement a custom cache replacement strategy.
- **Example:** Frequency-based (LFU) instead of the default LRU-like approach.

Advanced Options

Option A: Workload Characterization

- **Task:** Compare cache behavior under different patterns (Sequential Read vs. Random Write vs. Database).
- **Analysis:** Why does LRU fail for "Scan" workloads?

Option B: Visualizing the Hotspot

- **Task:** Create a heatmap of cached blocks. Which files are currently in RAM?
- **Challenge:** Mapping buffer heads back to filenames.

Outline

Deep Analysis: Topic 5 vs. Topic 10 (Conceptual)

While both manage memory, their **Triggers** and **Goals** are opposite.

Topic 5: Page Replacement

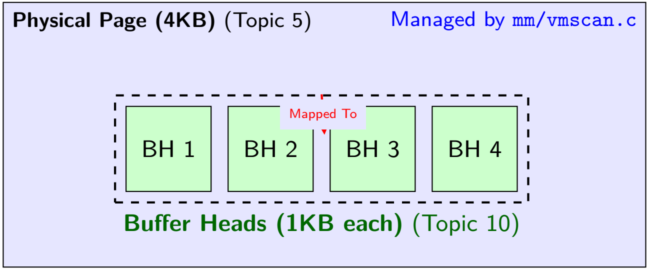
- **Role:** The "Janitor" (Garbage Collector).
- **Trigger:** **Memory Pressure** (RAM is full).
- **Question:** "Who do I kill?"
- **Mechanism:** Scanning LRU lists to find cold pages to swap out.
- **Key Metric: Page Fault Rate** (Minimizing disk access due to eviction).

Topic 10: Buffer Cache

- **Role:** The "Librarian" (Lookup Service).
- **Trigger:** **File Access** (Read/Write request).
- **Question:** "Do we have this?"
- **Mechanism:** Hashing/Tree search to find specific blocks.
- **Key Metric: Hit Rate** (Maximizing logical lookups served from RAM).

Deep Analysis: The Structural Relationship

Modern Linux uses a **Unified Page Cache**, but the data structures differ.



- **Topic 5** treats the whole 4KB page as one unit (Active/Inactive lists).
- **Topic 10** manages the `buffer_head` metadata attached to the page (fs/buffer.c).

Deep Analysis: The Code Path

Where do you insert your hooks? The battlefields are totally different.

Topic 5 (Global Reclaim)

- **Entry:** `kswapd` (Background thread) or Direct Reclaim.
- **Key Function:** `shrink_page_list()`.
- **Logic:**

```
if (!PageReferenced(page)) {
    reclaim_page(page);
}
```

- **Focus:** Checking hardware bits (A-bit) to guess "Recency".

Topic 10 (Filesystem Layer)

- **Entry:** `ext4_read_block` calls buffer layer.
- **Key Function:** `__find_get_block()`.
- **Logic:**

```
bh = lookup_hash(block, size);
if (bh) {
    hit_count++;
    return bh;
}
```

- **Focus:** Managing the Hash Table / Radix Tree lookup success.

Why Topic 10 Still Matters (Despite Unified Cache)

If Page Cache stores file data, what does Buffer Cache do today?

Metadata Matters

While file contents are in Page Cache, **Filesystem Metadata** still relies heavily on Buffer Heads.

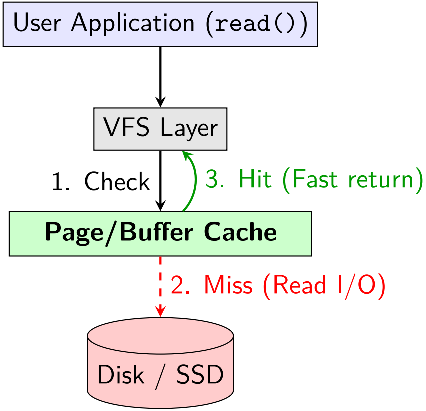
- **Superblocks**, **Inode Tables**, **Bitmaps**, **Directory Entries**.

The Topic 10 Nuance

In Topic 10, you are likely optimizing the caching of **Metadata** operations (e.g., ‘ls -R‘, creating millions of files), whereas Topic 5 optimizes the caching of **Data** content.

Theory 1: The Gap Redux

Disk access is expensive. We need a copy in RAM.



Two Layers:

- **Page Cache:** Caches file data (4KB Pages).
- **Buffer Cache:** Caches block metadata (Superblocks, Inodes, Bitmaps).

Theory 2: Replacement Policies (LRU vs LFU)

RAM is finite. When full, who leaves?

LRU (Least Recently Used)

- *"If you haven't been used lately, you probably won't be used soon."*
- **Standard:** Good for locality.
- **Weakness:** One-time scans flush hot data.

LFU (Least Frequently Used)

- *"If you are rarely used, get out."*
- **Requires:** A counter per block.
- **Strength:** Protects hot data from scans.
- **Weakness:** "Ghost" items (popular once, never again) stay forever.

Outline

Mechanics 1: The buffer_head (bh)

In fs/buffer.c, the atomic unit is the buffer_head. It maps a part of a page to a disk block.

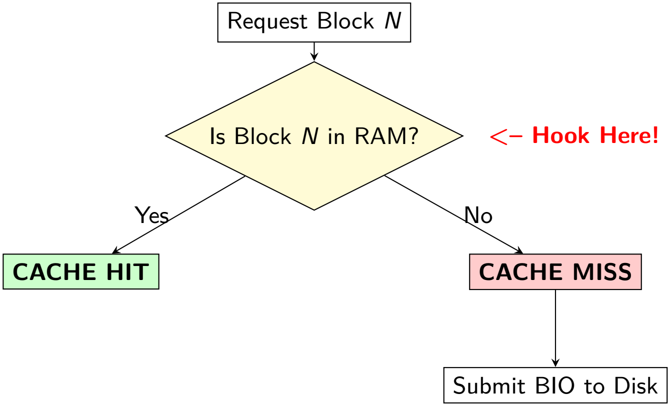
```c
struct buffer_head {
    unsigned long b_state;      // Uptodate? Dirty?
    struct buffer_head *b_this_page;
    struct page *b_page;        // Backing Page

    sector_t b_blocknr;         // Block # on disk
    size_t b_size;              // Block size

    atomic_t b_count;           // Users count
    ...
};
```

Key Functions:

- submit_bh(): Sends I/O to disk.
- mark_buffer_dirty(): Mark for writeback.
- brelse(): Release (decrement count).

Mechanics 2: The Lookup Flow (Hit/Miss)

How does the kernel find a block?



Project Strategy: Find the specific function that performs this check (likely `__find_get_block` or similar) and insert your counters.

Outline

Step 1: Instrumentation Points

Open fs/buffer.c.

Function 1: __find_get_block

This function searches the cache.

- If it returns a bh, it's a **HIT**.
- If it returns NULL, it's a **MISS** (and caller will trigger I/O).

Global Counters:

```
static unsigned long total_hits = 0;
static unsigned long total_misses = 0;
// Add spinlock for safety!

```

Step 2: Custom Replacement (LFU)

Modifying the global replacement policy is hard (it's deeply integrated into `mm/vmscan.c`).

Simpler Approach for Project:

1. Add a field `unsigned int access_count` to `struct buffer_head` (in `include/linux/buffer_head.h`).
2. Increment it in `touch_buffer()`.
3. In the reclaim path, prioritize keeping buffers with high counts.

Note: Modifying a core struct requires a full kernel recompile and is risky. Test in VM!

Step 3: Creating /proc/cache_stats

You need to see the numbers.

```c
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

static int my_stats_show(struct seq_file *m, void *v) {
    seq_printf(m, "Buffer Cache Hits: %lu\n", total_hits);
    seq_printf(m, "Buffer Cache Misses: %lu\n", total_misses);
    seq_printf(m, "Hit Rate: %lu%%\n",
               (total_hits * 100) / (total_hits + total_misses + 1));
    return 0;
}
// Register this in module_init

```

Outline

Step 4: Generating Workloads

How to prove your instrumentation works?

Scenario 1: Cold Cache

1. Drop caches: `echo 3 > /proc/sys/vm/drop_caches`
2. Read a large file: `cat file.txt > /dev/null`
3. **Expect:** High Misses.

Scenario 2: Warm Cache

1. Read the *same* file again immediately.
2. **Expect:** High Hits.

Tooling: Use `fio` for complex patterns (Zipfian distribution mimics databases).

Resources & Next Steps

Action Plan:

1. **Trace:** Use `ftrace` to confirm `__find_get_block` is called during reads.
2. **Modify:** Add counters to `fs/buffer.c`.
3. **Export:** Implement the proc file.
4. **Test:** Run the Cold/Warm experiments and graph the Hit Rate.

Resources:

- *Linux Kernel Development*: Chapter 12 (Memory Management) 16 (Page Cache).
- `fs/buffer.c` source code.