Operating Systems Project: Topic 7

Copy-on-Write (COW) Optimization Study

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.22

Outline

Outline

The Mission: Topic 7 Requirements

Objective: Analyze the behavior and performance of Copy-on-Write.

Requirement 1: Analysis (The "Observer")

- **Task:** Measure the latency of memory copying during COW.
- **Metrics:** How long does a write fault take? How does it scale with page size?

Requirement 2: Modification (The "Hacker")

- **Task:** Modify the COW trigger logic.
- **Example:** Implement **Pre-Copy** (copy neighboring pages speculatively) or **Delayed Allocation**.

Key Question:
Why copy a whole page if I only change 1 byte?

Advanced Options

Option A: Manual COW Trigger

- **Task:** Implement a custom system call to manually trigger COW on a memory range.
- **Use Case:** Snapshotting a database in user space without forking.

Option B: Huge Page COW

- **Challenge:** Copying 4KB is fast. Copying 2MB (Huge Page) causes latency spikes.
- **Task:** Optimize COW for Transparent Huge Pages (THP), perhaps by breaking them into 4KB pages only when needed.

Outline

Theory 1: Virtual Memory Recap (The Foundation)

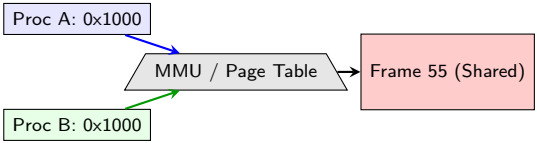
Before we understand COW, we must recall how memory works.

Key Concept: Indirection

- Processes never see physical RAM directly.
- They see **Virtual Addresses** (VA).
- The Hardware (MMU) translates VA to Physical Address (PA) using **Page Tables**.

Why is this important for COW?

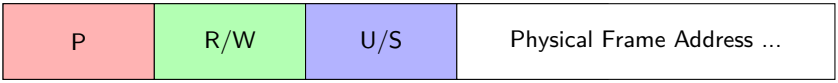
- Because multiple Virtual Addresses (from different processes) can point to the **Same Physical Page**.



Theory 2: The Power of Permissions (R/W Bits)

The Page Table does more than translation; it enforces **Protection**.

Page Table Entry (PTE) - 64 bits



The COW Trick:

- The OS marks a page as **Read-Only** in the hardware (R/W = 0).
- Even if the process *thinks* it has write permission logic, the hardware says NO.
- Writing to it triggers a **CPU Exception (Page Fault)**. This is how the kernel gets notified!

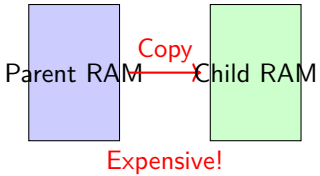
Theory 3: The Fork Problem

Historically, `fork()` created a **Complete Duplicate** of the parent's memory.

The Naive Approach (Deep Copy):

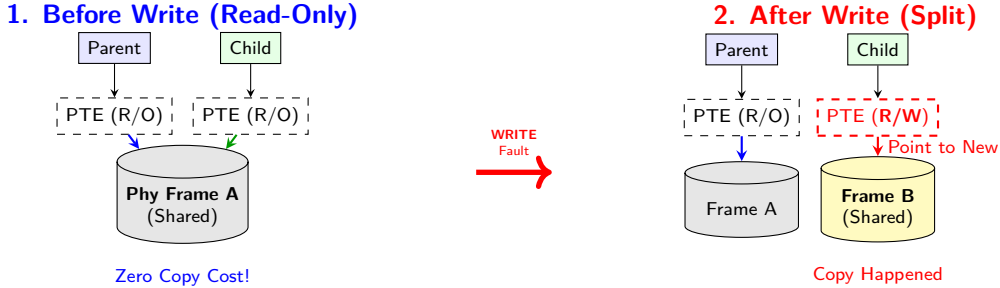
1. Parent has 1GB RAM.
2. `fork()` called.
3. Kernel pauses Parent.
4. Kernel allocates new 1GB frames.
5. Kernel copies 1GB data.
6. Resume.

Problem: Slow! And wasteful, because Child often calls `exec()` immediately, discarding that 1GB copy.



Theory 4: The COW Solution

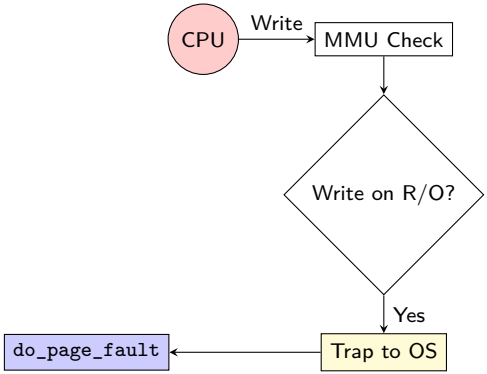
Core Concept: Sharing enables instant forking. Copying is delayed until absolutely necessary.



Outline

Mechanics 1: The Trap

What happens when the Child tries to write to that Read-Only page? **Hardware raises a Page Fault (Exception 14 on x86).**



The Kernel looks at the **VM Area Struct (VMA)**.

- If VMA says "This should be writable", but PTE says "Read-Only", the Kernel knows: **This is a COW page.**

Mechanics 2: The do_wp_page Function

This is the heart of COW logic in `mm/memory.c`.

1. **Check Reference Count:** How many processes map this page?
2. **Case A (Count == 1):** I am the last user.
   - No need to copy!
   - Just change PTE to Read/Write. Done.
3. **Case B (Count > 1):** Others are sharing it.
   - Allocate **New Page**.
   - **Copy** data from old page to new page.
   - Update my PTE to point to New Page (R/W).
   - Decrement count on old page.

```
1  // Pseudo-code mm/memory.c
2  int do_wp_page(...) {
3    old_page = vm_normal_page(vma,
       ...);
4
5    if (page_count(old_page) == 1) {
6        // Reuse!
7        make_writable(pte);
8        return;
9    }
10
11   // Copy!
12   new_page = alloc_page(...);
13   copy_user_highpage(new_page,
       old_page);
14   set_pte_at(..., new_page, RW);
15 }
16
```

Outline

Step 1: Locating the Code

You need to modify the memory management subsystem.

Key Files

- `mm/memory.c`: Contains `handle_pte_fault` and `do_wp_page`. This is your main workspace.
- `include/linux/mm.h`: Definitions of page flags.

Strategy for Requirement 1 (Latency Analysis):

- Wrap the `copy_user_highpage` function call with timing code.
- Use `ktime_get()` before and after.
- Printk the delta.

Step 2: Measuring Latency

```
// Inside do_wp_page or wp_page_copy
ktime_t start, end;
s64 actual_time;

start = ktime_get();

// The heavy operation
copy_user_highpage(new_page, old_page, vma_addr, vma);

end = ktime_get();
actual_time = ktime_to_ns(ktime_sub(end, start));

printk(KERN_INFO "COW: Copy took %lld ns for PFN %lx\n",
       actual_time, page_to_pfn(old_page));

```

Tip: Don't print every time! You will hang the system. Print only if latency $>$ threshold or sample 1 in 1000 events.

Step 3: Implementing Pre-Copy (Idea)

Hypothesis: Spatial Locality. If I write to Page N, I will likely write to Page $N+1$ soon.

Your Modification

Inside the fault handler:

1. Handle the current fault (Copy Page N).
2. **Look Ahead:** Check if Page $N+1$ exists and is also COW-shared.
3. If yes, allocate a new frame for $N+1$ and copy it **now**.
4. Update PTE for $N+1$ to Read/Write.

Benefit: Reduces the number of Page Fault Exceptions (Traps), which are expensive. **Risk:** Wasted memory if I don't actually write to $N+1$.

Resources & Next Steps

Action Plan:

1. **Setup:** Compile vanilla kernel.
2. **Trace:** Locate `do_wp_page` in `mm/memory.c`. Add a log.
3. **Test:** Run a program that forks and writes. Check `dmesg`.
4. **Measure:** Add timing code. Generate latency histogram.
5. **Hack:** Implement pre-copy logic for the next page.

Resources:

- *Understanding the Linux Kernel*: Chapter on Memory Addressing / Page Faults.
- Intel SDM (Software Developer Manual) Vol 3: Interrupt 14.