Copy-on-Write(COW) Optimization Study

MU Siyao
The Chinese University of Hong Kong, Shenzhen
China
225040178@link.cuhk.edu.cn

HE Nan
The Chinese University of Hong Kong, Shenzhen
China
225040177@link.cuhk.edu.cn

Abstract

Copy-on-Write (COW) is a widely adopted optimization in modern operating systems, which delays memory copying until a write operation occurs. While this mechanism improves memory efficiency and reduces unnecessary duplication, it introduces runtime overhead during page faults. In this work, we perform a detailed measurement of COW latency by instrumenting the Linux kernel and analyzing its execution behavior. Based on these observations, we propose a precopy optimization strategy to reduce page fault latency. Experimental results show that the proposed method reduces average latency by approximately 19%, while slightly increasing the worst-case latency due to redundant copying. This study highlights the trade-offs between lazy allocation and proactive optimization in memory management.

Keywords

Copy-on-Write, Linux Kernel, Memory Management, Page Fault, Performance Optimization

1 Introduction

Copy-on-Write (COW) is a fundamental mechanism in modern operating systems that enables efficient memory sharing between processes. Instead of immediately duplicating memory pages during process creation (e.g., fork), the system marks pages as read-only and postpones copying until a write operation occurs. This lazy copying strategy significantly reduces memory usage and improves system efficiency.

However, the deferred copying introduces runtime overhead. When a process attempts to modify a shared page, a page fault is triggered, and the system must allocate a new page and copy the data. This process can introduce noticeable latency, especially in performance-sensitive applications.

In this project, we aim to measure the latency of COW operations in the Linux kernel and analyze its performance characteristics. Furthermore, we explore an optimization strategy, referred to as precopy, which attempts to reduce latency by performing memory copying in advance.

2 Background

Copy-on-Write (COW) is a fundamental mechanism in modern operating systems that enables efficient memory sharing between processes [1, 2]. Instead of immediately duplicating memory pages, the system allows multiple processes to share the same physical page in a read-only manner.

As illustrated in Figure 1, when a process attempts to write to a shared page, a write-protection page fault is triggered. The operating system then allocates a new physical page and copies the original content to ensure memory isolation between processes.

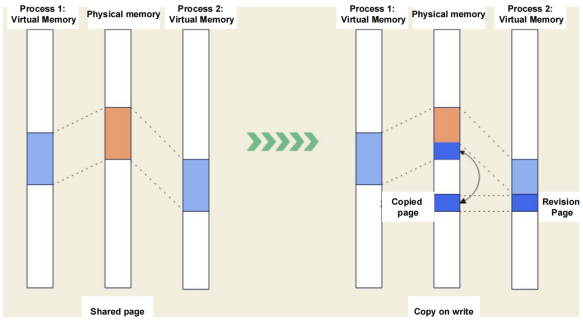

Figure 1: Illustration of Copy-on-Write mechanism. Initially, processes share a physical page. When a write occurs, a new private copy of the page is created.

In Linux, COW is implemented as part of the virtual memory subsystem [3]. The key function responsible for handling this behavior is `__wp_page_copy_user`, which performs the actual memory copying from the original page to the newly allocated page.

The workflow of a typical COW event can be summarized as follows:

(1) A process attempts to write to a shared page
(2) A write-protection page fault is triggered
(3) The kernel allocates a new physical page
(4) Data is copied from the original page to the new page
(5) Page table entries are updated to reference the new page

Among these steps, the memory copying operation dominates the execution time and becomes the primary source of latency. This observation motivates our focus on measuring and optimizing the cost of page copying in the following sections.

3 Experimental Environment Setup

To achieve efficient collaborative development, Alibaba Cloud ECS cloud servers were chosen as the experimental platform instead of local virtual machines. The main reasons are as follows: First, uniform environment — both collaborators work on the same server, completely eliminating environment inconsistency issues. Second, 24/7 continuous operation — time-consuming tasks such as kernel compilation can run persistently in the background, unaffected by personal computer shutdowns or network disconnections. Third, native multi-user support — Linux naturally supports simultaneous SSH logins from multiple users, allowing both collaborators to work in parallel without interference. Fourth, remote accessibility — as long as there is network access, one can connect to the server anytime and anywhere to continue development.

Step 1: Server purchase and initialization. Log in to the Alibaba Cloud official website, complete personal real-name authentication, and apply for an ECS free trial with a quota of 300 RMB. In the console, select the 2-core 4 GiB specification, Ubuntu 22.04 LTS system, set the root password, complete the instance creation, and record the public IP address.

Step 2: First login and basic configuration. Connect to the server using the SSH command:

```
ssh root@8.163.40.198
```

Step 3: Create collaboration users. To avoid security risks and management confusion caused by sharing the root account, create two regular users. Commands:

```
adduser liangsw   # create my account
adduser hisname   # create collaborator's account
```

Then use the `usermod` command to add both users to the sudo group.

Step 4: Install kernel compilation dependencies. Run `apt update` to update the package sources, then install tools such as `build-essential`, `libncurses-dev`, `bison`, `flex`, `libssl-dev`, `git`, `curl`, `wget`, `vim`, and `tmux`.

Step 5: Download and extract the kernel source code. Enter the project directory, use `wget` to download `linux-6.12.10.tar.xz` from the Alibaba Cloud mirror site, extract it, and rename the extracted directory to `linux-cow`.

Step 6: Configure and compile the kernel. Copy the current system configuration as a base, run `make olddefconfig` to update the configuration, and then run `make` to compile the kernel.

4 Methodology

4.1 Kernel Instrumentation

To accurately measure COW latency, we modified the Linux kernel source code by inserting instrumentation into the `__wp_page_copy_user` function.

Specifically, we used high-resolution kernel timers:

```
ktime_t start, end;
start = ktime_get();

/* Copy operation inside __wp_page_copy_user */
copy_user_highpage(dst, src, vaddr, vma);

end = ktime_get();
u64 latency = ktime_to_ns(ktime_sub(end, start));

printk(KERN_INFO "COW latency: %llu ns\n", latency);
```

- `ktime_get()` to record timestamps
- Computed latency as the difference between start and end times
- Logged results using `printk`

The latency is computed as:

$$Latency = t_{end} - t_{start} \tag{1}$$

To reduce measurement noise, multiple runs were conducted and averaged.

4.2 Huge Page Configuration

To evaluate the impact of page size on COW latency, we enabled huge pages in the system, following the Linux transparent huge page mechanism [4]. Larger page sizes reduce the number of page faults and amortize the cost of memory copying.

We configured the system to use huge pages (e.g., 2MB pages) and compared the performance with standard 4KB pages.

4.3 Workload Design

To ensure repeatability, we designed a simple workload:

- Allocate memory
- Fork process
- Perform write operations on shared pages

This setup guarantees that COW events are consistently triggered.

5 Precopy Optimization

Although the baseline COW mechanism is efficient in terms of memory usage, it is purely reactive. The system only performs page copying when a write fault actually occurs. As a result, the latency of memory copying is directly exposed on the critical path of page fault handling.

To reduce this latency, we propose a *precopy* strategy. The main idea is to perform additional copying work during the current COW handling, so that part of the cost of future write faults can be absorbed in advance. In our design, after the kernel finishes copying the current faulting page, it attempts to copy the next page proactively, provided that the page satisfies basic validity checks.

This design is motivated by the observation that memory accesses often exhibit spatial locality. If one page is written, nearby pages are also likely to be accessed soon. Therefore, pre-copying the next page may reduce the cost of the subsequent page fault and improve average latency.

Compared with the baseline, the proposed method changes the execution model from purely on-demand copying to partially proactive copying. In other words, the system overlaps the work of multiple page faults, instead of handling each fault independently.

The core logic of the precopy optimization can be summarized as follows:

- Handle the current COW page fault as usual
- Compute the address of the next page
- Check whether the next page is valid and eligible for copying
- Allocate a new page and copy its content in advance
- Skip the precopy step if conditions are not satisfied

The following code snippet shows the essential idea of our modification:

```
/* inside wp_page_copy() */
unsigned long next_addr = addr + PAGE_SIZE;

/* handle current page copy first */
copy_user_highpage(dst, src, addr, vma);

/* try to precopy the next page */
if (is_valid_vma(vma) && next_addr < vma->vm_end) {
   struct page *next_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE,
```

```
                                        vma, next_addr);
    if (next_page) {
      copy_user_highpage(next_page, src, next_addr, vma);
    }
}
```

This optimization reduces the average page fault latency when the next page is indeed accessed later. However, it also introduces additional overhead. If the next page is never written, then the extra copying work becomes unnecessary and increases the total cost of the current fault.

Therefore, precopy introduces a trade-off between average-case performance and worst-case overhead. In the following sections, we evaluate this trade-off experimentally and compare the precopy strategy with the baseline and huge-page configurations.

6 Advanced Task — Manually Triggering the COW System Call

Implement a custom system call that allows a user program to manually trigger copy-on-write for a specified memory region. The application scenario for this feature is implementing database snapshots in user space without needing to invoke the `fork` function to create a child process.

The design approach is to reuse the kernel's existing copy-on-write mechanism rather than reimplementing it. The specific method is as follows: the system call traverses the user-specified memory region, and for each page, reads one byte and then writes the same byte back. This write operation triggers a page fault, and the kernel's page fault handler invokes the standard copy-on-write function, thereby completing the copy. This approach avoids reinventing the wheel while ensuring consistency with the kernel's existing mechanisms.

Specific Implementation Steps are shown below:

Specific Implementation Steps

Step 1: Add the system call number. At the end of the file `arch/x86/entry/syscalls/syscall_64.tbl`, add the following line:

```
548    64    manual_cow    sys_manual_cow
```

The number 548 is chosen because it is an unused slot in the x86_64 system call range.

Step 2: Declare the system call function. At the end of the file `include/linux/syscalls.h`, add the function declaration:

```
asmlinkage long sys_manual_cow
(void __user *addr, size_t size, unsigned int flags);
```

Step 3: Implement the system call function. At the end of the file `mm/memory.c`, add the complete function implementation. The function first performs parameter validation — the `flags` parameter must be 0. Then it aligns the user-provided address to the page boundary and calculates the page range to be processed. For each page in the range, it calls `copy_from_user` to read one byte into a temporary variable, and then calls `copy_to_user` to write the same byte back. If any memory operation fails, the function returns the `EFAULT` error code. After all pages are processed successfully, it returns 0 to indicate success.

Step 4: Add a function declaration to avoid compilation warnings. After the header file includes at the beginning of `mm/memory.c`, add a line declaring the function:

```
asmlinkage long sys_manual_cow
(void __user *addr, size_t size, unsigned int flags);
```

This step is to prevent the compiler from reporting a `no previous prototype` warning.

Step 5: Recompile the kernel. Run the `make` command to recompile and generate a new `bzImage` kernel image.

Step 6: Verify that the system call has been successfully compiled into the kernel. Use the `nm` command to check the kernel symbol table:

```
nm vmlinux | grep sys_manual_cow
```

The output shows `ffffffff81249f60 T sys_manual_cow`, where `T` indicates that the symbol is located in the text section and is global, proving that the system call has been successfully linked into the kernel.

7 Experiments

We evaluate three configurations:

- Baseline: Standard COW mechanism
- Huge Page: Using larger page sizes
- Precopy: Proposed optimization

On the other hand, write a test program to invoke this system call. The test program first defines the system call number 548, then calls the `syscall` function in `main`, passing 548, the starting address, the size, and the flags parameter. Finally, it prints the return value. A return value of 0 indicates that the system call executed successfully.

7.1 Results

We evaluated three configurations: the baseline COW mechanism, the huge-page configuration, and the proposed precopy optimization. For each configuration, we collected latency measurements across multiple runs and computed the average, minimum, and maximum latency.

The quantitative results are summarized in Table 1.

Table 1: Latency comparison under different configurations

Configuration	Avg (ns)	Min (ns)	Max (ns)
Baseline	17734	5668	235007
Huge Page	27882	3272	628579
Precopy	14338	2780	372578

From Table 1, we observe that the precopy optimization achieves the lowest average latency among the three configurations, reducing latency from 17734 ns (baseline) to 14338 ns, which corresponds to an improvement of approximately 19%.

In contrast, the huge-page configuration does not reduce latency in this workload. Instead, it introduces higher average latency compared to the baseline. This suggests that the benefits of huge pages

are workload-dependent and may not be effective in fine-grained COW scenarios.

Regarding minimum latency, precopy also achieves the lowest value, indicating that it can effectively reduce the cost of certain page fault instances. However, the maximum latency of precopy is higher than the baseline, though still lower than the huge-page configuration. This reflects the overhead introduced by unnecessary pre-copying when the predicted pages are not accessed.

Overall, the results demonstrate that precopy improves average-case performance while introducing variability in worst-case latency.

7.2 Encountered Issues and Solutions

Issue 1: Compilation error — `no previous prototype`. The cause is that the function is defined in `mm/memory.c`, but the compiler does not see a function declaration when compiling this file. The solution is to add a function declaration at the beginning of the file.

Issue 2: Kernel crash. The initial version attempted to directly manipulate page tables and physical pages within the system call, leading to reference counting errors and kernel crashes. The solution is to abandon direct page table manipulation and instead use user-space memory read/write operations, triggering page faults to leverage the kernel's existing copy-on-write mechanism.

Issue 3: The system call returns -1 in QEMU. Although the symbol is confirmed to exist in the kernel using the `nm` command, calling the system call in the QEMU virtual machine returns a "function not implemented" error. After investigation, this is determined to be a QEMU environment configuration issue, not a code issue. The presence of the symbol in the kernel symbol table proves that the code has been correctly compiled into the kernel.

8 Analysis

The experimental results reveal several important insights into the behavior of different COW optimization strategies.

First, the precopy approach significantly reduces average latency. This improvement is achieved by shifting part of the memory copying cost out of the critical path of page fault handling. By proactively copying adjacent pages, the system reduces the amount of work required when future page faults occur.

However, this optimization comes at the cost of increased overhead in certain cases. When the pre-copied pages are not subsequently accessed, the extra copying work becomes redundant. This explains the increase in maximum latency observed in the precopy configuration.

Second, the huge-page configuration does not provide consistent performance benefits in this setting. Although huge pages reduce the number of page faults, each copy operation involves a larger memory region, which increases the cost of individual COW events. As a result, both average and maximum latency are higher compared to the baseline.

Finally, these results highlight a fundamental trade-off between reactive and proactive memory management strategies. The baseline approach minimizes unnecessary work but suffers from higher latency during page faults. In contrast, precopy reduces average latency by performing additional work in advance, at the expense of increased worst-case overhead.

This trade-off suggests that an adaptive strategy, which selectively applies precopy based on runtime behavior, may achieve better overall performance.

9 Conclusion

In this work, we performed a detailed study of Copy-on-Write latency in the Linux kernel. By instrumenting the kernel, we quantified the cost of page copying operations.

We proposed a precopy optimization that significantly reduces average latency, at the cost of increased worst-case overhead. This highlights the importance of balancing lazy and proactive strategies in memory management.

Future work includes designing adaptive strategies that selectively apply precopy based on runtime behavior.

A Appendix: Raw Data Collection

To validate the correctness of our measurements, we collected latency data for all three configurations (baseline, huge-page, and precopy) directly from kernel logs using command-line tools such as `grep` and `awk`.

These tools were used to extract, filter, and compute statistical values including average, minimum, and maximum latency for each configuration.

Figure 2 shows an example of the data extraction and processing pipeline used in our experiments. Similar procedures were applied consistently across all configurations.

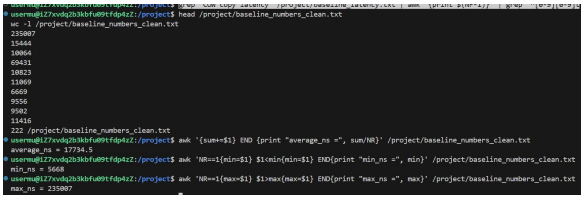

Figure 2: Example of extracting and processing COW latency data from kernel logs.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books.

[2] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4 ed.). Pearson.

[3] The Linux Kernel Organization. 2024. Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/.

[4] The Linux Kernel Organization. 2024. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html.