CS3500: Operating Systems
Lab 5: Copy On Write

16th September 2025

1 Introduction

In this lab, you will implement copy-on-write fork for xv6. This optimization will reduce memory usage and improve performance by allowing parent and child processes to initially share physical pages until one of them writes to a page.

Before you start coding:

- **Read Chapter 3 of the xv6 book**. This chapter is essential for understanding page tables and virtual memory in xv6.
- Review the following `xv6` source files:
  - `kernel/vm.c`: code for managing page tables and mappings.
  - `kernel/kalloc.c`: physical page allocator.
  - `kernel/trap.c`: code handling traps and page faults.
- For deeper understanding, consult the RISC-V privileged architecture manual.

2 Lab Setup

You have been provided with the `cowtest.c` file in the zipped folder. Copy this file into the `user/` directory of your `xv6` repository, and add it to the `Makefile` so that it will be compiled along with the other user programs.

Before proceeding, make sure you are working on the correct commit as mentioned in the mail:

```
git checkout 0c32c04b2931884cecf76a73e708a016908d5db6
```

Once you are on the correct commit, build and run `xv6`:

```
make qemu
```

At this stage, `cowtest` will compile but the tests will fail, as you have not yet implemented copy-on-write. Solving this lab will make `cowtest` (and `usertests -q`) pass successfully.

3 Problem

In this lab, you will add a feature to `xv6` that implements **copy-on-write (COW) fork**. With COW, the kernel maps the parent's physical memory pages into the child instead of allocating new pages and copying the contents. Both processes can then share the same physical pages until one of them writes to a page, at which point the kernel makes a private copy for the writer.

Your solution is correct if it passes `cowtest` and `usertests -q`.

(a) Modify `fork` so that it maps the parent's pages into the child as copy-on-write instead of creating private copies immediately. Both processes should see the same memory contents after the fork.

(b) Implement page fault handling so that when a process attempts to write to a copy-on-write page, the kernel allocates a new physical page, copies the data, and updates the page table entry to give the process a private writable copy.

(c) Ensure that reference counts for physical pages are correctly maintained. When a process exits or unmaps memory, decrement the reference count, and free a page only when no process is using it.

Note: Pay special attention to synchronization and reference counting, since multiple processes may be sharing the same physical page.

`cowtest` stresses your implementation with various cases of fork, write, and memory sharing. Your solution is complete when `cowtest` produces the expected output and `usertests -q` also runs correctly:

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED

$ usertests -q
...
ALL TESTS PASSED
```

Steps:

1. Modify `uvmcopy()` to map the parent's physical pages into the child, instead of allocating new pages. Clear `PTE_W` in the PTEs of both child and parent for pages that originally had `PTE_W` set.

2. Modify `copyout()` to recognize page faults. When a write page fault occurs on a COW page that was originally writable, allocate a new page with `kalloc()`, copy the old page to the new page, and install the new page in the PTE with `PTE_W`

set. Pages that were originally read-only (not mapped `PTE_W`, such as text segment pages) should remain read-only and shared between parent and child; a process that tries to write such a page should be killed

3. Implement page fault handling in `trap.c`. In `usertrap`, when a process tries to write to a copy-on-write page, the kernel should allocate a new physical page, copy the data into it, and update the page table entry to give the process a private writable copy. This is similar to what happens in `copyout`.

4. Ensure that each physical page is freed only when the last PTE reference to it goes away. Maintain a **reference count** for each physical page:

   - Initialize the reference count to one when `kalloc()` allocates a page.
   - Increment the count when `fork` causes a child to share the page.
   - Decrement the count each time any process removes the page from its page table.

   `kfree()` should only place a page back on the free list if its reference count is zero. You can store reference counts in a fixed-size array of integers, indexed by the physical address divided by 4096. The array size can be determined from the highest address initialized by `kinit()` in `kalloc.c`. Modify `kalloc()` and `kfree()` accordingly.

Hints:

1. It may be useful to have a way to record, for each PTE, whether it is a COW mapping. You can use the RSW (reserved for software) bits in the RISC-V PTE for this.

2. `usertests -q` explores scenarios that `cowtest` does not test, so don't forget to check that all tests pass for both.

3. Some helpful macros and definitions for page table flags are at the end of `kernel/riscv.h`.

4. If a COW page fault occurs and there's no free memory, the process should be killed.

4 Submission

- You are required to zip and submit the full folder and a report highlighting the changes you made in the relevant files. Note that only your contributions/modifications to the code are required.

- In the report, very briefly explain the logic/reasoning of the code you implemented for each question. We are looking for the core concept and not a line-by-line explanation of your code.

- Your zipped folder should be named `<Roll_No>_Lab5.zip`.