

Lab: Copy-on-Write Fork for xv6

Virtual memory provides a level of indirection: the kernel can intercept memory references by marking PTEs invalid or read-only, leading to page faults, and can change what addresses mean by modifying PTEs. There is a saying in computer systems that any systems problem can be solved with a level of indirection. This lab explores an example: copy-on-write fork.

To start the lab, switch to the cow branch:

```
$ git fetch
$ git checkout cow
$ make clean
```

The problem

The `fork()` system call in xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time. Worse, the work is often largely wasted: `fork()` is commonly followed by `exec()` in the child, which discards the copied memory, usually without using most of it. On the other hand, if both parent and child use a copied page, and one or both writes it, the copy is truly needed.

The solution

Your goal in implementing copy-on-write (COW) `fork()` is to defer allocating and copying physical memory pages until the copies are actually needed, if ever.

COW `fork()` creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages. COW `fork()` marks all the user PTEs in both parent and child as read-only. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

COW `fork()` makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears. In a simple kernel like xv6 this bookkeeping is reasonably straightforward, but in production kernels this can be difficult to get right; see, for example, [Patching until the COWs come home](#).

Implement copy-on-write fork(**hard**)

Your task is to implement copy-on-write fork in the xv6 kernel. You are done if your modified kernel executes both the `cowtest` and `'usertests -q'` programs successfully.

To help you test your implementation, we've provided an xv6 program called `cowtest` (source in `user/cowtest.c`). `cowtest` runs various tests, but even the first will fail on unmodified xv6. Thus, initially, you will see:

```
$ cowtest
simple: fork() failed
$
```

The "simple" test allocates more than half of available physical memory, and then `fork()`s. The fork fails because there is not enough free physical memory to give the child a complete copy of the parent's memory.

When you are done, your kernel should pass all the tests in both cowtest and usertests -q. That is:

```
$ cowtest
simple: ok
simple: ok
three: zombie!
ok
three: zombie!
ok
three: zombie!
ok
file: ok
ALL COW TESTS PASSED
$ usertests -q
...
ALL TESTS PASSED
$
```

Here's a reasonable plan of attack.

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 have `PTE_W` set.
2. Modify `usertrap()` to recognize page faults. When a write page-fault occurs on a COW page that was originally writeable, 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`, like pages in the text segment) should remain read-only and shared between parent and child; a process that tries to write such a page should be killed.
3. Ensure that each physical page is freed when the last PTE reference to it goes away -- but not before. A good way to do this is to keep, for each physical page, a "reference count" of the number of user page tables that refer to that page. Set a page's reference count to one when `kalloc()` allocates it. Increment a page's reference count when fork causes a child to share the page, and decrement a page's count each time any process drops the page from its page table. `kfree()` should only place a page back on the free list if its reference count is zero. It's OK to keep these counts in a fixed-size array of integers. You'll have to work out a scheme for how to index the array and how to choose its size. For example, you could index the array with the page's physical address divided by 4096, and give the array a number of elements equal to highest physical address of any page placed on the free list by `kinit()` in `kalloc.c`. Feel free to modify `kalloc.c` (e.g., `kalloc()` and `kfree()`) to maintain the reference counts.
4. Modify `copyout()` to use the same scheme as page faults when it encounters a COW page.

Some hints:

- 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.
- `usertests -q` explores scenarios that `cowtest` does not test, so don't forget to check that all tests pass for both.
- Some helpful macros and definitions for page table flags are at the end of `kernel/riscv.h`.
- If a COW page fault occurs and there's no free memory, the process should be killed.

Submit the lab

Time spent

Create a new file, `time.txt`, and put in a single integer, the number of hours you spent on the lab. `git add` and `git commit` the file.

Answers

If this lab had questions, write up your answers in `answers-*.txt`. `git add` and `git commit` these files.

Submit

You will turn in your assignments using the google classroom.

After committing your final changes to the lab, type `make tarball` to create archive which you'll submit to classroom.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make tarball
$
```

If you run `make tarball` and you have either uncommitted changes or untracked files, you will see output similar to the following:

```
M hello.c
M bar.c
```

```
You have uncommitted changes. Please commit or stash them.
make: *** [Makefile:348: handin-check] Error 1
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with **M**. You can cause git to track a new file that you create using `git add filename`.

If `make tarball` does not work properly, try fixing the problem with the Git commands.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make tarball`

Optional challenge exercise

- Measure how much your COW implementation reduces the number of bytes xv6 copies and the number of physical pages it allocates. Find and exploit opportunities to further reduce those numbers.