Lab cow: Copy-on-write fork

courses.cs.washington.edu/courses/cse451/21au/labs/cow.html

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. The lazy allocation lab provided one example. This lab explores another example: copy-on write fork.

To start the lab, update your repository and create a new branch for your solution:

```
$ git fetch origin
$ git checkout -b cow origin/xv6-21au
$ make clean
```

The problem

The `fork()` system ca [illegible] ace memory into the child. If the parent [illegible] work is often largely wasted; for exa [illegible] ill cause the child to discard the copied [illegible] n the other hand, if both parent and chil [illegible] ruly needed.

The solution

The goal of copy-on-w [illegible] ing physical memory pages for the [illegible]

COW `fork()` creates [illegible] memory pointing to the parent's physical p [illegible] oth parent and child as read-only. Wh [illegible] pages, the CPU will force a page fault. [illegible] 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.

Implement copy-on write

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` 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.

When you are done, your kernel should be able to run both `cowtest` and `usertests`. That is:

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

Here's one reasonabl[illegible]

- Modify `uvmcopy(`[illegible]d, instead of allocating new p[illegible]nd parent.
- Modify `usertrap`[illegible]ccurs on a COW page, allocate a[illegible]e new page, and install the new page in the PTE with `PTE_W` set.
- 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 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`.

- Modify `copyout()` to use the same scheme as page faults when it encounters a COW page.

Some hints:

- The lazy page allocation lab has likely made you familiar with much of the xv6 kernel code that's relevant for copy-on-write. However, you should not base this lab on your lazy allocation solution; instead, please start with a fresh copy of xv6 as directed above.
- 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` 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.

This completes the lab `make tarball`, and submit the tarball