Lab1.2: Copy-on-Write Fork - Evaluation Plan

Lab 1: Copy-on-Write Fork for xv6

> 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.

Link to the GitHub Classroom assignment: https://classroom.github.com/a/UDtW9Z8r.

To start, clone the private lab assignment repo from GitHub classroom. Refer to the setup tutorial page for help.

```
$ git clone git@github.com:UCR-CS202/lab-...
```

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. In addition, the copies often waste memory; in many cases neither the parent nor the child modifies a page, so that in principle they could share the same physical memory. The inefficiency is particularly clear if the child calls exec(), since exec() will throw away the copied pages, probably without using most of them. On the other hand, if both parent and child use a page, and one or both writes it, a copy is truly needed.

The solution

The goal of copy-on-write (COW) fork() is to defer allocating and copying physical memory pages for the child 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 not writable. 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.

The cowtest program

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
$
```

Submission2: Evaluation Plan

In the second submission, you need to come up with an evaluation plan. You can refer to the exokernel paper to see how to design experiments for an OS research idea. From high-level, you need to address two key categories of questions.

1. Implementation Correctness. Evaluation results will be invalid if what got evaluated is wrong. So first of all, you need to make sure your implementation actually reflects the solution above. The assignment already contains a cowtest, check this test, and think about what else should be added to make sure you code (1) is correct and (2) will not break other kernel functionalities.
2. Validity of the Idea. CoW-fork is proposed as a performance optimization. Like in the exokernel paper, think about what are the hypotheses behind this idea and design. Then come up with experiments that can validate the hypotheses. In the next assignment, you'll need to implement testing code for these experiments and show whether the results validate or invalidate the hypotheses.

Submit a write-up addressing these problems: (1) how would you test the correctness of your implementation, and (2) what are the hypotheses and what experiments you would do to validate the hypotheses.