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. The lazy allocation lab provided one example. This lab explores another 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; for example, a fork() followed by exec() in the child will cause the child to discard the copied memory, probably without ever using most of it. 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.

Implement copy-on write(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 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. 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 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.
2. Modify usertrap() to recognize page faults. When a page-fault occurs on a COW page, 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.
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 r~~eference 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.
4. 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.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Submit

You will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type **`make handin`** to submit your lab.

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

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let **`make handin`** generate it again (*myapi.key* must not include newline characters).

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

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

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 `??`. You can cause `git` to track a new file that you create using `git add filename`.

If **`make handin`** does not work properly, try fixing the problem with the curl or Git commands. Or you can run **`make tarball`**. This will make a tar file for you, which you can then upload via our web interface.

> - Please run `make grade` to ensure that your code passes all of the tests
> - Commit any modified source code before running `make handin`
> - You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional challenge exercises

- Modify xv6 to support both lazy page allocation and COW.
- 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.