Project 5

Due Mar 30 by 11:59pm **Points** 10 **Available** Mar 21 at 12am - Mar 30 at 11:59pm

This assignment was locked Mar 30 at 11:59pm.

CS537 Spring 2023 - Project 5: Copy-on-Write (CoW) Fork in xv6

In this project, you will change the current xv6 fork() implementation to use copy-on-write (CoW). The current version does a simple copy of each page in the address space. You will modify the xv6 kernel to do copy-on-write instead.

Administrivia

- **Due Date** by March 30th, 2023 at 11:59 PM.
- Questions: We will be using Piazza for all questions.
- Collaboration: This is a group assignment. It can be solved individually, or in a group of 2 students. We highly recommend doing this in a group of 2. Make sure you are assigned a group on canvas for this project.
- Copying code (from others) is considered cheating. **Read this (http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html)** for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- Some tests will be provided at `~cs537-1/tests/P5`. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P5/README` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Compress your modified version of xv6 using `tar -czvf P5.tar.gz path-to-your-xv6-dir/` and copy it to `~cs537-1/handin/login/P5` where login is your CS login. Please run `make clean` before compressing. **If you are working in a group of 2, both partners need to submit the same final code with all the changes under their handin. You need to do the same for README.md. The README.md should include the names of both partners including the section and group number.**

Introduction

The goal of this project is to become familiar with Unix-style forking and x86 memory management. To achieve this, you will convert the simple fork() implementation in xv6 to a copy-on-write fork(). This will involve writing a trap handler for page faults, augmenting the physical memory management code, and, of course, manipulating page tables.

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.

Before you begin

- You will be working on the xv6 copy provided specifically for this project. Copy the xv6 source code using the following commands:

```
prompt> cp ~cs537-1/public/xv6_p5_scratch.tar.gz .
prompt> tar -xvf xv6_p5_scratch.tar.gz
```

- Read chapter 2 from the **xv6 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** book to understand the memory management layout of xv6.

Implementing Copy-on-Write Fork

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. You will implement copy-on-write (CoW) fork in the following steps.

Part A: Adding system call

Add a system call `int getFreePagesCount(void)` to retrieve the total number of free pages in the system. This system call will help you see when pages are used, and help you in debugging the code.

Part B: Tracking page reference counts

Currently, xv6 does not allow physical page frames to be shared. The first step in copy-on-write support will be adding a reference count to each physical page descriptor. First, you will need to understand how physical pages are allocated. Begin by reading `kernel/kalloc.c`. Here, each 4KB page of free physical memory is represented as a `struct run`, and these structures are organized into a free list.

- You need to track the reference counts of memory pages to implement CoW fork correctly. The reference count of a page indicates the number of processes that map the page to their virtual addresses. The reference count of a page is set to one when a free page is allocated to be used by some process. When an additional process points to an already existing page (Eg: when a parent forks a child and both share the same page), the reference count must be incremented. The reference count is decremented when a process no longer points to the page from its page table,

say, after acquiring its own copy of the page. A page can be freed and returned to the freelist only when no other process is pointing to it.
- Make changes to `kernel/kalloc.c`. Add a reference count to the page descriptor structure. You should set the count to one when a page is allocated, write a helper function to increment and decrement the count (using appropriate locks), and assert that the count is one when a page is freed (Hint: Methods `kalloc()` allocates and `kfree()` frees physical pages). Checking that a page is not in use by more than one process at the time of freeing will help you find bugs later.

Part C: Copy-on-write

Begin by reading and understanding the default fork() implementation. The system call is defined in `proc.c`, although the main workhorse is the function `copyuvm()`, defined in `vm.c`.

Part 1

You will implement a variant of `copyuvm()` called `cowuvm()` in `kernel/vm.c` that does the following:

- It should not allocate new page frames using `kalloc()` for the process. This should be done lazily in the page fault handler. This is a place where you may want to call `kalloc.c` to increment the reference count of the kernel pages.
- It should convert each writeable page table entry in the parent and child to read-only. [Hint: How would you know if a page is writable or not? Check the flags in `mmu.h` (e.g., PTE_W)]. The page tables of both, parent and child should point to the same physical pages, and these must be marked read-only. Make sure to flush the TLB entries as the page permissions have changed. Refer to the Notes section to understand how you can achieve this.

Part 2

After you have the above changes in place, the parent and child will execute over the same read-only memory. If either of the process (parent or child) tried to invoke a write operation to the page marked read-only, it will result in a page fault. The current trap handling code in xv6 does not handle `T_PGFLT`, you will need to implement this in `kernel/trap.c`. You will need to implement a CoW page fault handler that can be written in `kernel/vm.c` and should do the following:

- Note that the CR2 register holds the faulting virtual address during a page fault. You can get this address by calling the xv6 function `rcr2()`.
- Based on what the virtual address is, you need to decide how you want to handle the page fault.
- If this address is not mapped in the page table of the process, i.e, it is an illegal address, print an error message: `CoW: Invalid virtual address`, and kill the process.
- If the trap was generated due to CoW pages that were marked read-only, you should proceed by making copies of the pages as needed.
- If the page has more than one reference, copy the page and replace it with a writeable copy in the local process. Be sure to invalidate the TLB! Decrement the reference count on the original page.

- If the page has only one reference, you need to restore write permission to the page. No need to copy, as the other process has already copied the page. Be sure to invalidate the TLB!

Testing

You should be able to pass the default `forktest` provided in xv6 after finishing the implementation. In addition to this, we have provided a few tests under `~cs537-1/tests/P5`.

Notes

Modifications for tracking free physical pages and reference counts

We have added the required variables in `kernel/kalloc.c` to keep track of the free pages in the freelist, and the reference count for each physical page. You will be using these variables in your implementation.

```
struct{
  struct spinlock lock;
  struct run *freelist;
  unit  free_pages; //track free pages
  uint ref_cnt[PHYSTOP / PGSIZE]; //track reference count

}kmem;
```

Flushing TLB entries

The CR3 register holds a pointer to the top-level page directory, and entries from this page table are cached in the hardware-managed TLB. The OS has no control over the TLB; it can only build the page table. Whenever any changes are made to the page table, the TLB entries may not be valid anymore. So, whenever you make any changes to the page table of a process, you must re-install that page table by writing its address into CR3, using the following function provided by xv6.

`lcr3(PADDR(pgdir));`