Computer Science 432
Spring 2022
Lab 5: Copy-on-Write Memory Optimization, due before break.

Objective. To reduce memory pressure by sharing pages whenever possible.

Discussion. When processes are created they get access to a dedicated virtual address space. This *isolation* of memory supports:

- State independence. Processes must compute as though they had access to a machine dedicated to just their work.
- Controlled concurrency. We want processes to be able to coordinate concurrent activities. We must be able to treat processes differently in critical sections where several process states are coordinated through a shared memory.
- Basic security. We do not want the sensitive information that is often stored in memory to escape the process' sandbox.

Though processes imagine they make use of dedicated virtual memory, there are good reasons to actually *share* memory among processes, even if they're not aware their memory is shared:

- Read-only features of a process—like common library code—can be freely shared without concern. Code sharing can considerably reduce the *memory pressure* associated with running many related processes concurrently.
- Writable features of related processes that are initialized once and read many times. Memory that is dynamically allocated, for example, is often developed as the problem unfolds, and then is relatively static as the problem is solved by teams of child processes.
- Spatial locality suggests that there may be some process startup improvements if we delay cloning memory that has write access until the last possible moment.

We have seen, in our readings and lectures, that there are several optimizations that leverage the flexibility of per-process virtual address translation. Among these is the potential of *copy-on-write* (COW) sharing of memory between processes and their offspring. The goal of this optimization is to share memory between processes until the last possible moment—when one of the processes attempts to write to a page they all share. Only at that moment is the additional physical memory allocated to allow divergence of state.

As was discussed in Chapter 4, copy-on-write requires some very careful bookkeeping in order to maintain the facade of a process-dedicated physical store. This is the subject of the work we will be doing in lab before Spring Break.

The Assignment. Implement Copy-on-Write (COW) optimization for `xv6`. The basis for this lab is found in the `lab5` repository (use your user id rather than `22xyz`):

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab5.git
```

Again, this is a traditional `xv6` repository with very few extensions. If you have questions about last week's work on page tables, make sure we talk those through before you attempt this more completed work, here.

I assume you're up to speed on the readings, through Chapter 4. Given that, I believe the following is a workflow that will get you through the the COW optimization:

1. Before we begin any substantial modification of the kernel, it is important that we be able to identify when a process is attempting to write to a read-only page. (We will use this behavior to trigger the construction of a writable copy.) In `usertrap` (`kernel/trap.c`), identify where page faults could be recognized. Specialize the development of `usertrap` error messages to report read-based page faults, write-based page faults, and execution-based page faults. To indicate a process must be killed, use the code:

   ```
   p->killed = 1;  // this was a fatal error
   ```

   Notice these messages are *not* kernel panics. They're simply recognition of bad behavior in userland.

   Verify your work by attempting to write to a page you know the user cannot write. (Hint: `TRAMPOLINE` in `kernel/memlayout.h`, which is based on `MAXVA` in `kernel/riscv.h` is not readable, writable, or executable by the user.)

2. Encapsulate your code that manages all write-based page faults in a dedicated routine or *handler*. I suggest declaring this function in the following manner:

```
uint64
COWhandler(pagetable_t pagetable, uint64 va);
```

This function should return 0 if the fault could be fixed and -1 otherwise. At this point, of course, it's difficult to return 0. Think carefully about which kernel file to put this in, or be prepared to move it, later. While there is no real modularity in the kernel, there are better and worse places to locate your code.

Within your handler, distinguish between:

- Write-faults that are not valid virtual address mappings. These cannot be fixed and should fail.
- Write-faults accessing valid addresses that do not have appropriate `PTE_W` settings. Eventually, some of these faults will represent a demand to build a writable copy and retry the original instruction.

For those that have valid mappings, print the associated page table entry with `vmprintpte`, a utility function in `kernel/vm.c`. This will allow you to verify that your handler is getting used in the right situations.

Verify your implementation. For example, what happens when you try to write at an address above `MAXVA`?

Add and commit your work to this point.

3. Now, implement reference counting for each page in physical memory. Here is an approach:

   (a) Declare (do not use `kalloc`) an appropriately sized array whose entries are sufficiently large for our purpose. How many physical pages are *actually* in the machine? (Hint: see `kinit`, which builds the `kalloc` free list.) How many agents *could* share a single page? This array should be initialized in `kinit` in `kernel/kalloc.c`.

   (b) To support modularity, you may find it useful to write a helper function to, given a physical address, return a pointer to the reference counter associated with the page. You should make sure you return a null (0) pointer when the physical address is not meaningful. Place this function in `kalloc` and declare it in `defs.h`.

   (c) Modify the `vmprintpte` method so that it now includes a count of references to the page that contains the physical address mapped by the page table entry. This may be useful in debugging the handling of COW pages.

   (d) The `kalloc` (`kernel/kalloc.c`) routine allocates (*ie.* hands over for referencing) pages from a pool that are currently free. A page is free because no one is referring to it. Modify `kalloc` to set the page's reference count appropriately.

   (e) Write a new routine, `kshare` (see prototype in `kernel/kalloc.c`), that takes a physical page address and increases the reference count associated with that page, recording the fact a new process is now referring to an old page. We would use `kshare` in situations where we might otherwise use `kalloc` to reduce a process' memory footprint.

   (f) The `kfree` routine frees one of the references to a page allocated by `kalloc`. Modify this to reduce the reference count. Only when the page no longer appears to be referenced should it actually return the page to the free list.

   With this behavior, `kfree` can be thought of as undoing the work of *either* `kalloc` or `kshare`.

   At this point, the reference counting should work correctly for the current use case: `kalloc` and `kfree` work in opposite ways and we never actually call `kshare`. Compile `xv6` and verify it is still working as expected. Notice that any write-fault printing of PTEs currently have a reference count of 1.

   Add and commit your work to this point.

4. Now, we'll start to make substantial changes, writing a new version of the `fork` routine that allows parent and child processes to translate virtual addresses to shared physical pages. (The page tables, themselves, should not be shared.)

   While it is convenient to think of our work as a *modification* of `fork`, I suggest you duplicate the `fork` method, naming the unmodified code `forkOrig`. This will allow you to recall how the unshared version of the page table was constructed.

Now, carefully modify `fork` so that the child page table is shared with the parent:

(a) Write a helper method, `uvmshare`, in `kernel/vm.c`.

(b) For non-writable pages, we should simply share them by including a second reference to the parent's page. Copy the metadata from the parent PTE to the child PTE.

(c) Add a definition for `PTE_COW` in `kernel/riscv.h`. This should make use of one of the two bits (8 or 9) reserved for use by the operating system. This bit logically means "this page is temporarily write-protected."

(d) For writable pages, after sharing the page, you should clear the `PTE_W` and set the `PTE_COW` bit in the metadata for *both the parent and child pages*.

Notice that this page now becomes a read-only page. Convince yourself that this approach is consistent.

5. Once you've modified `fork`, *anyone* attempting to write will cause errors. Try booting the operating system and observe how the system fails. How did you expect it to fail? Did it fail in the expected manner?

6. Our task is to carefully implement, in the `COWhandler`, the "copy" part of the copy-on-write optimization. Here are some hints and observations:

(a) When a process attempts to write to a shared page it will fault and call your write-fault handler. The path to your handler will not have changed the exception program counter (`epc`), so if you were to return the value 0, the instruction would re-execute. This can be exciting.

(b) When you attempt to write to a page that is protected against writes, but *not* a COW page, it remains reasonable to flag an error. Just because the user wants to write to a page that is read-only does not mean they should be able to. Don't spoil your children by letting them do bad things.

(c) When you get a write-fault and it is a COW page, our goal should be to re-execute the instruction, but with a new, unshared, writable copy of the currently shared COW page.

(d) The treatment of the COW page is agnostic of whether a process was a parent or a child when the page was first shared. Either the parent or child or both may eventually want to write to this page.

(e) When a write-fault happens for a page that is shared, a new page should be allocated, the contents copied over. (Hint: `uvmunmap`, `mappages`, and `memmove`. The model for this code is found in the `forkOrig` code, above.) The metadata on the new page should revert to its pre-shared state. Since this page is no longer a reference to a shared page, the reference count should be decreased. How?

(f) When a write-fault happens for a page that is "shared" but has reference count 1, *do not create a new copy of the page*. Use the original. Make sure, of course, that you modify the metadata appropriately.

7. There is one final part of the kernel that must be made aware of the copy-on-write process: `copyout` (in `kernel/vm.c`). This routine, recall, is responsible for reading data from a kernel address and writing it to a user address.

Here's an approach: Before `copyout` attempts to write to a page, it should check to see if the `PTE_COW` bit is set. If it is, it should call your code to handle a write-fault. Because you are calling this from the kernel, you must be very careful to pass it the appropriate page table. This is why we wrote the COW handler to accept a pagetable as well as the virtual address associated with the fault.

8. At this point, you should be able to pass all the tests in the `cowtest` testing facility from MIT.

9. Finally, add, commit, and push your code for review and grading.

Thought Questions. Please think about the following questions in preparation for meeting in small groups:

1. Page tables, of course, are allocated by `kalloc`. Will they end up being shared, or not? (Hint: these are structures only ever written by the *kernel*.)

2. It seems that, through `fork`, we're always constructing new page tables by sharing their target pages with some other process. Where are pages originally constructed?

3. How does the `cowtest` application work?