CS333: Operating Systems Lab

Lab 6: Copy-on-Write Fork in xv6

Goal

The goal of this lab is to understand memory management in operating systems. We will achieve this goal by implementing the copy-on-write fork feature in xv6. This implementation will require you to thoroughly understand how memory management and paging work in xv6.

Implementing Copy-on-Write Fork

The current implementation of the fork system call in xv6 makes a complete copy of the parent's memory image for the child. On the other hand, a copy-on-write fork will let both parent and child use the same memory image initially, and make a copy only when either of them wants to modify any page of the memory image. We will implement copy-on-write (CoW) fork in the following steps.

1. Add a system call `getNumFreePages()` to retrieve the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation.

2. Understand the following subtle point related to page tables in x86. 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(v2p(pgdir));
   ```

   This operation ensures that the older TLB entries are flushed as well. Note that xv6 already does this TLB flush when switching context and address spaces, but you may have to do it additionally in your code when you modify any page table entries as part of your CoW implementation.

3. Add a kernel data structure that keeps track of the reference count of pages, and funtions to increment and decrement these counts. The reference count of a page is set to one when it is allocated, and is incremented every time a new child points to the same page. The reference count is decremented when a process no longer points to it, say, after acquiring its own copy of the page. A page can be freed and returned to the free pool only when no other process is pointing to it. Carefully think about where this count is incremented and decremented, and make sure you do these changes to the count with proper locks held.

4. You must change the `copyuvm` function called from fork to implement CoW. When you fork a child, the page tables of the parent and the child should point to the same physical pages, and these pages must be marked read-only. Given that the parent's page table has changed (with respect to page permissions), you must reinstall the page table and flush TLB entries, as described above. This function is one place where you may have to increment reference counts of kernel pages.

5. When the parent or child processes attempt to write to a page marked read-only, a page fault occurs. The trap handling code in xv6 does not currently handle the `T_PGFLT` exception (that is defined already, but not caught). Add a trap handler to handle page faults. You can simply print an error message initially, but eventually this trap handling code will call the function that makes a copy of user memory.

6. The bulk of your changes will be in this new function you will write to handle page faults. When a page fault occurs, the CR2 register holds the faulting virtual address, which you can get using the xv6 function call `rcr2()`. You must now look at this virtual address and decide what must be done about it. If this address is in an illegal range of virtual addresses that are not mapped in the page table of the process, you must print an error message and kill the process. Otherwise, if this trap was generated due to the CoW pages that were marked as read-only, you must proceed to make copies of the pages as needed.

   Note that between the parent and the child, the first one that tries to write to a page should get a new memory page allocated to it. This new page's content must be copied from the contents of the original page pointed to by the virtual address. Even after this copy is made, note that the page is still marked as read only in the page table of the second process, and it will soon trap as well when it attempts to write to the read-only page. When the second process traps, no new pages need to be allocated; it suffices to remove the read-only restriction on the trapping page, since the first process already has its copy. Your page fault handling code should distinguish between these two cases using the reference count variable, and handle them suitably. Make sure you modify the reference counts correctly, and remember to flush the TLB whenever you change page table entries.

7. Finally, you must write a test program `testcow` that tests the copy-on-write implementation. You must fork a process, print some information to show how the parent and child are sharing the memory image, modify the memory (say, by changing one of the variables in the program), print out when the copy happens, and so on. Use the new system call to print out the number of free pages in your system at various points. If your copy-on-write implementation is done correctly, you will see that free pages are reduced when one of the processes writes to memory. Design your test program in such a way that running it and inspecting the output will clearly show all the steps in the CoW implementation. Test with more than one child processes as well for correctness.

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf` should contain a brief description of your implementation (which files were modified, and how), any new data structures used, and how you tested your implementation.

- A patch of your code. Your patch must include all modifications to the source code as well as to the Makefile. Do not forget to include your test program.

Evaluation of this lab will be as follows.

- We will read through your report to understand your CoW implementation.
- We will patch your code onto our codebase and run your test program. Please make sure that the output of your test program is clean enough to understand the results of the tests.
- We will read your code to understand what you have done. Please make sure your code is well-documented and readable.