CSCI350 Project 4

Due: October 30th by 11:59pm

Objectives

The goal of this assignment is to thoroughly understand how memory management and paging works in operating systems. We will achieve this goal by implementing Copy-on-Write (CoW) fork feature in xv6. Chapter 2 of the xv6 book describes how xv6 manages its page table structures.

Task 1 Enhanced process details viewer

Prior to updating the memory system of xv6, you will develop a tool to assist in testing the current memory usage of the processes in the system. To do so, enhance the capability of the xv6's process details viewer. Start by running xv6 and then press ***ctrl + P*** (^*P*). You should see a detailed account of the currently running processes. Try running any program (e.g., usertests) and press ^*P* again (during and after its execution). The ^*P* command provides important information on each of the processes. However, it reveals no information regarding the current memory state of each process. Add the required changes to the function handling ^P (see `procdump()` in `proc.c`) so that the user space memory state of all the processes in the system will also be printed as written below. The memory state includes the mapping from virtual pages to physical pages (for all the used pages) and the value of the writable flag.

For each process, the following should be displayed (words in italics should be replaced by the appropriate values):

Process information currently printed with ^P
Page mappings:

virtual page number -> physical page number, writable?
...
virtual page number -> physical page number, writable?

For example, in a system with 2 processes, the information should be displayed as follows:

```
1 sleep init 80104907 80104647 8010600a 80105216 801063d9 801061dd
1 -> 300, y
200 -> 500, n
2 sleep sh 80104907 80100966 80101d9e 8010113d 80105425 80105216 801063d9
1 -> 306, y
200 -> 500, n
```

Showing that the first process (`init`) has 2 pages. Virtual page number 1 is mapped to physical page number 300 and its writable bit is set. The virtual page number 200 is mapped to physical page number 500 and its writable bit is not set. The second process (`sh`) also has 2 pages. Virtual page number 1 is mapped to physical page number 306 and its writable bit is set. The virtual page number 200 is mapped to physical page number 500 and its writable bit is not set.
Note that:

- The values above are made up.
- Don't print anything for pages or page tables that are currently unused.
- Only print pages that are user pages (`PTE_U`).
- The information obtained from ^P will be used for the grading.

Task 2 Copy on Write (COW)

For this project you should use the initial version of xv6 (the one you used in project 1). Most changes will be in vm.c.

Upon execution of a fork command, a process is duplicated together with its memory. The xV6 implementation of fork entails copying all the memory pages to the child process, a scheme which may require a long time to complete due to the many memory accesses needed. In a normal program, many memory pages will have no change at all following fork (e.g., the text segment), furthermore, many times fork is followed by a call to exec, in which copying the memory becomes redundant. In order to account for these cases, modern operating systems employ a scheme of copy-on-write, in which a memory page is copied only when it needs to be modified. This way, the child and parent processes share equal memory pages and unneeded memory accesses are prevented.

In this task you will implement the COW optimization in xV6. To accomplish this task, change the behavior of the fork system call so that it will not copy memory pages, and the virtual memory of the parent and child processes will point to the same physical pages. Note that it is OK to copy page tables of a parent process (though you may also share/cow the page tables if you would like to).

In order to prevent a change to a shared page (and be able to copy it), render each shared page as read-only for both the parent and the child. Now, when a process would try and write to the page, a page fault will be raised and then the page should be copied to a new place (and the previous page should become writeable once again).

In order to be able to distinguish between a shared read-only page and a non-shared read-only page, add another flag to each virtual page to mark it as a shared page. Notice that a process may have many children, which in turn, also have many children, thus a page may be shared by many processes. For this reason, it becomes difficult to decide when a page should become writeable. To facilitate such a decision, add a counter for each physical page to keep track of the number of processes sharing it (it is up to you to decide where to keep these counters and how to handle them properly). Since a limit of 64 processes is defined in the system, a counter of 1 byte per page should suffice. Since the wait system call deallocates all of the process's user space memory, it requires additional attention – remember do not deallocate the shared pages.

Note that:

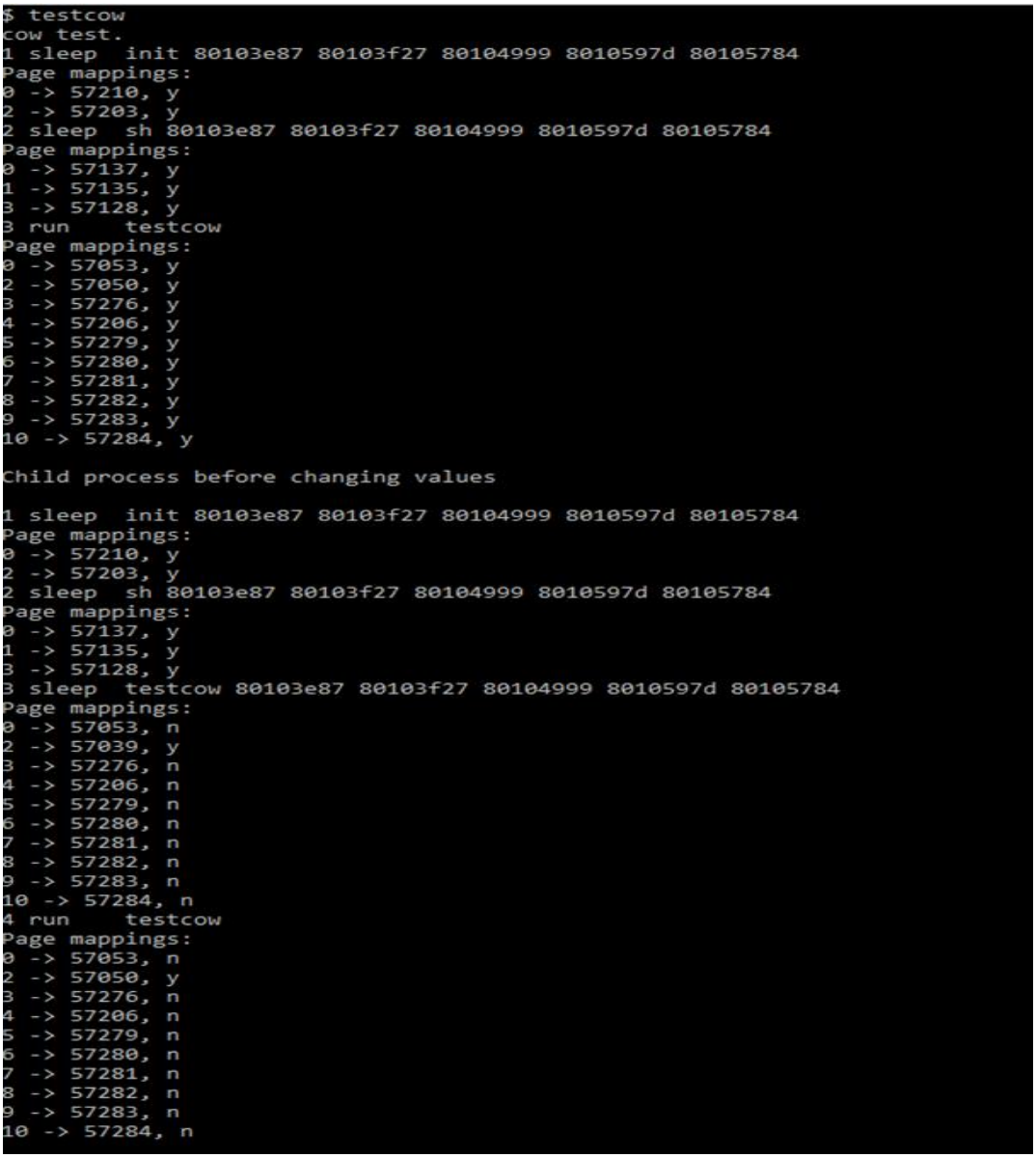
- When a page fault occurs, the faulty address is written to the **`cr2`** register.
- The **`CR3`** register holds a pointer to the top-level page directory, and entries from this page table are cached in the hardware managed TLB. The OS has no control over the TLB; it can only build the page table. 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.

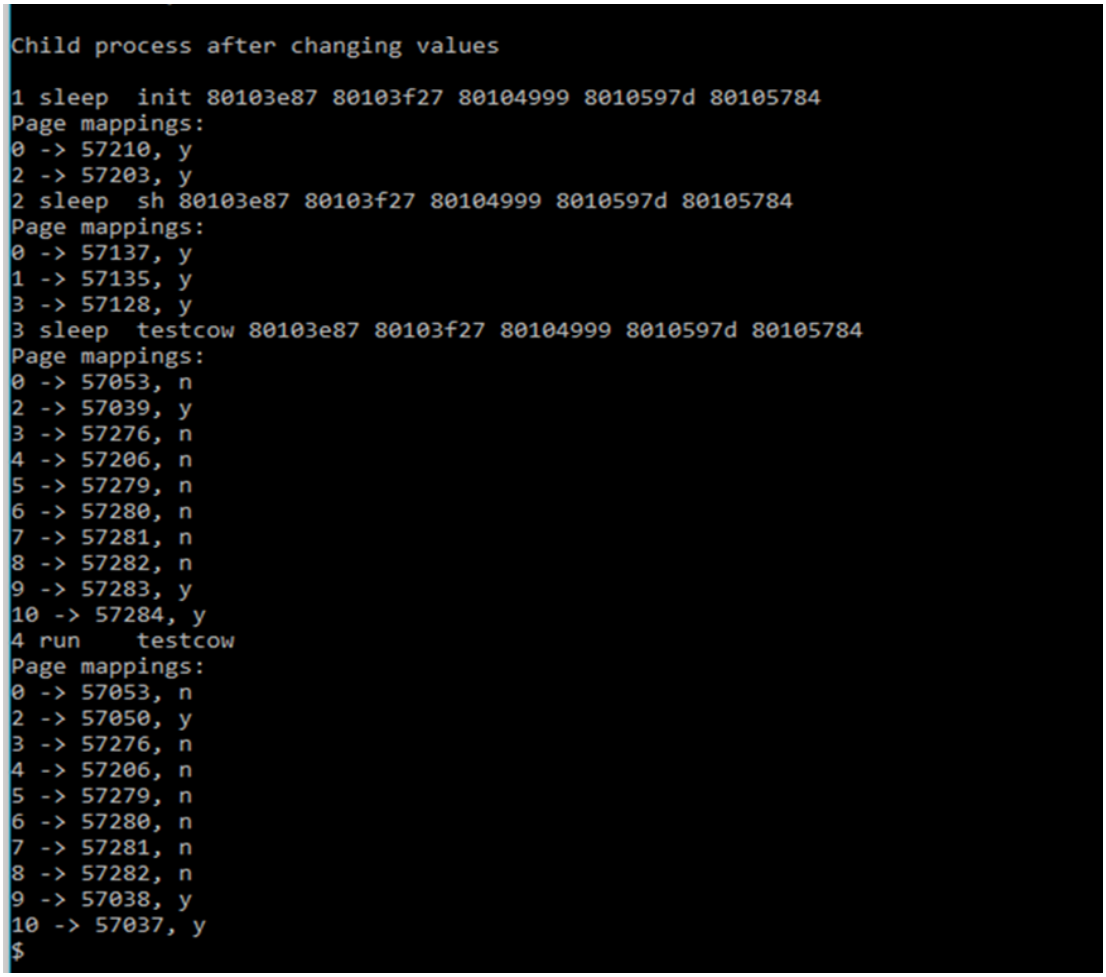
```
lcr3(V2P(pgdir));
```

This operation ensures that the old 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.

Task 3 Testing

To make sure that COW optimization implemented properly write a simple user program and name it `testcow.c`. In this program use `malloc` to create new values that sit in **different** pages (in the heap). Use `fork` system call and make sure that the user space pages are shared. Now update a variable in the child process. Was the proper page copied? Use your updated `procdump()` to print out the result. Sample output:





Grading and Submission

The total number of the points for this project is 100.

Your report (see SubmissionIstructions document in Resources for details) is worth 10 points.
DESIGNDOC should have .txt or .pdf extension and no other extension will be accepted.

- Describe your changes to the xv6 code
  - Name the files your modified and clearly describe the changes you made as a part of Task 2 to extend xv6 memory management with CoW.
- Your code must match the general code style of xv6

Correct extension of `procdump.c()` 5 points

Free Response Questions: 15 points

Correct implementation of the CoW functionality: 60 points

Correct implementation of the `testcow.c`: 10 points. No points if the test does not compile.