Advanced Memory Management for xv6

xv6 implements a very basic memory management system. In this assignment, you will enhance it to a great extent; namely, you will add COW and Paging.

Important Notes:
This document is complete. Unless necessary, no further update will be made.
Last Modified: 12:40AM, February 15, 2023

Important Instructions:

- ***Do not copy. Any proof of copy will result in -100%.***
- *Don't start implementation right away. First understand what actually happens in xv6. During implementation, try to test after each small change and make sure everything runs as expected (maybe it is supposed to panic).*
- *If you cannot find where a kernel function is implemented, check out* `kernel/defs.h`*.*
- *Keep the testing codes. You will be marked based on them too. All red colored texts after this ask for such code.*
- *Implement COW and Paging independently first. Keep their patch files. Submit these two patch files if you cannot do a combined implementation. Otherwise submit one patch file for the combined implementation.*
- *Put the patch files in a folder named by your student id (ex: 1805123), zip it and submit. Make sure all the necessary codes are in the patch files so that it does not have any external dependency.*
- ***Submission Deadline: 11:59PM, February 27, 2023***

COW

COW or Copy-On-Write is an improvement on regular memory allocation of `fork()` system-call. In xv6, this system-call directly copies all the *user-space memory* of the parent process to the child. Now, if a page is only ever used to read (or execute), then it creates unnecessary copy. COW tries to minimize this by deferring the copy until a write operation is done on a page. It sets up the pagetable in the child process such that all the PTEs for *user memory* point to the parent's physical pages and clears the *write flag*s of all PTEs for *user memory* in both child and parent. When a write operation is done on any non-writable page, the CPU forces a page fault. Kernel keeps track of which pages are COW and when these faults occur for a COW page, it allocates a new page for the faulting process, copies the data from the COW page to it and updates the corresponding PTE such that it points to this new page with the *write flag* set. Now, the write operation can carry on.

Follow these steps:

1. Find out and understand the function that creates pagetable of the child process during `fork()` system-call. [Hint: it is `uvmcopy()` in `kernel/vm.c`.]

2. You need to handle the page fault that occurs while writing in a COW page. When any type of trap occurs from user code (page fault belongs to this type), it calls `usertrap()` in `kernel/trap.c`. Understand how the trap for system-call is handled in this code. Also, check what happens for any other type of trap. [Don’t get bothered with `which_dev`. `devintr()` returns `0` when the CPU itself generates the trap (like system-call, page fault).]
3. Update the function you found in step 1 so that no new page is allocated, and the PTEs in the child process’ pagetable points to the parent's physical pages. Also, make sure that the *write flag* is clear for both parent and child processes. [Many macros and definitions related to page tables can be found at the end of `kernel/riscv.h`]

This is a good place to test your implementation. Write a user code to check if `fork()` system-call works and creates page faults if a write operation is done.

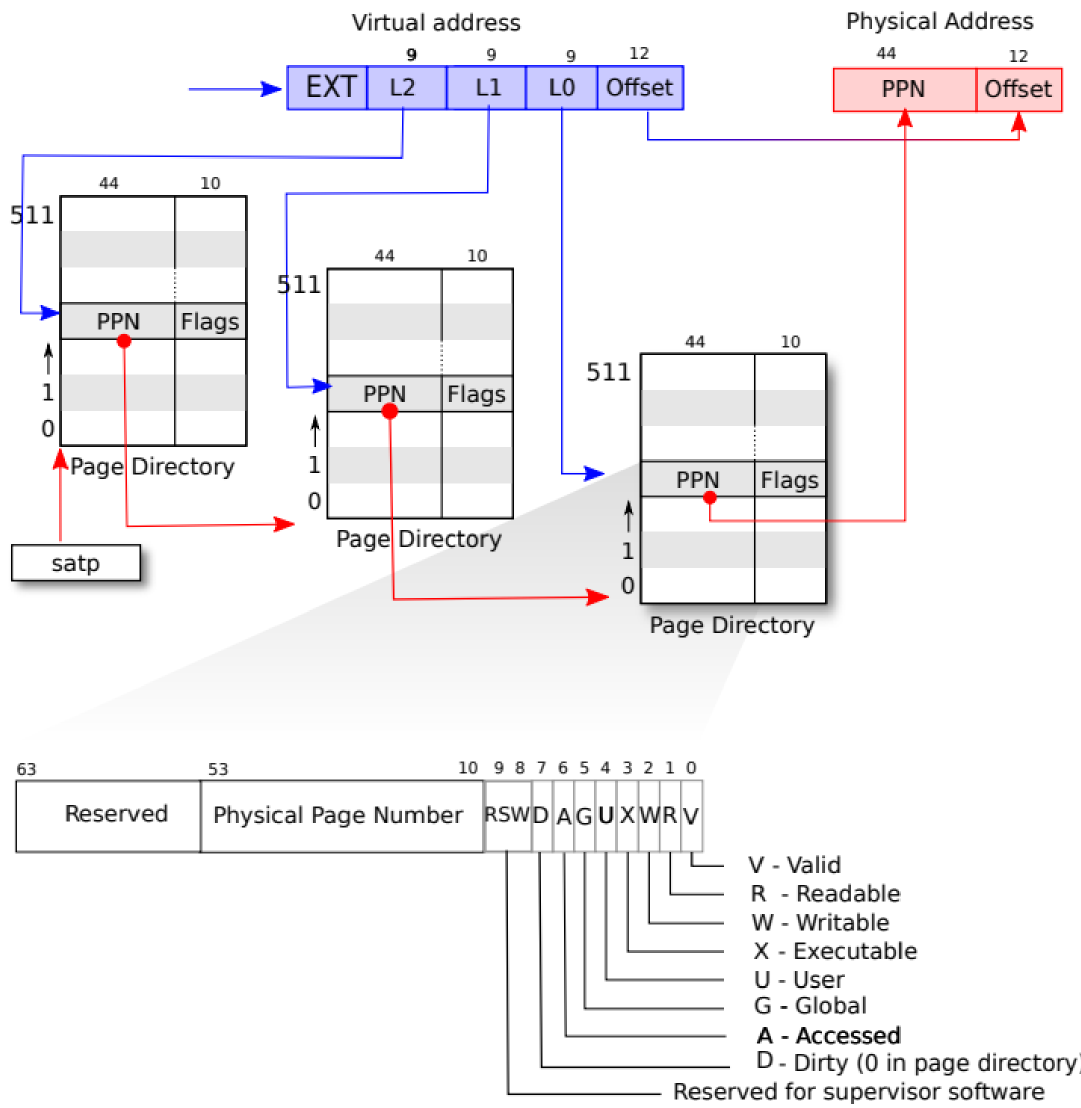

Figure 3.2: RISC-V page table hardware.

4. Update the `usertrap()` function in `kernel/trap.c` so that write operation on a COW page creates a writable copied page for the faulting process. So, you need to keep track of if a page is COW. PTEs in riscv have 10 flags of which 8 are used by hardware as in Figure 3.2 and 2 other reserved for software (RSW) (although xv6 provides definition for first 4 bits as given in `kernel/riscv.h`.). You can use any of the RSW bits. [Original `uvmcopy()` you modified already implements this copying process (albeit for all pages). Take inspiration from it.]

This is a good place to test your implementation. Write a user code to check if `fork()` system-call works and handles a write operation correctly.

5. You may think it is done. Unfortunately, no! If you have carefully followed the process, you can see once a page is set non-writable, it is never set writable. Now, if both parent and child write on the same COW page, the non-writable COW page is lost (as both parent and child created its copy), i.e., memory leak by kernel! Also, a COW page can be referenced by many processes (depending on number of forks). The only way to solve it is to keep a “reference count” for each page so that a “kalloc”ed page can be garbage collected. You can choose various schemes to keep the reference counts for all pages. You may use a fixed size array where you can index by the page's physical address divided by page size.
   This is a good place to test your implementation. Write a user code to check if `fork()` system-call works and a COW page is freed when both parent and child process writes to it. Write a system call that prints the statistics for page usage: the number of pages being used by each running process and the number of pages in freelist.
6. Now, we should be done. Or are we? Unfortunately, there is still a problem. Up to this point, we assumed that the write operation to a COW page is only done in user mode. But it is possible that a write to a COW page is done in kernel mode (for example, `copyout()` in `kernel/vm.c`). A page fault in kernel mode generates a kernel trap which is handled by `kerneltrap()` in `kernel/trap.c` as opposed to `usertrap()`. You can solve this issue by changing `copyout()` and any other functions like it, or by changing `kerneltrap()`.
   This is a good place to test your implementation. Write a user code to check if `fork()` system-call works and `copyout()` works correctly in both parent and child processes. [Hint: `copyout()` gets called for pipe and file I/O. You can use any one of these or write a new system-call that uses this.]

Paging

As the memory is normally very small compared to the need for a process. Paging operation puts some pages to persistent storage (disk) when memory gets full and brings pages from the swapped file to memory when it is used. It makes available virtual memory much larger than physical memory. Now which page gets swapped depends on the page replacement algorithm being used.

Follow these steps:

1. Download and apply the following patch file.
   [swap.patch]()
   This patch file will help you swap out and in pages to and from disk respectively. Understand how it does its job. Update any other code that this code depends on to work.
   It contains a struct named `swap`. This structure contains the metadata to retrieve a page that has been swapped out to disk. When a page is *swapped out*, it saves the block no. to the blocks that store that page. A swapped out page can be *swapped in* using the swap struct that was created when that page was swapped out.
   It implements the following functions:
   - `void swapinit(void)` : Initializes necessary variables for allocating and freeing swap structs.
   - `struct swap* swapalloc(void)` : Allocates a swap struct and returns a pointer to that struct.
   - `void swapfree(struct swap*)` : Frees a given swap struct that can be reused during some future `swapalloc()`.
   - `void swapout(struct swap *dst_sp, char *src_pa)` : Writes the page `src_pa` to disk and saves the block no.s to `dst_sp`.
   - `void swapin(char *dst_pa, struct swap *src_sp)` : Reads a page into `dst_pa` that has been previously swapped out (by calling `swapout`()) with `src_sp` as argument.

   You will use these functions to do the swap operations.
   - **Swap out**: Allocate a swap struct using `swapalloc`. Then use `swapout` to write the page to disk. The swap struct should be saved somewhere so that you can swap in this page when needed.
   - **Swap in**: First find the swap struct that was created when the page was swapped out. Then use `swapin` with this struct to copy the data from disk.

   Note: `swapout` *and* `swapin` *does neither allocate or deallocate anything.*
2. You need to keep all the live pages in a data structure so that you can decide which page to swap out. A simple array is good enough for this lab. However, you can use any other data structure (like linkedlist). Implement the necessary datastructure and check if you can keep track of all pages being used across different processes.

This is a good place to test your implementation. Write a user code that uses some number of pages provided by the command line. Make sure the user code takes some time to execute. Write a system-call that prints the number of **live** pages being used by different processes. Now run multiple instances of the user code from shell (you know this from first xv6 offline) and use the system-call to check if the counts match.

3. At any time, you should have no more than MAXPHYPAGES (=50) pages being used by all user processes in total. Implement a simple page replacement algorithm (like FIFO). You don't need to care about COW in this step. Make sure you correctly free the swap structs and physical pages.
   This is an important place to test your implementation. You can run the test code that you implemented in step 2. You should test after each small update of the code and check if it works as expected (for example, first check if you can swap out pages and reduce the number of live pages; then check if you can swap in a page when required.).
4. Your simple implementation is not likely to work in the presence of fork() (even without COW). You need to keep the number of references to each swap struct so that you can garbage collect. Update the swap struct accordingly.
   Write appropriate test codes here. The user code should include some number of forks. Make sure the parent process is big enough so that some page is already swapped out before fork is called, then the child process has to access a swapped out page from the parent process.
5. Now, update your codes so that it supports COW also.
   Make sure your test code covers all combinations of a page being COW or not and Swapped or not.

Bonus

Implement any other page replacement algorithm apart from FIFO (ex: NFU, NRU, Aging etc.). Bonus is proportional to the complexity of the algorithm.