Advanced Memory Management for xv6

xv6 implements a very basic memory management system. In this assignment, you will enhance it by adding Paging.

Important Notes:
Last Modified: 1:00 PM, August 28, 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.*
- *Rename the patch file by your student id (ex: 1805123.patch) and submit. Make sure all the necessary codes are in the patch file so that it does not have any external dependency.*
- ***Submission Deadline: 11:55 PM, September 11, 2023***

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` *do not 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 data structure 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. Swap out one of the live pages to disk (replace the page that implements **FIFO**). Make sure you correctly free the physical page afterwards. You need to keep track of if a page is present in physical memory. 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 uses the first 4 bits as given in `kernel/riscv.h`). You can use any of the RSW bits. You also need to keep the information of the *swap struct* corresponding to the swapped-out page. PTE itself has enough space to keep a pointer to swap struct.

4. Now, xv6 should panic when a swapped out page is accessed. This panic is generated due to a usertrap that corresponds to page fault. You need to handle the page fault. 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()) != 0`]. Update the `usertrap()` function so that the swapped-out page is properly swapped-in to a physical page and correctly mapped in the pagetable. [Note: Steps 3 and 4 are not as straightforward as you may think. You need to change in many places in `kernel/proc.c` so that any locks being held by the process are unlocked when `swapout` and `swapin` are performed.]

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, and get panic when a swapped-out page is accessed; then check if you can swap in a page when required.).

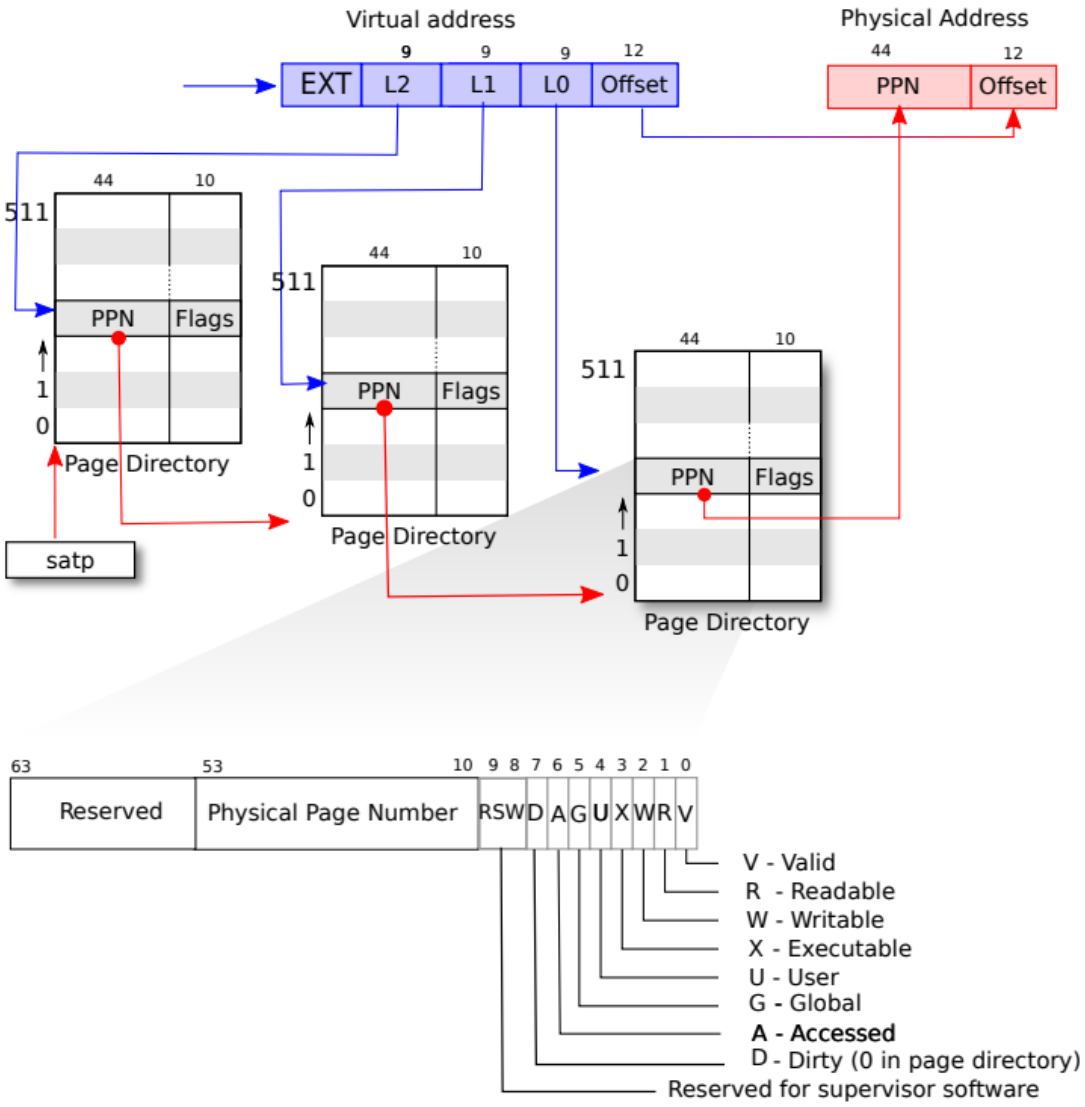

Figure 3.2: RISC-V page table hardware.

5. Your simple implementation is not likely to work in the presence of `fork()`. If `fork()` is called after some pages have been swapped out, then the child process won't be able to access the swapped-out pages. Change `uvmcopy()` to solve this issue.
***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.***

6. Your implementation up to this leaks memory and disk space. The *swap struct* s created by `swapalloc` has never been freed. You can try to free a swap struct right after its corresponding page is swapped-in. However, it won't work in the presence of `fork()`. If `fork()` is called after some pages have been swapped out and both parent and child processes try to access a swapped out page after that, then the corresponding swap struct will be used after being freed (and also will be freed twice). You need to keep the number of references to each swap struct so that you can garbage collect. Update the swap struct accordingly.
   Check if swap struct is freed correctly.

Bonus

1. Implement another page replacement algorithms apart from FIFO (ex: NFU, NRU, Aging etc.). Bonus is proportional to the complexity of the algorithm.
2. Do step 2 with a linked list.

Marks Distribution

Without test codes, the marks will be zero.

Task	**Marks**
Step 2 + test	20
Steps 3, 4 + test	40
Step 5 + test	20
Step 6 + test	20
Total	**100**
Bonus 1	40
Bonus 2	20