COL331 Project
Implementing Copy-On-Write (COW) in xv6

Harsh Vora (2021CS10548)
Rishabh Kumar (2021CS10103)
Vivek Gupta (2021CS10100)

1 Introducton

The goal of this project is to implement the copy-on-write (COW) mechanism in the xv6 operating system to improve memory utilization during the fork() system call. COW defers allocating and copying physical memory pages for the child process until the copies are actually needed, if ever. The goal is to implement COW paging, and make it consistent with the implementation of swap space.

2 Enabling Memory Page Sharing and Handling Writes on Shared Pages

2.1 vm.c: copyuvm()

Did not kalloc() a new memory page for the child, but initialised it's new page table entry with the address of the parent's corresponding page. Set the PTE_W bit of both child and parent's pages to false. Increased the reference count of the parent's page.

2.2 trap.c: page fault handler

If a page fault occurs, the page fault handler checks if the faulting page is accessed by multiple processes, from its PTE_W bit. If the reference count of the parent's page is 1, the page is marked as writeable and the page fault handler returns. Otherwise, a new page is allocated to the faulting process, the parent's page is copied to the new page, and the PTE is updated to point to the new page.

3 Implementing COW with Swapping

3.1 rmap data structure

A 2D array of size (NUM_PAGES + NUM_SWAP) x NPROC was created to store the struct proc* of the processes that have mapped that page. All entries are initialised to 0.
The rmap data structure is updated whenever a page is mapped or unmapped by a process.
Whenever a new process is mapped to the page, a 0 entry in the corresponding row of the 2D array is obtained and the proc pointer is stored at that index. To unmap a process, its pointer is located in the row corresponding to that page, and is replaced by a 0. The previously implemented refcount array is retained.
A spinlock was added to protect the rmap data structure.
The lock is acquired before updating the rmap data structure and released after updating it.

3.2 vm.c: copyuvm()

A separate function mappages_swap() is created to add a new pagetable entry to the child process for pages in the swap space, as the PTE_P bit is not set for these pages. The rmap data structure is updated whenever a page is mapped to a process.

3.3 vm.c: allocuvm() and inituvm()

The rmap data structure is updated whenever a page is allocated to a process. The rss value of the process is increased whenever a page is allocated to it.

3.4 vm.c: deallocuvm()

The rmap data structure is updated whenever a page is deallocated from a process. The rss value of the process is decreased whenever a page is deallocated from it.

3.5 kalloc.c: kalloc()

A new function swap_out() is created to swap out a page to the swap space.
A free swap_slot is found, free flag is set to 0, and the page is written to the swap space.
All processes mapping the page are obtained from the rmap data structure and their PTEs are updated to point to the swap space. The rss value of each process is decreased. Flags are stored in the swap slot's page perms.
The rmap data of the memory page is copied into the data of the swap_slot, and is then cleared.

3.6 Handling page faults - swap_in()

If a page fault occurs, the page fault handler checks if the faulting page is in the swap space.
The page is swapped in from the swap space, and the PTE is updated to point to the new page.
This is done using the swap_in() function. In the swap_in() function, the swap_slot is read into a new page allocated using kalloc().
For each process mapping the page in the swap space(obtained from the rmap data structure), the PTE is updated to point to the new page. The rss value of each process is increased.
The rmap data of the swap_slot is copied into the rmap data of the new page, and the rmap data of swap_slot is cleared.
The swap_slot is then marked free. The pagefault handler then returns.

If the page is a COW page, then the previous implementation is followed, with updates to the rmap data structure.

3.7 free_swap()

The free_swap() function is created to free swap slots if a process mapped to them exits.
The rmap data structure is updated to remove the process from the mapping of the page.
If the reference count of the page is 0, the swap slot is marked free.

3.8 Handling race conditions

The rmap data structure is protected by a spinlock to prevent multiple processes from updating it simultaneously.
A sleeplock is used to protect the complete code of the swap_in() and swap_out() functions.
It ensures that only one process can swap in or out a page at a time. Also, if swap_out() is called from the kalloc() call inside swap_in(), the holdinngsleep() function is used to prevent a deadlock due to acquiring the same sleeplock multiple times by the same process.
The swap space is protected by a spinlock to prevent multiple processes from accessing it simultaneously, and this is used mainly during freeing swap slots, as the sleeplock is already used to protect the swap space during swapping in and out.