COL331 Project Report

Kashish Goel(2021CS50602) Piyush Chauhan(2021CS11010)

Prakkhyaat Prajaapat (2021CS10914)

April 24, 2024

§1. Introduction

This project was to implement Copy on Write (CoW) with demand paging. This mechanism allow both parent and child processes to share memory pages initially without duplication. If either process writes to a shared page, the page is duplicated and modified. This technique not only conserves memory but also speeds up the process creation by delaying the need for memory copies.

§2. Part1 : Page Sharing

- To enable page sharing we modified copyuvm() function to remove `(mem = kalloc())` and `(memmove(mem, (char*)P2V(pa), PGSIZE))`, which created and copied the content existing pages into new pages. We are rather copying the `pte` entries of the parent process into the page table of the child process.
- We updated permission of parent page to be readable using `*pte = *pte & ~PTE_W` and `flags = flags & ~PTE_W`. Mappage commamd was modified to `mappages(d, (void*)i, PGSIZE, pa, flags)` to include `pa` of parent's page instead of address of previously allocated new page address.
- The `rmap` (discussed later) of every page in parent process was updated , `ref_count` was increased and `childProcID` was added in pages' `procRefList`. `rss` count for child proc was also increased. TLB was flushed using `lcr3(V2P(pgdir))` to update the changes in buffer.

§3. Part 2: Managing Shared Page Writing

- We created a new struct called `rmap` which had `uint ref_count` and `int procRefList[NPROC]`. `procRefList` is a fixed array of `NPROC` integers which store the process_id of processes which are referring that page. An array of `rmap`,named `rmap_list` of length `PHYSTOP >> PTXSHIFT` = 1024, was declared inside `kmem`, associating `rmap` with each page.
- The new Array was initialized in `kinit1` to default with zero `ref_count` and -1 array in `procRefList`. Appropriate APIs to get, increment, decrement `ref_count`, and update `procRefList` were declared to update `rmap` of a page. APIs were also implemeted to retrive index of a process in proc_table from pid stored in `procRefList`.
- `kfree()` was modified to include `process_pid` as a parameter.This was to update the `rmap` of the page which is being freed. We first decreased the `ref_count` and removed `process_pid` from `procRefList`. If `ref_count` became 0, then we freed the page and reinitialized its `rmap` to default.
- `ref_count` and `procRefList` were updated after `kalloc()` in `inituvm()`, where we are allocating the first page of process and in `allocuvm()`, where new pages were allocated when process grew.

- `inituvm()` , `freevm()` and `deallocuvm()` were modified to include `pid` of the calling process to pass into `kfree()` (as mentioned earlier).

§4. Part 3: Enabling Swapping

- To enable swapping we declared a `struct swap_slot` and `struct swap_block` in `bio.c`.
- The `struct swap_block` contains an array of `swap_slots`. `swap_struct` includes page_perm(issions), is_free bit , procRefListofPage , refCountPage and indexInRmap (index of the page in `rmap` structure). The swap_block was initialized at boot-time using `swapSpaceinit()` which initilized `is_free` to 1 and `page_perm` to 0.
  The offset for `sb.logstart`, `sb.inodestart` and `sb.bmapstart` were increased by `SWAPBLOCKS` amount to include the swapblock size.
- Several APIs were declared in `bio.c`, most relevant ones are as follows:
  - `updateSwapSlot(int diskblock, int isFree, int page_perm, int refCount, int* procreflist, int indexinrmap)` : to update the swap_slot present at the given `diskblock` with values given in parameters.
  - `void writeToDisk(uint dev, char* pg, int blockno)` : to write page in buffer into disk
  - `void readFromDiskWriteToMem(uint dev, char *pg, uint blockno)` : to read a page from disk and write it in memory
  - `void clear_swap_slot(pte_t* page, int pid)` : resets the `rmap` items if `refCount` is zero otherwise, decreases `refCount` by 1 and updates `procRefList` accordingly.
- A new file `copyonwrite.c` was created which included the following functions :
  - `void swapOut(void)` :
    It finds the victim process and page using policy mentioned in `lab4` .It then proceeds to locate an available swap slot and determine the corresponding disk block number. It write swapped out page in swapBlock, update the `*pte` to reflect swapped operation, and re initializes its `rmap`. Finally, it iterates over the `procRefList` of swapped page, decrementing reference counts and updating page table entries accordingly.
  - `void pgfault_handler(void)` :
    When a page fault occurs it checks for two possibilities
    * **if page is not present** `(!(*pte & PTE_P))` :
      It retrieves the page from disk and allocates memory for it using `kalloc()`. It iterates over the `procRefList` to increase rss, increase `refCount` and update `procRefList` as a page has been swapped in. In end, it marks the swapBlock free from where page was retrived.
    * **if page is not writable** `(!(pte & PTE_W))` :
      It checks the reference count of the page. If the reference count is one, indicating a single owner, it permits Write directly. Thereby, implementing the optimization asked in 2İ.
      Otherwise, it allocates a new page, update `rmap` , copies the content from the original, decreases `refCount` of parent process, removing child pid from parent swapped in page `procRefList`, updates the `pte` and flushes TLB to update the changes made.
  - `void swapIn(pte_t *pte)` :
    While copying the page_table in `copyuvm()`, it could be the case the parent pages are swapped and not present in memory. Earlier, a panic was raised but now we have to bring the page from swapBlock and then map it to child process.

    This function allocates a new page,Read data from swapped block and write to new page. It iterates over the `procList` of page and increases `rss`, `refCount` and updates `procRefList` to include the process_id of referenced process . It restores permission from swapBlock and updates pte accordingly.