COL331 Project

Rajat Golecha 2021CS10082
Mihir Kaskhedikar 2021CS10551
Akash Voora 2021CS10104

April 2024

A new file `pageswap.c` was added to the folder along with major changes in the 4 files `kalloc.c`, `vm.c`, `proc.c` and `bio.c`.

1 pageswap.c

The struct for swap space consisted of 4 attributes :

1. A boolean value for whether the slot is free.
2. A `uint` of the starting block number of the slot (a slot consists of 8 blocks)
3. An array `pte_array` of size `NPROC` of `pte*` datatype. This is used to store all the page table entry pointers of different processes that mapped to the physical page that was brought in the swap slot.
4. A number `refcnt` denoting the number of processes that pointed to the physical page before it was brought in this swap slot.

We defined a global array `rmap` of size `PHYSTOP >> PTXSHIFT` where `rmap[i]` denotes the refcnt of the ith physical page. We also defined a global 2D array `reverse_map` of size (`PHYSTOP >> PTXSHIFT`) $\times$ `NPROC` where `reverse_map[i]` stores all the page table entry pointers of different processes that map to the ith physical page.
Then the following functions were implemented :

- `pageswapinit`: This initializes the starting block numbers of swap slots and other attributes to 0.
- `inc_rmap(pte_t* pte)/dec_rmap(pte_t* pte)`: This takes a page table entry pointer as input with physical address being `pa`, increments/decrements `rmap[pa]` and pushes `pte` to `reverse_map[pa]`.
- `inc_swap_table(pte_t* pte1, pte_t* pte2)` : This function takes a page table entry of a swapped page of a parent process (`pte_1`), finds the swap slot corresponding to this entry and updates the struct parameters with `pte_2`, the page table entry of the child process.
- `swapout_helper(uint pa,int block)`: This function is called to update swap space struct and rmap/reverse_map when we want to move the contents of page at physical page `pa` to the swap slot with id `block`. So we copy `rmap[pa]` to `swap_space[block].refcnt` and `reverse_map[pa]` to `swap_space[block].pte_array`. Since we have pointers in `reverse_map[pa]`, we modify the flags of all these page table entries : turn on `PTE_SWAPPED`, turn off `PTE_P` and put `block` in the first 20 bits. Finally we clear `rmap[pa]` and `reverse_map[pa]` as this page is now free to use.
- `swapin_helper(uint pa,int block)`: Same as above with roles of swap slot and physical page swapped.
- `swap_page_out()`: We find the victim page (say having physical page number `pa`) according to the page replacement policy of Lab-4. We decrement the rss values of all processes mapping to this page. Next we find an empty swap slot and if there isnt we panic. Then using another `page_disk_interface` helper function in `bio.c` we copy the page into the slot, call `swapout_helper` and free the page at `pa` using `kfree`.
- `flush(pte_t* page)`: This function takes a page table entry of a swapped page as an input, finds the swap slot `s` corresponding to this page, removes `page` from `swap_slot[s].pte_array` (we panic if we don't find) and decrements `swap_slot[s].refcnt`. If it turns into 0, we free the slot.
- `case_swap(uint va, struct proc* p, pte_t* pte)`: This function handles the case when page fault occurs due to the hardware accessing a swapped out page. So we bring in the page from swap space to a new page and update the swap space struct parameters, rmap and reverse_map and rss values of processes that map to this page.

- `case_cow(uint va,struct proc* p,pte_t* pte)`: This function handles the case when page fault occurs due to the hardware accessing a shared page that has `PTE_W` off. We check the `rmap` value of this page. If it is 1, we just turn on `PTE_W` else we create a new page, copy the contents to it and modify `pte` to point to this new page. We also update the swap space struct and rmap/reverse_map parameters.
- `page_fault()`: This is the function that is called when `T_PGFLT` is raised. We consider only 2 cases of page fault - either the page is swapped or the page is shared and some process wants to write on it. Depending on the case we call `case_swap` or `case_cow` appropriately.

2 vm.c

- `mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm, int var)` : We modify this function to consider 3 cases now instead of just the default implementation : first case is the default implementation itself (which is needed in `setupkvm()`), the second case is default implementation + updating `rmap` and `reverse_map` and third case is default implementation + updating `rmap` and `reverse_map` + turning OFF `PTE_P`. The third case is required when you copy a page table entry corresponding to a swapped page.
- `allocuvm(pde_t* pgdir, uint oldsz, uint newsz)` : For every new page that we get from `kalloc()`, we increment the rss value of the current process.
- `dellocuvm_proc(struct proc* p, pde_t* pgdir, unit oldsz, uint newsz)` : If a page table entry has `PTE_T` bit ON, we just decrement the `rmap` values of the physical page and the rss values of process `p` and call `kfree` on this page. Else if `PTE_SWAPPED` bit is ON, we call `flush` defined in `pageswap.c`.
- `freevm_proc((struct proc* p, pde_t* pgdir)` : We call `deallocuvm_proc` in it and moreover also free the pages of the page tables of the process `p`.
- `copyuvm(pde_t *pgdir, uint sz, struct proc* p)` : When we copy a page table entry with `PTE_P` on, we turn off the `PTE_W` off in both the parent and child process's entries. Else if `PTE_SWAPPED` is on, we copy the entry as well as call `inc_swap_table` in `pageswap.c` to update the struct entries of the slot in which the page of the parent process was swapped into (update is needed because the child process also points to this swapped page).

3 proc.c

- `select_victim_process()` : This finds the process with largest `rss` value and least `pid` among multiple such processes.
- `select_victim_page(struct proc* p)` : This selects any page of the victim process that has both `PTE_P` and `PTE_U` on but `PTE_A` off.
- `clear_access(struct proc* p)` : If we fail to find a victim page in above funtion, we clear the access bit of some 10% of pages which have both `PTE_U` and `PTE_P` on.
- `page_replacement()` : This returns the victim page's page table entry that would be swapped.
- `rss_incrementer(uint pa)/rss_decrementer(uint pa)` : This iterates through all processes and checks which all of them have mappings to this `pa` and increments/decrements their `rss` value.

4 kalloc.c

- `kfree()` : When we free a page we first check if its `rmap` value is 0. If it is then we fill it with all 1s (to make it a free page) and make it as the head of `kmem.freelist`.
- `kalloc()` : Here if we don't find any free page, we call `swap_page_out()` and again call `kalloc()`.

5 bio.c

- `page_disk_interface(char* page, uint blockno, int param)` This function implements transfer of data from swap slots to physical pages. The direction of transfer is determined by `param`. The data is transferred by copying to/from each of the 8 blocks of the swap slot one-by-one by first bringing the block in the buffer and then using `memmove(void* dst,void* src,int size)` to copy to/from the buffer and then doing `bwrite()` and `brelse()`.