CS450 -PA3
Jenifer Rodriguez Delgado
Yousef Suleiman

Design of `GetSharedPage()`

`GetSharedPage()` takes an integer `key` (any integer other than 0) and an integer `count` (greater or equal to 1) signifying how many shared pages the user process is requesting under that `key`. Any other process using a subset of those shared pages under the same key will be writing onto the same physical pages in memory.

The `key` was designed to have an integer value to make simple and quick comparisons. It is not allowed to have a value of 0 as 0 is used as a sentinel value by a function called `get_proc_key()` that returns 0 if it can't find a key associated with a process. This will be explained more in depth later in this document.

The design of `GetSharedPage()` was made possible by keeping track of metadata about the physical pages that are shared as well as metadata about the address spaces of the processes that can call `GetSharedPage()`.

The metadata about the shared pages themselves were kept in an array of 32 `struct shared_page` called `all_shared_pages`. The `struct shared_page` holds metadata about a single shared page. This includes the page's `key`, its `page_number` (incase this page belongs to a set of pages under a specific key when a user requests more than one page), the physical address of the page in memory, an array of `proc`'s sharing that page, and their corresponding virtual addresses of the page in their own address space.

The metadata about the address spaces of the processes that call `GetSharedPage()` are stored in a bitmap `sh_bit_map` within each `struct proc` (`proc.h`) with 0s to represent free pages and 1 to represent otherwise.

Because we are using arrays to save metadata, the number of max shared pages and max processes sharing a page are defined. It was chosen to use arrays as xv6 lacks a `malloc`-like allocator so it is difficult to use data structures that require dynamic allocation.

`GetSharedPage()` uses 2 helper functions to get the job done: `where_does_fit()` and `find_shared_page()`.

After checking if the parameters passed to it are valid, `GetSharedPage()` calls `where_does_fit()` and stores its return. Function `where_does_fit()` takes a `count` of requested pages, gets the bitmap `sh_bit_map` of the calling process, and looks for a position that can fit the number of pages specified by count then returns it. If no position is found that can fit the requested count, -1 is returned.

GetSharedPage() then makes calls to find_shared_page() which takes a key, page_number, and position to put the page in sh_bit_map. In short, function find_shared_page() calls kalloc() if needed, updates the metadata, and maps the shared page to the calling process's pgdir. However, find_shared_page() only does the job for a single page. If the user requests more than one page, GetSharedPage() uses a for loop to call find_shared_page() multiple times.

To explain what find_shared_page() is doing, it takes the given key and page_number used to uniquely identify every physical shared page, then walks through the array all_shared_pages to find a shared page with matching key and page_number. If found, the calling process is added to the array of processes sharing the page in all_shared_pages, bitmap sh_bit_map is updated at the specified position, then uses mappages in vm.c to put the page in the pgdir of the calling proc by passing the virtual_address (computed through sh_bit_map and macros KERNBASE and PGSIZE) and the physical address already stored in all_shared_pages since the page already exists.

If an existing page with the key and page_number is not found, a new page is allocated using kalloc(). The page is zeroed out with memset() and similar process occurs as before where the calling process is added to the array of processes sharing the page in all_shared_pages (along with the new key, page_number, and physical address), sh_bit_map is updated, and mappages is called with the respective parameters.

- Note: because mappages is static, a separate wrapper function getMappages was used to access it

If anything fails such as kalloc() or mappages() returning -1 or too many processes sharing the page, then find_shared_page() returns -1. If successful, it returns the virtual_address of the page.

After calling find_shared_page() in a for loop, GetSharedPage() returns the virtual_address from the last call to find_shared_page(). (it breaks its for loop and returns -1 if find_shared_page() returns -1)

Design of `FreeSharedPage()`

`FreeSharedPage()` takes an integer key and returns the number of pages removed from the address space of the user process if successful and -1 if not.

The design of `FreeSharedPage()` was made possible by again using 2 helper functions: `get_num_sh_procs()` and `remove_shared_page()`.

After checking for valid parameters, `FreeSharedPage()` begins by calling and saving the return of `get_num_sh_procs()` which takes a `key` and returns the number of pages that the calling process has under that `key`. `FreeSharedPage()` then calls `remove_shared_page()` that many times in a for loop. Function `remove_shared_page()` takes a `key` and `page_number` to remove the specified page from user address space and possibly `kfree()` the physical page if this was the last process sharing it.

Function `remove_shared_page()` begins by looking in `all_shared_pages` for a corresponding `key` and `page_number`. If found, the calling process is looked for in the array of process sharing the page and is removed, its corresponding virtual address is also removed, and the page is unmapped from the process's `pgdir` by using `walkpgdir` to get the `pte` and set it to 0. Lastly, the corresponding bit in bitmap `sh_bit_map` is unset.

- Note: again, because `walkpgdir` is static, a separate wrapper function `getWalkpgdir` was used to access it

Function `remove_shared_page()` then calls its own helper function `zero_procs()` which returns 0 if the calling process appears in `all_shared_pages` at least once and 1 otherwise. If `zero_procs()` returns 1, `remove_shared_page()` uses `kfree()` on the physical page, Finally the physical address of the page (previously used to possibly `kfree`) is set to 0 in `all_shared_pages`.

Function `remove_shared_page()` returns -1 if the key is not found or the key does not belong to the calling process. If that's the case, `FreeSharedPage()` breaks its for loop and also returns-1.

Side note:

`Exit()` in `proc.c` was modified to do something similar to `FreeSharedPage()` where it takes all the keys associated with the exiting process using a function `get_proc_key()` which returns the first key it finds associated with the calling process and 0 if no key is found. `Exit()` makes calls to `remove_shared_page()` until `get_proc_key()` returns 0, freeing all the shared pages of the exiting process.

- Note: 0 was used as a sentinel value by `get_proc_key()` so if a process had a `key` with the value 0, it would cause complications when freeing at `exit()`. For this reason, `key` was designed to have a value other than 0.

Changes were made in the following:

- `sysproc.c`
  - contains the actual body of `GetSharedPage()` and `FreeSharedPage()`
- `types.h`
  - makes a `typedef` for `int key_int`
- `defs.h`
  - defined and declared struct `shared_page` and array of struct `shared_page` called `all_shared_pages`
    - struct `shared_page` contains metadata on a particular shared page such as an integer `key`, the integer `page_number` under a certain `key`, the physical address `phy_add`, and an array of struct `proc* procs_sh_page`, and an array of int virtual addresses `vir_add_of_page`
  - declared `init_shared_page()`, `find_shared_page()`, `where_does_fit()`, `remove_shared_page()`, `zero_procs()`, `get_num_sh_procs()`, and `get_proc_key()`
    - These are all defined in our own . c file called `sharepage.c`
- `vm.c`
  - in `deallocuvm()` don't `kfree` a page if its `phy_add` is still in the `all_shared_pages` array (meaning some process is still using the shared page). This is done by searching `all_shared_pages` for `phy_add` equal to `pa` in the function
  - in `copyuvm()` the parent's shared address space was completely copied and mapped over to the child's using the beginning and end of the shared memory
    `uint sh_mem_start = KERNBASE - PGSIZE;`
    `uint sh_mem_end = KERNBASE - MAXSHPAGES * PGSIZE;`
    Then walking `pgdir` to make sure the page is there and using `mappages()` to map to the child

- also added `getMappages()` and `getWalkpgdir()` to easily access `mappages()` and `walkpgdir()` static functions in `sharepage.c`

- `proc.h`
  - added bitmap of shared pages `sh_bit_map` to struct `proc`
- `proc.c`
  - `userinit()` initializes `all_shared_pages_array` through `init_shared_page()` by zeroing everything out as well as using `memset()` to zero out the bitmap `sh_bit_map`
  - change `fork()` to copy the bitmap `sh_bit_map` and `all_shared_pages` metadata
  - change `exit()` to make calls to `get_proc_key()` and `remove_shared_page()` to remove shared pages from exiting processes