COMP312005 Operating System, Spring 2019

Project 3 : Virtual Memory

**---- GROUP ----**

Hyosang Park <latter2005@gmail.com>

Yongho Lee <jhyn207@gmail.com>

Jisu Kim <wltn4511@gmail.com>

**---- PRELIMINARIES ----**

Contribution : 1:1:1

**---- Data structures and functions ----**

**Struct page**

**{**

**Void \*addr : user’s virtual address**

**Bool read\_only : read\_only bit for page**

**Struct thread \*thread : owning thread**

**Struct hash\_elem hash\_elem : struct thread pages.**

**Struct frame \*frame : frame of page.**

**Block\_sector\_t sector : starting sector of swap area, or -1 if none.**

**Bool private : write-back bit for files**

**Struct file \*file : file structure for page**

**Off\_t file\_offset : file offset**

**Off\_t file\_Bytes : bytes to read/write.**

**}**

**Struct frame**

**{**

**Struct lock lock : lock to prevent simultaneous access**

**Void \*base : virtual base address**

**Struct page \*page : mapped page of the frame**

**}**

**Functions**

1. **Page.c**

Static void destroy\_page()

* Destroys a page in current process’s page table. If it exists in process’s page table. Locks page with frame\_lock, and releases after page is freed, by frame\_free() function to ensure no other process accesses the page during destroying procedure.

Void page\_exit()

* Destroys whole page table of the current process, by hash\_destroy() function. hash\_destroy() function frees all entry of the page table.

Static struct page \* page\_for addr()

* Returns page containing the given virtual address by parameter. First, function checks if address is valid, by check if address is not beyond the bound of stack space. If page exists in the address, it returns page of that address. If not, If address is within 32-bytes of stack pointer, it is also valid address, so this function allocates one more stack page, and returns new space for page allocation.

Static bool do\_page\_in()

* Function implementing page-in. Finds allocatable space, and locks it. If page exists in the disk(p->sector != 1), it swaps in the page. If page exists in the file, it reads the page from the file. Else, it provides all-zero page.

Bool page\_in()

* Gets address of fault-occurred address, and handles page fault. Allocates new page by do\_page\_in, it updates page table. Returns true on success.

Bool page\_out()

* First, function makes page invalid, to prevent race condition. And then, it checks whether this page’s dirty bit is on or not, and its file content is modified or not(can be checked by p->private. Modified files or pages are written back to the disk. Otherwise, evicts it without write-back.

Bool page\_accessed\_recently()

* Checks whether page is recently accessed using pagedir\_is\_accessed() function. it returns true if accessed, and sets accessed bit to false, and false if not.

Struct page \* page\_allocate()

* Adds new mapping for vaddr to the page in the table. Initializes pages, and sets virtual address for the page(p->addr = pg\_round\_down(vaddr), and returns the page.

Void page\_deallocate()

* Evicts the page of the virtual address, and removes it from page table with hash\_delete(). Have to check p->private(is file modified?) before deleting.

Unsigned page\_hash()

* Returns hash value for the page element.

Bool page\_less()

* Returns true if page A precedes page B

Bool page\_lock()

* Locks the page containing address into the physical memory. First, it locks the page. If p->frame is NULL, it pages in, and sets the page by do\_page\_in() and pagedir\_set\_page(). If p->frame already exists, it just locks the page and returns true.

Void page\_unlock()

* Unlocks the page by frame\_unlock()

1. **Frame.c**

Void frame\_init()

* Initializes the frame. it initializes lock(new semaphore), allocates frame, and initializes all frames.

Static struc frame \* try\_frame\_alloc\_and\_lock()

* Acquires lock, and checks free frame. If free frame is found on first loop, It immediately returns the free frame. If first attempt fails, it finds frame to evict in second loop. Second loop searches frame 2 times(frame\_cnt\*2). If page\_accessed\_recently() is true, it sets the bit to 0 and continues. If page\_accessed\_recently is false, that frame is evicted, and frame is used to allocate the page

Struct frame \* frame\_alloc\_and\_lock()

* Invokes try\_frame\_alloc\_and lock() 3 times, and sleeps for 1000 milisecond each time it fails, to ensure better possibility of success.

Void frame\_lock()

* Fixes the mapping of frame, using the semaphore lock.

Void frame\_free()

* Releases frame by make f->page NULL, and lock\_release().

Void frame\_unlock()

* Unlocks frame f by lock\_release(&f->lock), so it can be evicted.

1. **Swap.c**

void swap\_init()

* Gets swap device by block\_get\_role(), and creates bitmap of the size (block size of the device)/(page sector), and initializes semaphore lock swap\_lock.

Void swap\_in()

* Reads the block (PAGE\_SECTOR) times from device. Updates p->sector after reading.

Bool swap\_out()

* Acquire lock, and swap out page P. writes out page sectors for each modified blocks. Updates page structures after swap out.

Page Table Management

=================

---- ALGORITHMS ----

When page fault is occurred, page\_fault() in exception.c handles page fault. This function checks cause of the page fault. If page fault’s cause is known to be handled by page in,(caused by user, and page is not present) page\_fault function handles it by invoking page\_in() function.

Structure page and frame has each data structure to store it’s mapping information. When page\_in is invoked, struct page’s frame member is set to NULL. To allocate frame to page, do\_page\_in() function is invoked.

In do\_page\_in(), frame\_alloc\_and\_lock() function is invoked. frame\_alloc\_and\_lock function is implemented with lock(scan\_lock) for synchronization,(to allow only one process to search for available memory space to allocate new frame).

Frame\_alloc\_and\_lock() searches for free frame(frame f->page == NULL). If found, it allocates free frame to page. But in case that there’s no free frame to allocate, existing frame should be evicted to allocate frame to current page. Policy implemented for eviction will be explained below. Victim frame chosen by policy is evicted by function page\_out().

After search and allocating, lock is released.

If all attempts above failed, NULL is returned.

Swap-in and out

=================

To Swap in-and out from disk, if there’s no free frame to be allocated, eviction of existing frame is needed. least-recently accessed policy is implemented to choose eviction frame.

Policy is implemented by function try\_frame\_alloc\_and\_lock() in frame.c. Before eviction, frames are checked by page\_accessed\_recently(). Function checks whether certain page is accessed recently or not, by pagedir\_is\_accessed(). pagedir\_is\_accessed() returns true if page is accessed before, false is not. If pagedir\_is\_accessed returns true, page\_accessed\_recently() returns true(means page is accessed recently), which means it can’t be evicted.

If non-recently-accessed frame is found, it becomes victim block, and get evicted by page\_out.

When evicting the frame, we should check whether frame is modified or not, and whether frame is containing memory-mapped files. We can check frame is dirty or not by function pagedir\_is\_dirty. Also, we have to check if frame contains memory-mapped file information.

Memory-mapped frame is checked by checking (p->file)member of page structure. If file is null, it means that frame is not memory-mapped frame. We don’t have to write back to the file in the disk.

If frame is memory-mapped(p->file member is not NULL), contents of frame is written to disk before eviction.

Stack Growth

=================

Initially, stack is set to certain size(ex. 1mb), and page fault occurs when we try to access out of boundary of stack. Our goal is to grow our stack, when page fault can be handled by growing our stack.

To implement stack growth, we have to check if page fault is occurred by user, and if it’s access to the stack. If true, we can allocate new page for the request.

We get esp, stack pointer of current process, and fault\_address, address that user is trying to access. We check 2 conditions to check if it’s access to the stack.

First, we have to check if access is stack area of user process’s memory, by

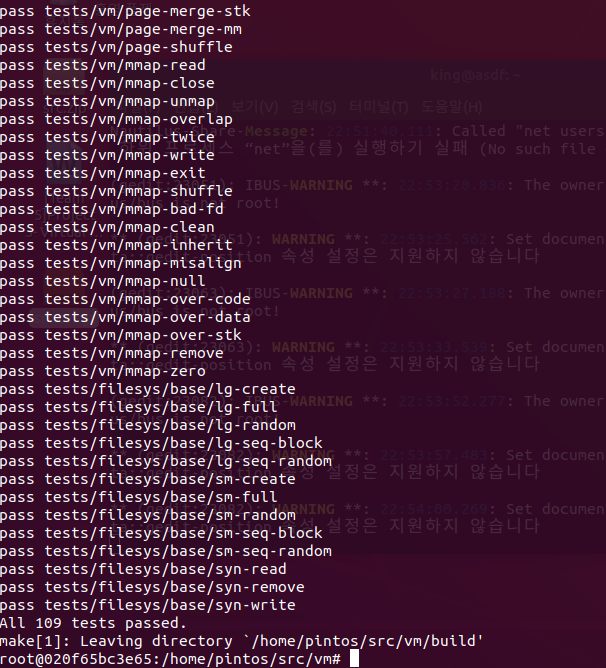
PHYS\_BASE – MAX\_STACK\_SIZE <=fault\_address.

Second, we have to check if it’s access to the stack by

Esp-32 <fault\_address,

Because PUSH, PUSHA uses at most 32 bytes beyond stack pointer.

By checking two conditions, we can allocate new page at the end of current-allocated page.



참고:

<https://github.com/ChristianJHughes/pintos-project3>

<https://github.com/Nikhila511/Virtual-Memory-PintOS>

[https://bogus919.tistory.com/entry/pintos-%EC%9A%94%EC%95%BD-3-Virtual-Memory](https://bogus919.tistory.com/entry/pintos-요약-3-Virtual-Memory)