Operating System (Fall 2014) Project 3 Virtual Memory

Jongwook Choi

Introduction

The goal of the third pintos project is to implement Virtual Memory . There are three major requirements as follows.

- **Paging:** By mapping user pages and frame pages to virtual memory , more memory can be used through frame eviction and swapping (as much as swap allows) even if memory is insufficient.
- **Stack Growth:** Even if the stack area spans two or more pages , a memory error does not occur and the pages allocated to the stack area are automatically allocated so that the stack can grow automatically .
- **Memory-Mapped Files:** Memory- mapped to enable file read/write by mapping files to virtual memory. Implements the file scheme and the related system calls mmap() and munmap() .

This report focuses on the three requirements above , how each was designed and implemented, and what problems may arise. Let's briefly explain what it is and how we solved it.

1. Paging

1.1. Data Structure

To implement paging and virtual memory, several data structures need to be implemented. The implemented ones are (1) Frame Table, (2) Supplemental Page Table, (3) Swap Table . First, we briefly explain the implemented data structures and operations.

(1) Frame Table

Frame refers to the area of physical memory . One frame has a size of PGSIZE and is all aligned . All memory areas that can be paged must be managed through the frame table .

There is exactly one frame table entry for each frame page , which is information contained in the frame table , and it stores the following information.

```
/** Frame Table Entry */
struct frame_table_entry
  {
    void *kpage;                           /* Kernel page, mapped to physical address */

    struct hash_elem helem;                  /* see ::frame_map */ /*
    struct list_element element;             see ::frame_list */

    void *upage;                           /* User (Virtual Memory) Address, pointer to page */ /* The associated
    struct thread *t;                      thread. */

    bool pinned;                           /* Used to prevent a frame from being evicted, while it is acquiring some resources.
                                              If it is true, it is never evicted. */
  };
```

• kpage : This is the kernel page address of the mapped frame that becomes the key value in the Hash table . In PintOS, the physical address and kernel page correspond 1:1, so we will simply handle the physical address by unifying it as kpage .

• upage : Virtual memory address of the user page where the frame is loaded . • t : Refers to the owner thread that loaded the frame .

• pinned : Required for frame pinning . This is explained in detail below. • helem, lelem : Element objects required to be included in the frame hash table and linked list .

Of course, there is only one frame table as a global scope . The supported operations are as follows.

• void vm_frame_init() : initialization. It is called once at the beginning. • void* vm_frame_allocate(enum palloc_flags, void *upage) : Creates a frame page corresponding to the user virtual address upage , performs page mapping , and returns the kernel address of the created page frame .

- void vm_frame_free(void*) : Plays the role of releasing the page frame . The corresponding entry is removed from the frame table. and the resource (page) is released.
- vm_frame_pin(), vm_frame_unpin() : Required for pinning . This is explained in detail below.

To implement the above operations, a hash table (struct hash frame_map) of (kpage ÿ frame table entry) was used internally. This allows you to quickly look up the information of the frame when the key value, kpage, is given .

In addition, all entries existing within the frame table are separately managed as a linked list (struct list frame_list), which is necessary for the frame eviction (clock) algorithm explained below . It was implemented (straightforward) to always guarantee data consistency for all operations that add or delete pages from the frame table . In addition, since operations accessing the frame table must consider concurrency , all operations on the frame table must be held in a lock so that they can be performed in the critical section (frame_lock).

(2) (Supplemental) Page Table

Page table conceptually serves to map user virtual addresses to physical addresses . When a user program accesses a virtual address , it has the same effect as accessing the memory area of the corresponding frame . In PintOS, because of the format of the page table , a separate data structure is needed to store all additional information related to the (user virtual) page, and this is the Supplemental Page Table (hereinafter SUPT) .

SUPT is created once per thread (process) , and can be understood as a mapping from uaddr (user page) to SPTE (supplemental page table entry) . SPTE stores the following information.

```
enum page_status {
    ALL_ZERO,               // All zeros
    ON_FRAME,               // Actively in memory
    ON_SWAP,                // Swapped (on swap slot) // from
    FROM_FILESYS            filesystem (or executable)
};

struct supplemental_page_table_entry
  {
    void *upage;                    /* Virtual address of the page (the key) */ /* Kernel page
    void *kpage;                    (frame) associated to it.
                                      Only effective when status == ON_FRAME.
                                      If the page is not on the frame, should be NULL. */
    struct hash_elem elem;

    then page_status status;

    bool dirty;                     /* Dirty bit. */

    // for ON_SWAP
    swap_index_t swap_index; /* Stores the swap index if the page is swapped out.
                                      Only effective when status == ON_SWAP */

    // for FROM_FILESYS
    struct file *file;
    off_t file_offset;
    uint32_t read_bytes, zero_bytes; bool
    writable; };
```

• upage : This is the key value and stores the user virtual address . •
Status : Indicates the status of the page .

– ON_FRAME : When loaded in frame. At this time, the address of the loaded frame must be stored in kpage , In the frame table, you can find the corresponding frame table entry with kpage as the key.
– ON_SWAP : This is when the current page is evicted from the frame and exists on the swap disk . At this time, the location on the swap disk is stored in swap_index , and through this value, the appropriate location on the disk can be read and swap in later.
– ALL_ZERO : This means that the page has not been loaded into the frame, but all content is filled with 0.
– FROM_FILESYS : This means that the page is not in the frame , but its contents must be loaded from the file system (executable or memory-mapped , etc.). The offset of a file can be determined using file, file_offset, read_bytes, zero_bytes , etc. If writable is false, it is a read-only page.

• dirty : dirty bit of the page . Since the dirty bit of a swapped out page cannot be retrieved through pagedir_is_dirty(), it is necessary to separately store whether it is dirty . Details are explained in memory mapped files, where it is used.

The use of SUPT is explained in detail in the Page Fault handling algorithm below.

(3) Swap Table

Provides operations for swap .

• swap_index_t vm_swap_out(void *page) : Performs Swap-Out . The contents of the page are recorded on the swap disk and a swap_index that identifies the location is returned.
• void vm_swap_in(swap_index_t swap_index, void *page) : Performs Swap-In . swap_index location
Read a single page from and write it to the
page . • void vm_swap_free(swap_index_t swap_index) : Just discards the relevant swap region .

The number of total sectors on the swap disk divided by PGSIZE is the number of available swap slots . Since the page size is larger than the size of the swap sector , PGSIZE / BLOCK_SECTOR_SIZE contiguous blocks are required to store one page in swap . Which swap slots are available is managed using the bitmap data structure, and implementing swap in/out by mapping the corresponding blocks and pages is very straightforward , so detailed explanations are omitted. (see commit b20fe2c9)

1.2. Swapping and Evicting

During frame allocation , if page allocation fails, there is insufficient memory, so swapping is required in this case. In other words, a certain (frame) page is swapped out to a swap disk and a new page is allocated to the empty space. At this time, the following things happen:

• Select the frame to evict . (second-chance clock algorithm) • Release
page mapping of the frame . That is, remove the upage mapping from pagedir of the owner thread. • Swap out the
contents of the frame . (When readonly, use the file system) • The relevant frame
is removed from the frame table (the page is also freed) • After that,
the page is reallocated, the newly allocated frame is inserted into the table, and the mapping is processed.

To select the frame to be eviction , the second-chance algorithm was used.

• Maintain the victim pointer (clock_ptr) and circularly traverse the elements of the frame table . • If the
reference bit is 1 , set the reference bit to 0 and move on to the next entry . • If the reference
bit is 0 , select it as the eviction target.

To check/set the reference bit , a method was used to simply check the upage mapped to the corresponding frame in pagedir . To find the reference bit , use pagedir_is_accessed(), and to set the reference bit to 0 , use pagedir_set_accessed() (when accessing a page in a user program , the reference bit is internally set to 1). Since frame entries are managed through a linked list , the above algorithm can be implemented efficiently.

```
/** Frame Eviction Strategy : The Clock Algorithm */ struct
frame_table_entry* clock_frame_next(void); struct frame_table_entry*
pick_frame_to_evict( uint32_t *pagedir ) {

  size_t n = hash_size(&frame_map); if(n == 0)
  PANIC("Frame table is empty, can't happen - there is a leak somewhere");

  size_t it; for(it
  = 0; it <= n + n; ++ it) // prevent infinite loop. 2n iterations is enough {

    struct frame_table_entry *e = clock_frame_next(); if(e->pinned)
    continue; else
    if( pagedir_is_accessed(pagedir, e->upage)) {
      // if referenced, give a second chance.
      pagedir_set_accessed(pagedir, e->upage, false); continue;

    }

    // OK, here is the victim : unreferenced since its last chance
    return e;
  }
  PANIC ("Can't evict any frame -- Not enough memory!\n");
}
```

1.3. Page loading: Page Fault Handler

A page fault occurs when a page that has not been loaded is accessed, that is, when a memory area that does not exist in pagedir is accessed . At this time, if there is a page in SUPT, it is not an incorrect memory area access, but the frame has not been loaded due to swapped out or lazy-load, etc. , so it is necessary to reload the page . This task is implemented in the vm_load_page() function.

1. Check the SUPT entry . If not, it is an incorrect memory access and is killed after a fault .
2. Obtain a frame page to store the contents of the page . At this time, other frames may be evict .
3. Check the status saved in SUPT and load appropriate data into memory.

   • ALL_ZERO : Just fill with 0 (memset). •
   ON_FRAME: NO-OP. (can't happen?)[1] •
   ON_SWAP: Swap in is required. Load data by calling vm_swap_in() . • FROM_FILESYS : Read
   data from the file system through information such as file pointers stored in SPTE . Offset
   Read_bytes are read from file_offset , and the remaining area can be filled with 0.

4. Upage and map the new frame created in 2. (pagedir_set_page()). Now the state of this frame is again
   It becomes ON_FRAME .

By loading the page by filling it with data through this handler , resolving the page marked as lazy load (eg FROM_FILESYS) is also naturally performed.

1.4. Processing during Process Loading (Lazy-Load)

Since the above paging scheme has been implemented, when PintOS loads the process , it has become possible to lazy load only the necessary segments instead of loading all segments into memory . load_segment() (see process.c:639) In the past, while the allocated page was filled from a file and install_page was done immediately, the pointer to the file is set so that it can be loaded lazily using vm_supt_install_filesys() (this file is executable , so the pointer remains valid until the process terminates). and offset and size information are recorded in SUPT .

However, when setting up the first user stack (setup_stack()), lazy loading is not used. Of course , appropriate processing was required so that the pages and frames allocated to the initial stack area could be correctly loaded into the SUPT and frame tables .

1 Of course, since it is already loaded, we just return without allocating a new frame .

1.5. Processed upon Process Termination

When a process terminates, the pagedir occupied by this process is destroyed and all pages allocated are released. In addition, the SUPT table must also be deleted. Since the SUPT table is per-thread scope , you can simply delete it. However, if there are frames or swap slots allocated by this process, these must also be properly released. If you do not release it (the thread is freed), a problem may occur because when you select a frame to evict later , all necessary information such as pagedir has already been destroyed.

This part is simple in concept, but somewhat difficult to implement. For details, see commit eb9d8be0 Let's refer to the diff and message . To point out the most important point, since we previously saved the kpage of the frame mapped to SPTE , we can use this as the key to delete frames that are no longer valid in the frame table . If the state of the page is SWAPPED , you can release the occupied swap . In addition, if there is a memory mapped file , it is also released (described in 3.).

Since SUPT is the process of being destroyed , the destructor callback performed for each element of the hash table is used to remove the corresponding frame entry (vm_frame_remove_entry()) . When a thread (process) terminates and resources are released, all pages related to pagedir are also freed . At this time, care must be taken to avoid double-freeing .

1.6. Frame Pinning

When accessing user memory , problems can occur if another page fault occurs while the kernel is processing virtual memory paging2 . For example, if a page fault occurs while reading user memory and writing to the file system by performing a write system call (the user page has been swapped) , different access to the file system is required when swapping in . However, in PintOS, the file system can only be entered by holding the lock once, so a kernel panic due to double-lock occurs.

To solve this, a method called frame pinning must be used. Pinned frames are excluded from eviction . As an implementation method, a boolean field called pinned was placed in the frame table entry to record whether it was pinned , and the pinned bit of a specific frame could be set using functions such as vm_frame_unpin() and vm_frame_pin() .

- When allocating a frame for the first time , pin it to prevent eviction from occurring , and when the frame is finished loading, (Refer to vm_load_page()) Release the pin .
- Before performing system calls of read() and write() , pin all user pages corresponding to the target buffer . This operation is implemented in vm_pin_page() , which looks up the page in SUPT and pins the frame if it is loaded in the frame . It may not be in SUPT , but since lazy-load is used, there may be stack areas that have not yet been accessed (stack growth), so it is ignored.
- After the system call is completed, all previously pinned frames are unpinned . vm_unpin_page() in that The action is implemented.

[2] Test cases such as 'page-merge-mm', 'page-parallel', etc.

2. Stack Growth

Previously, the stack of a user program consisted of only one page , and if this was exceeded, a page fault occurred. The goal is to allow the stack to grow within the limit allowed by the page .

First, a page fault occurs when the user program exceeds the stack . Therefore, when a page fault occurs, it is determined whether it is a stack access, and if so, a new page is allocated. Of course, there are cases where incorrect memory is accessed rather than stack access, so an appropriate method is needed.

2.1. How to determine Stack Access

First, obtain the stack pointer esp . When the address where a page fault occurred is called fault_addr , if both of the following conditions are satisfied, a new page must be allocated to increase the stack.

• Is it the stack area of user memory : PHYS_BASE - MAX_STACK_SIZE <= fault_addr && fault_addr < PHYS_BASE (MAX_STACK_SIZE is defined as 8MB). • Is it above the stack frame : esp <= fault_addr

or fault_addr == f->esp - 4 or fault_addr == f->esp - 32. This is according to the manual, 80x86 PUSH/PUSHA operation is 4 bytes below the stack pointer (PUSH) or the 32-byte base (PUSHA) .

However, obtaining esp is slightly different depending on user mode and kernel mode . If it is in user mode, it can be obtained directly from the intr_frame (f->esp), but a page fault may occur in the stack area while performing a system call such as read() or write() . Previously , in Project 2, incorrect user memory access was identified using a page fault- based method (refer to the Accessing User Memory section), so at this time , the user program's stack pointer does not exist in f->esp .

To solve this problem, the user's esp is stored in the thread structure when making a system call . Then, even if a page fault occurs and enters the interrupt handler , the esp can be determined based on this . If (1) in user mode, use f->esp as is, (2) in kernel mode, use curr->current_sep stored in the thread .

```
@@ -118,4 +118,8 @@ struct thread
      struct list file_descriptors;                 /* List of file_descriptors the thread contains */

      struct file *executing_file;                  /* The executable file of associated process. */
+
+     uint8_t *current_esp;                         /* The current value of the user program's stack pointer.
+                                                      A page fault might occur in the kernel, so we might need to store esp on
+                                                      transition to kernel mode. (4.3.3) */
 #endif
```

```
  /* (4.3.3) Obtain the current value of the user program's stack pointer.
   * If the page fault is from user mode, we can obtain from intr_frame `f`, * but we cannot from kernel
   mode. We've stored the current esp * at the beginning of system call into the thread
   for this case. */
  void* esp = user ? f->esp : curr->current_esp;
```

2.2. Paging processing when stack increases

If the previously determined conditions are met, the fault_page (the page start address that aligns fault_addr with PGSIZE) is found and the page is assigned to it.

Therefore, add a (new) entry in the supplemental page table with fault_page as the user address . The type of the added entry is ALL_ZERO , and all data is zero-filled (see vm_supt_install_zeropage()).
In Linux systems, uninitialized stacks contain garbage values such as 0xcc , but zero is used here for convenience.

Now, after allocating a frame through vm_load_page() and loading the page into memory, the interrupt handler processing for stack growth is completed.

```
// Stack Growth
bool on_stack_frame, is_stack_addr;
on_stack_frame = (esp <= fault_addr || fault_addr == f->esp - 4 || fault_addr == f->esp - 32); is_stack_addr = (PHYS_BASE -
MAX_STACK_SIZE <= fault_addr && fault_addr < PHYS_BASE);
if (on_stack_frame && is_stack_addr) { // OK. Do not
    die, and grow. // we need to add new
    page entry in the SUPT, if there was no page entry in the SUPT.
    // A promising choice is assign a new zero-page. if
    (vm_supt_has_entry(curr->supt, fault_page) == false) vm_supt_install_zeropage
        (curr->supt, fault_page);
}

if(! vm_load_page(curr->supt, curr->pagedir, fault_page) )
          goto PAGE_FAULT_VIOLATED_ACCESS;
```

For detailed implementation, see commit 0703878. Let's take note.

3. Memory Mapped Files

For the memory mapped files function, the mapid_t mmap(fd, addr) and munmap(mapid_t) system calls must be implemented. Their implementation is very straightforward .

First, the struct thread stores information on descriptors memory-mapped to the process .

```
@@ -127,6 +127,9 @@ struct thread
 #ifdef VM
       // Project 3: Supplemental page table. struct
       supplemental_page_table *supt; /* Supplemental Page Table. */
+
+      // Project 3: Memory Mapped Files. struct list
+      mmap_list;                                   /* List of struct mmap_desc. */
 #endif

 /* Owned by thread.c. */
```

The memory-map descriptor must store the following information. It is almost identical to Project 2's file descriptor . Remembers the file, its size, and mapped page address.

```
struct mmap_desc {
   mmapid_t id;
   struct list_elem elem;
   struct file* file;

   void *addr; // where it is mapped to? store the user virtual address size_t size; // file size };
```

3.1. mmap()

The implementation of the mmap() system call is as follows. It is also straightforward .

1. Check the arguments . According to the manual's specification , if the page address is 0, if it is not aligned , if fd is 0 or 1, if it is an incorrect memory area, or if it overlaps with another existing memory area (check whether the page exists in SUPT) etc. are applicable.
2. Open the file and check its size. You can use file_reopen() to manage files separately from the file pointer managed by the file descriptor . Of course, the file opened here is closed by munmap() . 3. Using

vm_supt_install_filesys() , set the page (FROM_FILESYS). You can store which area to read according to the offset in the mmap_desc structure. At this time, the previous pages except the last page are mapped as many as PGSIZE bytes, and the last page must map the remaining bytes . Now, the page is in a state where it can lazy load data from the file system .

3.2. munmap()

The implementation of the munmap() system call is as follows. It is also straightforward .

1. Find mmap_desc corresponding to the given mapping id (fail if not found). While traversing all pages allocated here , vm_supt_mm_unmap() is called to perform appropriate release for each page (described below). 2. In vm_supt_mm_unmap(), each mapped page is released and the necessary tasks are performed.

   • Release the mapping from pagedir to the user address (all later accesses are page faults). • Depending on the status of the page — remove it from the frame table (ON_FRAME) or discard the allocated swap slot (ON_SWAP). • In addition to this, if the page is in a dirty state, it is necessary to write the contents of the page to the file . If it is in a dirty state, the contents of the frame page or swap are recorded in the corresponding file area (swap-in to a temporary page and then recorded again in the file).

Since dirty status is tracked for each page , pagedir_is_dirty() is used to find out whether the page is in a dirty state . The conditions for determining whether a page is dirty are (1) marked as dirty in SPTE3, (2) the user page is dirty, (3) the kernel page (frame) is dirty (using pagedir_is_dirty()) , or one of the three is satisfied. (You have to do this because the user page is a kind of aliasing to the frame page).

When a process terminates (process_exit()), it is also necessary to release all memory-mapped files, if any , just as all open files are closed . When the process terminates, the munmap() system call handler is called to ensure that all of the above processes (in particular, write back on the file) can occur.

Implementation details regarding memory-mapped files are in commit 754ce211 . and a6ebf11c Please refer to diff and message .

[3] Of course, for this purpose , when a frame is evicted and swapped out , whether it is dirty is recorded in SPPE . If you don't record it, write back later This is because it is impossible to know from pagedir information alone whether it should be done or not . When the frame is loaded later , SPTE's dirty flag is reset.

4. Test Results

Project 3 requirements such as Paging, Stack Growth, mmap system call, etc. were implemented and all 109 test cases were passed. These test cases cover the basic functions of the user program and system call completed in Project 2 , as well as the basic operations of stack growth / paging / mmap, handling of exception situations, and complex situations in which multiple processes run simultaneously (parallel merge sort). , communication through files) , etc. are verified.

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

..............................

SUMMARY BY TEST SET

Test Set                                                Pts Max % Ttl % Max
--------------------
tests/vm/Rubric.functionality tests/vm/                   55/ 55 50.0%/ 50.0%
Rubric.robustness                                         28/ 28 15.0%/ 15.0%
tests/userprog/Rubric.functionality tests/userprog/     108/108 10.0%/ 10.0%
Rubric.robustness tests/filesys/base/Rubric               88/ 88 5.0%/ 5.0%
                                                          30/ 30 20.0%/ 20.0%
--------------------
Total                                                          100.0%/100.0%
```

Conclusion

The difficulty of implementation was considerable. Because Paging / Frame -related operations are somewhat complex, it is important to design the kernel-level API neatly, and it is also necessary to implement it accurately without bugs by considering various cases to avoid race conditions . Fortunately , we were able to implement virtual memory to operate correctly through appropriate synchronization and lock processing , even in various concurrency situations such as multiple processes running simultaneously . In most cases, once the details and rough direction of what to do are determined, the implementation is straightforward , but documentation (comments and commit messages) was well done during the work process, so please refer to the comments and commit messages for detailed information .

Some parts seem to need some refactoring, but I wish they had designed a more robust and beautiful design from the beginning and started implementing it. I was able to understand the virtual memory scheme and the kernel's operation process to properly manage it , and although there were many difficulties in design/implementation/debugging, it was a fun and useful time.