2016 Spring CS330 Project 3: VIRTUAL MEMORY

---- TEAM ----

>> Team name.

Team 10

>> Fill in the names, email addresses and contributions of your team members.

Tuan Manh Lai <laituan245@kaist.ac.kr> (Contribution1 = 50%)

Nham Van Le <levn@kaist.ac.kr> (Contribution2 = 50%)

>> Specify how many tokens your team will use.

We use 2 tokens for project 3-1.

We don’t use any token for project 3-2.

2016 Spring CS330 Project 3: VIRTUAL MEMORY

PAGE TABLE MANAGEMENT

==========

---- DATA STRUCTURES ----

A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

In vm/page.h and vm/page.c:

enum data_location {
  NONE,
  MEMORY, // in memory
  SWAP, // in a swap slot
  EXECUTABLE, // in an executable
  MMAP // in a memory-mapped file
};
	This enumeration represents the places where the content of a page can be at (e.g., MEMORY means the page's content is currently in memory).

struct page {
  struct hash_elem hash_elem;
  tid_t pid;
  void * base;
  bool writable;
  bool from_executable;
  bool is_mmapped;
  enum data_location loc;
  struct swap * swap;
  struct frame * frame;
  struct file * mmappedfile;
  off_t mmapped_ofs;
  off_t ofs;
  size_t page_read_bytes;
  struct semaphore page_sema;
};
	Each instance of this struct represents a page of some user process. It contains all important information about a page.

In vm/frame.h and vm/frame.c:

struct frame {
  void * base;
  struct page * page;
  struct list_elem elem;
  bool pinned;
  struct thread *thread;
};
	Each instance of this struct represents a frame in physical memory. It contains all important information about a frame.

static struct list frames_list;	This is a list of all active frames (frames that contain user pages) in the system.
static struct list_elem * cur;	Logically, *frames_list* is a FIFO queue. *cur* is a pointer that points to the first element of the queue (i.e., the element at the front).
static struct semaphore sema;	This semaphore is used to prevent race conditions associated with the global list of active frames.

In threads/thread.h, we added new members to the struct *thread*.

struct hash * pt;	This is the supplemental page table of the thread.
struct file * executable;	This keeps track of the executable file that the user process associated with the thread is loaded from.
void * data_segment_end;	This pointer points to the end of the code and data segments of the user process associated with the thread (in logical address space).

---- ALGORITHMS ----

A2: In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.

We rely on the original page table for doing virtual-to-physical mappings. Therefore, except for when there is a page fault or some process wants to exit, we don't actually have to do anything. The original page table will automatically locate the physical frame given a virtual logical address.

However, for example, when there is a page fault, we will look at our supplemental page table to look for the struct representing the faulted page (by calling find_page). If the struct page is found, we load it (by calling the function load_page). 3 cases can happen:

- It is a swap page
- It is from an executable
- It is an mmap

In those cases, we will need to call frame allocation to allocate a frame for that page. If a free frame is not found, we need to evict one of the occupied frames. Both cases are handled in the allocate_frame function.

If the page is not found, this means it is either an invalid access or a stack access. We will use the heuristic described in B4 to determine whether it is a stack access or not. If yes, stack growth will occur. Otherwise, we call exit command (with exit status -1).

A3: How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?

We avoid the problem by only accessing user data through the user virtual address.

---- SYNCHRONIZATION ----

A4: When two user processes both need a new frame at the same time, how are races avoided?

In our design, whenever a process wants to obtain a new frame, it has to call the function *allocate_frame* (defined in the file *frame.c*). So in order to avoid races that can occur when two or more user processes need a new frame at the same time, we decide to use a semaphore (the struct *sema* defined in *frame.c*) to ensure that the function *allocate_frame* is being executed by at most one process at a time. In other words, whenever a process wants to execute the function, it will first need to down the semaphore; and when the process is about to finish executing the function, it will up the semaphore. In this way, races are avoided. We understand that this approach may limit parallelism. However, this approach is simple.

---- RATIONALE ----

A5: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

The original page table is still used for doing virtual-to-physical mappings. Only when needed, the supplemental page table will be used (e.g., when a page fault occurs).

The supplemental page table of each user process is represented by a hash table data structure. Given the base logical address of some page, the hash table can find the struct representing the page (if any) in O(1) on average. If the struct representing the page is successfully retrieved, then we can easily know where the content of the page is currently at by looking at the information contained in the struct. For hash tables, insertion and deletion also take O(1) on average. So this implementation of supplemental page table is really efficient.

PAGING TO AND FROM DISK

==========

---- DATA STRUCTURES ----

B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

In vm/swap.h and vm/swap.c:

struct swap {
  disk_sector_t base;
};
	Each instance of this struct represents a swapping slot. One swapping slot consists of exactly 8 sectors.

static struct bitmap * used_map;	This bitmap is used to track the usage of all swapping slots. If a swapping slot n is in use, the bit n is true.
static struct lock lock;	This lock is used to prevent race conditions associated with the global bitmap mentioned above.

static struct lock swap_disk_lock;	This lock is used to ensure that only one process at a time is accessing the swap disk (mutual exclusion).

---- ALGORITHMS ----

B2: When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

```
1.          while (true) {
2.            f = list_entry(cur, struct frame, elem);
3.            move_cur_ptr();
4.            if (!f->pinned) {
5.              struct thread *t = f->thread;
6.              if (pagedir_is_accessed(t->pagedir, f->page->base))
7.                pagedir_set_accessed(t->pagedir, f->page->base, false);
8.              else {
9.                if (f->page->from_executable) {
10.                 // Case 1. The page is from an executable
11.                 ...
12.               }
13.               else if (f->page->is_mmapped) {
14.                 // Case 2. The page belongs to a mmap region
15.                 ...
16.               }
17.               else {
18.                 // Case 3.
19.                 swap_out(f->page);
20.               }
21.               ...
22.               break;
23.             }
24.           }
25.         }
```

The page replacement algorithm we implemented is based on the second-chance algorithm. The basic idea is that there is a FIFO list of active frames (frames that contain user pages). Whenever some frame must be chosen for eviction (because a frame is required but none is free), the algorithm will first look at the front of the FIFO list as the FIFO page replacement algorithm does *(line 2)*. However, instead of immediately evicting the frame, it checks to see if the referenced bit of the related page table entry is set *(line 6)*. If it is not set, the frame is evicted *(the else branch starting at line 8)*. Otherwise, the referenced bit is cleared *(line 7)*, the frame is inserted at the back of the queue (as if it were a new page) and this process is repeated *(this is done as the effect of line 3)*.

As illustrated by the code listing, whenever some frame has been chosen to be evicted, there are three possible cases.

1. Case 1. *The related page is from an executable.* If the page has not been modified since load, it will not be written to swap because it can always be read back from the executable. Otherwise, if the page has been modified, the content of the page will be written to swap.
2. Case 2. *The related page belongs to a mmap region.* This case is discussed in "MEMORY MAPPED FILES" part of this report.
3. Case 3. *The related page is not from an executable and also it does not belong to any mmap region (e.g., it is a stack page).* The content of the page will simply be written to swap.

Note that we won't consider a frame for eviction if the frame is pinned *(line 4)*. This is discussed in B8.

B3: When a process P obtains a frame that was previously used by a process Q, how do you adjust the page table (and any other data structures) to reflect the frame Q no longer has?

A process P obtains a frame that was previously used by Q only on the event of an eviction. The frame which was previously occupied by Q will be in swapped out to the disk. In the case of mmap, the pages written to process Q are written back to the file. After eviction, we clear the frame, set the appropriate page, set the occupying thread to P, use palloc_get_page to allocate a new base address.

```
f->page = p;
p->frame = f;
p->swap = NULL;
f->thread = thread_current(); //Set the thread that occupies frame to P
palloc_free_page(f->base);
f->base = palloc_get_page(flags);
```

B4: Explain your heuristic for deciding whether a page fault for an invalid virtual address should cause the stack to be extended into the page that faulted.

Whenever a page fault for an invalid virtual address occurs, we check three conditions to decide whether the page fault should cause the stack to be extended into the page that faulted or not.

- We look at the supplemental page table of the current process and check if the page containing the faulting address has never been mapped to any physical frame.
- Since the PUSHA instruction can cause a page fault 32 bytes below the stack pointer, we check if the faulting address is between esp-32 and PHYS_BASE.
- We check if the faulting address is above the end of the code and data segments (in the logical memory).

If all of the conditions are true, the stack should be extended.

---- SYNCHRONIZATION ----

B5: Explain the basics of your VM synchronization design. In particular, explain how it prevents deadlock. (Refer to the textbook for an explanation of the necessary conditions for deadlock.)

Necessary conditions for deadlock are:

- Mutual exclusion
- Hold and wait or partial allocation
- No pre-emption
- Resource waiting or circular wait

We have a static list of frames and a static bitmap of swapping slots.

To protect the frame list, we have a semaphore that we will wrap its “up” and “down” around allocate_frame().

To protect the swap table, we have a lock and a swap_disk_lock. Those 2 locks are never called at a same time, hence those 2 locks won't cause any deadlock (no thread can have a swap_disk_lock while waiting for lock).

Another possible deadlock case is when a thread P holds the lock of the swap table and request the semaphore of the frame table while a thread Q has the reverse. However, in our design, this will not happen because P has the lock of the swap table iff it is executing allocate_swap or free_swap, but allocate_frame is not called in those 2 functions.

For the same reason, deadlock won’t happen with the lock of the swap disk and the frame table.

B6: A page fault in process P can cause another process Q's frame to be evicted. How do you ensure that Q cannot access or modify the page during the eviction process? How do you avoid a race between P evicting Q's frame and Q faulting the page back in?

It is possible that a page fault in process P can cause another process Q’s frame to be evicted. In order to ensure that Q cannot access or modify the page during the eviction process, we clear the page before the eviction process actually begins. This is done by using the command *pagedir_clear_page*.

For each user page, we have a special semaphore associated with it (the member *page_sema* of the struct *page*). Whenever a process wants to change something related to the page (e.g., moving the content of the page from memory to the swap disk), it will need to first down the semaphore.

In this way, we have a good solution to avoid a race between P evicting Q’s frame and Q faulting the page back in. First, if P wants to evict a page of Q, it needs to down the semaphore *page_sema* of the page. P won’t up the semaphore until the eviction process is done. If Q wants to fault the page back

in, it also needs to down the semaphore and if the eviction process it not yet done, process Q has to wait. In this way, Q can only fault the page back in after the P has finished evicting the page.

B7: Suppose a page fault in process P causes a page to be read from the file system or swap. How do you ensure that a second process Q cannot interfere by e.g. attempting to evict the frame while it is still being read in?

We ensure that by using the “pinned” flag. In eviction, we will iterate through a list of frame and will only evict frame that is not pinned. So P will set it “pinned” flag to true during its read and write operation and unpin it once it is done.

```
//the code to iterate through the list of frame
while (true) {
      f = list_entry(cur, struct frame, elem);
      move_cur_ptr();
      if (!f->pinned) {...}
```

```
//system call read
int read (void * esp) {
  ...
  pin_pages(buffer, size);
  lock_acquire(&filesys_lock);
 ...
  lock_release(&filesys_lock);
  unpin_pages(buffer, size);
  return result;
}
```

```
//system call write
int write (void * esp) {
  ...
  pin_pages(buffer,size);
  ...
  unpin_pages(buffer,size);
  return result;
}
```

B8: Explain how you handle access to paged-out pages that occur during system calls. Do you use page faults to bring in pages (as in user programs), or do you have a mechanism for "locking" frames into physical memory, or do you use some other design? How do you gracefully handle attempted accesses to invalid virtual addresses?

During a system call, we first bring in all the pages to be needed by the system call but currently not in memory. Then we lock every frame containing a page needed by the system call into physical memory by using the pinned flag. This prevents the frame table from evicting the frame. If a virtual

address is invalid, the user process will be terminated with the exit command (with an exit status -1). All pages will be freed to prevent memory leaks (more specifically, this is handled inside the function *process_exit* defined in the file *process.c*).

---- RATIONALE ----

B9: A single lock for the whole VM system would make synchronization easy, but limit parallelism. On the other hand, using many locks complicates synchronization and raises the possibility for deadlock but allows for high parallelism. Explain where your design falls along this continuum and why you chose to design it this way.

We used more than one synchronization primitive to implement the VM system. Our VM system has more parallelism than a VM system implemented by using a single lock. For example, there are many possible situations where our VM system handles more than one page fault concurrently. However, for VM system implemented by using a single lock, if there is more than one page fault at the same time, the page faults must always be handled sequentially, one after the other.

We probably can make our VM system to have more parallelism. For example, as discussed before, we used some semaphore to make sure that the function *allocate_frame* is being executed by at most one process at a time. However, consider the situation where there are still many free frames in the physical memory. So it seems that it should be possible for more than one process to execute the function *allocate_frame* concurrently.

Nevertheless, we think that our current VM system has the right balance between complexity and parallelism. Therefore, we decide not to opt for more parallelism.

MEMORY MAPPED FILES

==========

---- DATA STRUCTURES ----

C1: Copy here the declaration of each new or changed `struct' or 'struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

In vm/page.h and vm/page.c:

enum data_location {
  NONE,
  MEMORY, // in memory
  SWAP, // in a swap slot
  EXECUTABLE, // in an executable
  MMAP // in a memory-mapped file
};
	This enumeration represents the places where the content of a page can be at (e.g., MEMORY means the page's content is currently in memory).

struct page {
  struct hash_elem hash_elem;
  tid_t pid;
  void * base;
  bool writable;
  bool from_executable;
  bool is_mmapped;
  enum data_location loc;
  struct swap * swap;
  struct frame * frame;
  struct file * mmappedfile;
  off_t mmapped_ofs;
  off_t ofs;
  size_t page_read_bytes;
  struct semaphore page_sema;
};
	Each instance of this struct represents a page of some user process. It contains all important information about a page.

In syscall.c

struct file_mapping_info {
  mapid_t mid;
  struct file * file;
  void * addr;
  struct list_elem elem;}
	Contains important information about a mapped file

static struct list file_mapping_info_list;	To keep track of the usage of the mapped files
struct lock mm_lock;	To protect the aforementioned global file_mapping_info_list
struct lock mid_lock;	To protect the global counter of mid

---- ALGORITHMS ----

C2: Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages.

Like pages for executables, pages for memory mapped files are loaded lazily, that is, only as the kernel intercepts page faults for them. When a page fault occurs due to a page for a memory mapped file, we will just simply read the page from the file into memory. This is quite similar to how we handle page fault that occurs due to pages for executables.

During eviction, a page for a memory mapped file will be written to its original file if the page is dirty. However, if a page for executable is evicted and the page is dirty, it should be written to swap instead.

C3: Explain how you determine whether a new file mapping overlaps any existing segment.

```
1.  int filelength = file_length(myfile);
2.  ...
3.  int nbofpages = filelength / PGSIZE;
4.  if (filelength % PGSIZE > 0)
5.    nbofpages++;
6.  ...
7.  void * tmp_addr = addr;
8.  for (i = 0; i < nbofpages; i++) {
9.    if(find_page(tmp_addr)) {
10.     ...
11.     return -1;
12.   }
13.   tmp_addr += PGSIZE;
14. }
```

First, we calculate the size of the file *(line 1)*. Then we calculate the number of memory pages needed for the file mapping *(lines 3 and 4)*. Now suppose that we need n pages in total. Also, let addr denotes the starting address of the mapping. Then, by looking at the supplemental page table, we check if any of addr, addr + PGSIZE, …, addr + PGSIZE * (n - 1) has ever been mapped to any frames *(the for loop starting at address 8)*. If any of the virtual addresses has been mapped to some frame, we know that the new file mapping will overlap with some existing segment. In this case, the system call should return -1 *(line 11)*, which indicates failure.

---- RATIONALE ----

C4: Mappings created with "mmap" have similar semantics to those of data demand-paged from executables, except that "mmap" mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain you're your implementation either does or does not share much of the code for the two situations.

We do understand that mappings created with "mmap" have very similar semantics to those of data demand-paged from executables. However, we have decided not to make our implementation share much of the code for the two situations because we want to have better code clarity. In our current code, we can quickly identify which parts are for mappings created with “mmap” and which parts are for pages from executables. However, if we did try make our implementation share code for the two cases, the code could be a little bit shorter but likely to be confusing.