Project 3b: Virtual Memory

Preliminaries

> Fill in your name and email address.

Yuxing Xiang 2000012959@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

- The previous submission already passes all tests in Lab3, and we pretty much introduced everything about our VM system in the design document for Lab3a. Therefore, we focus on what's new for the implementing the tasks in Lab3b only. For a thorough illustration of our VM system, please check out the previous design document.

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Stack Growth

ALGORITHMS

> A1: 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.

First, since we build the **Supplemental Page Table (SPT)** based on VMAs, we can describe stack growth by decreasing `vma->lower_bound`. However, to avoid troubles when stack growth overlaps with other VMA, we just set the VMA of stack area to `STACK_PG_CNT` pages **at loading**, namely the maximal number of pages for one process stack. Do note that stack pages are still zeroed on demand; it's just we don't really grow the VMA in our implementation.
(Linux also adopts a similar method to ours, in that the entire 8MB space starting from stack bottom is just readily pre-marked as stack. Moving `%esp` within that space is not considered hazardous. The `pt-grow-bad` test, for example, would be tolerated in Linux.)

The stack growth heuristic, then, is put in the PF handler code for allocating a new zeroed frame. By locating VMA for `fault_addr`, we easily tell if the access is within stack (which, since VMAs are sorted in terms of virtual address in increasing order, is always the last VMA), then compare `fault_addr` with process's `esp` value to see if that access **seems** valid. `fault_addr` must (1) be not smaller than `esp` (2) equals `esp - 4` (for `push` instructions) (3) equals `esp - 32` (for `pusha` instructions) to be accepted by our heuristic. Otherwise, the access is treated as invalid, and the process gets killed.

Memory Mapped Files

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.

We repeat below the relevant data structures for implementing MMAPs.

```
/* ----------In mm.h---------- */

/** The abstraction for a mapped area in a process.
    There can be multiple VM_AREAs for one process, chained by its
    vma_list member in struct thread. Upon loading, one VM_AREA is
    created for each segment. MMAP syscall also creates new VM_AREAs.
    The VMA holds lower and upper bound for valid virtual addresses
    in the area. It also records type of the area: file-backed or
    anonymous; in the former case, VMA contains pointer to a MAP_FILE
    struct and the offset into the backing file. */
struct vm_area
{
  /**< The lower bound of valid virtual address. */
  void *lower_bound;
  /**< The upper bound of valid virtual address. */
  void *upper_bound;
  /**< List element for owner process. */
  struct list_elem proc_elem;
  /**< Owner process of this VMA. */
  struct thread *proc;

  /**< Flags: private or shared, read-only or writable. */
  int flags;

  /* Only relevant for file-backed area. */

  /**< Pointer to backing file. NULL indicates anonymous area. */
  struct map_file *mapfile;
  /**< List element for map_file struct. Also used by frame_reverse_map(). */
  struct list_elem mapfile_elem;
  /**< Offset into backing file. */
  off_t offset;
  /**< Size of file-segment. */
  uint32_t filesize;
};


/** Describes a unique memory mapped file (called address_space in
    Linux).
    For each different running executable and MMAPed file, exists
    one MAP_FILE. Sharing is essentially implemented by VMAs in different
    processes pointing to the same MAP_FILE. */
struct map_file
{
  /**< The backing file pointer. */
  struct file *backing_file;
  /**< Marks the end of the mapped area of the file. */
  off_t file_end;
  /**< Whether the backing file is writable. */
  bool writable;

  /**< Hash element for global pool of MAP_FILEs. */
  struct hash_elem elem;
  /**< Inode pointer for backing file. */
  void *inode;

  /**< The list of VMAs that point to this struct. */
  struct list vma_list;
  /**< Contains all data pages from the file that are currently in memory. */
  struct hash page_pool;

  /**< Lock for page_pool. */
  struct lock page_pool_lock;
};


/* ----------In mm.c---------- */
/** Global pool of existing MAP_FILEs. */
struct hash mapfile_pool;

/** Lock protecting the pool of MAP_FILEs. */
struct lock mapfile_pool_lock;


/* ----------In mapfile.h---------- */
/** Flags for VM mappings. */
/**< Private mapping, with COW. */
#define MAP_PRIVATE 1
/**< Shared mapping, by using the same physical frame. */
#define MAP_SHARED  2
/**< Read-only mapping. */
#define MAP_READ       4
/**< Writable mapping. */
#define MAP_WRITE   8
```

ALGORITHMS

> B2: 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.

The MMAPed files integrate seamlessly into our VM system, by respectively corresponding to **MAP_FILE instances**. Every valid MMAP syscall creates one VMA that describes the virtual area, which is attached to one MAP_FILE instance in the global `mapfile_pool`: either the file is mapped for the first time, and a new MAP_FILE instance is created, or we locate an existing instance and attach onto it. MUNMAP syscalls, on the other hand, frees one VMA attaching to a MAP_FILE and, if the MAP_FILE becomes orphaned, also frees the MAP_FILE instance.

The VM system is then **agnostic** about the differences between MMAPed files and loaded executables. If an address within MMAPed area faults, the data page is read in from the backing file recorded in the MAP_FILE instance. By making multiple VMAs pointing to one MAP_FILE, the MMAPed area is essentially **shared** between each MMAP call to the same file.

The only deviation happens at eviction: for loaded executables, its MAP_FILE is marked as **non-writable**, and its frames are not possibly dirty (COW ensures modified data/bss pages are anonymous, which are written to swap), so their contents are just discarded. On the other hand, MMAPed files have their MAP_FILEs set to be **writbale**, and the dirty bits are looked through in **reverse mapping** to decide whether the frame content needs writing back.

It should be obvious that our VM system can be easily extended to support configurable MMAP syscall: MMAPed areas can also be read-only, or privately writable (ensured by COW).

> B3: Explain how you determine whether a new file mapping overlaps any existing segment.

VMAs describe valid ranges of user virtual addresses, and we can just search the `vma_list` of a process to see if the wanted mapping overlaps with existing areas.

```
/* Fail if [ADDR, ADDR + SIZE] intersects with existing VMAs. */
struct list_elem *e;
for (e = list_begin (&cur->vma_list); e != list_end (&cur->vma_list);
     e = list_next (e))
{
  struct vm_area *vma = list_entry (e, struct vm_area, proc_elem);
  if (vma->lower_bound < addr + size)
  {
    if (vma->upper_bound > addr)
      /* Found overlapping. */
      goto bad_mmap;
  }
    else
      /* No overlapping. */
      break;
}
```

RATIONALE

> B4: 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 why your implementation either does or does not share much of the code for the two situations.

As is hinted previously, our implementation of VM system is agnostic about MMAPed mappings and loaded executables, excepting only for the top-level `load()` and `syscall_mmap()` APIs. During our designing, we made sure to take all the requirements in Lab3 (and even other semantics in standard Linux that is not required in Pintos) into account, and wrote the code to be compactible across Lab3a and Lab3b.

Despite difficulty in designing, we don't see why demand paging from executables and MMAPed files should differ greatly in their implementation. The only obvious requirement that deviates between them is the eviction process, but can be handled straightforwardly, by checking dirty bits and some if-blocks for doing I/O. What is achieved, however, is a general-purpose VM system with complete sharing. Sounds like good deal to us!