Project 3b: Virtual Memory

Preliminaries

> Fill in your name and email address.

Yixin Yang (杨艺欣) yangyixin@stu.pku.edu.cn

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

All tests passed (tested for > 5 times).

> 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.

Each thread has a fixed stack size of `STACK_SIZE` bytes (256 KB).

On a page fault, page fault handler will check if `%esp` is within the range of user stack (`PHYS_BASE - STACK_SIZE` to `PHYS_BASE`). If it is, we further check if the faulting address:

- is within the region `[%esp, PHYS_BASE)` or
- equals to `%esp - 4` (PUSH instruction) or
- equals to `%esp - 32` (PUSHA instruction).

If `%esp` is within the user stack range and the faulting address is one of the above, this page fault will be considered a stack growth fault.

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.

```
/* vm/vm_region.h */
/* REGION_EXEC: Executable region
   REGION_ZERO: Zeroed region
   REGION_MMAP: Memory-mapped region */
enum region_type { REGION_EXEC, REGION_ZERO, REGION_MMAP };

/* VM region list entry */
struct vm_region
  {
    struct list_elem l_elem;       /* List element */
    enum region_type type;         /* Type of the region */
    upage_t start;                 /* Start address of the region */
    size_t length;                 /* Length of the region */
  };

/* Create and insert a new VM region entry into the list. */
bool vm_region_install (struct list *vm_region_list,
                        enum region_type type, upage_t start, size_t length);

/* Destroy the VM region list, freeing all VM region entries. */
void vm_region_destroy (struct list *vm_region_list);
```

```
/* Check if the given region is not overlapping with any existing region
   in the VM region list. */
bool vm_region_available (struct list *vm_region_list,
                          upage_t start, size_t length);

/* Remove a VM region entry from the list. */
void vm_region_uninstall (struct list *vm_region_list, upage_t start);


/* vm/mmap.h */

typedef int mapid_t; /**< Mapping ID type. */

/* mmap table entry */
struct mmap_entry
  {
    struct list_elem l_elem;       /* List element */
    mapid_t mapid;                     /* Mapping ID */
    struct file *file;             /* File being mapped */
    upage_t uaddr;                 /* User virtual address */
    size_t page_cnt;             /* Number of pages mapped */
  };
```

```
/* Create and insert a new mmap entry for the given file. */
bool mmap_create (struct list *mmap_list, mapid_t mapid,
                  struct file *file, size_t length, upage_t uaddr);

/* Write back all mmap entries and destroy the mmap list. */
void mmap_write_back_and_destroy (struct list *mmap_list);

/* Write back mmap entry to file, and remove it from the list. */
void mmap_write_back_and_delete (struct list *mmap_list, mapid_t mapid);


/* vm/mmap.c */
/* Lookup mmap entry by mapid (panic if not found). */
static struct mmap_entry *mmap_lookup (struct list *mmap_list, mapid_t mapid);

/* Write back a mmap entry to file. The file is closed after writing back,
   and the corresponding vm region will be uninstalled. */
static void write_back_mmap (struct mmap_entry *entry);

/* vm/page.h */
/* Supplemental page table entry */
struct spt_entry
```

```
  {
    /* ... */
    bool write_back;    /* Should write back to file when swapped out.
                           Used to tell executable-backed pages and mmap pages
                          apart. */
    /* ... */
  };

/* Create and insert new supplemental page table entries for multiple
   contiguous file-backed pages. */
bool spt_install_file_pages (struct file *file, off_t ofs, upage_t upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable,
              enum region_type type);

/* Create and insert a single supplemental page table entry for
   file-backed page upage.
   Helper function for spt_install_file_pages (). */
static bool spt_install_file_page (struct hash *spt, struct lock *spt_lock,
                           upage_t upage, struct file *file, off_t ofs,
                           size_t read_bytes, size_t zero_bytes,
                           bool writable, bool write_back)

/* Swap out a dirty page to swap space (PAGE_SWAP) or its backing file
   (PAGE_FILE with write_back == true). */
void spt_swap_out_page (struct hash *spt, struct lock *spt_lock,
                        upage_t upage, kpage_t kpage);
/* Removed spt_set_page_swapped (). Its logic is moved to spt_swap_out_page (). */

/* threads/thread.h */
typedef void * uaddr_t;  /**< Type for user virtual addresses. */
struct thread
  {
    /* ... */
    uaddr_t esp;                        /**< Stack pointer. */
    /* ... */
  };

/* userprog/process.h */
struct proc_info
  {
    /* ... */
    struct list mmap_list;          /**< List of mmap entries. */
    mapid_t mmap_next_mapid;        /**< Next mapid to allocate. */
    struct list vm_region_list;     /**< List of VM regions. */
  };

/* userprog/process.c */
/* load_segment () is refactored into spt_install_file_pages (), declared in
   vm/page.h. */
/* lazy_load_page () is removed to improve clarity. It is introduced in my
   Lab 3a code. */

/* userprog/syscall.c */
/* System calls for mmap and munmap. */
static mapid_t syscall_mmap (int fd, void *addr);
static void syscall_munmap (mapid_t mapid);

/* userprog/exception.c */
/* Check if the page fault is a stack growth fault. Extend the stack if it is. */
static void extend_stack (void *fault_addr, bool user, void *esp)
```

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.

Thanks to the good design of my Lab 3a code, I implemented mmap with minimal modifications to the existing code for demand paging and swapping in/out.

Pages backed by mmap files are treated nearly the same as pages backed by executable files. They are both marked as `PAGE_FILE` in the supplemental page table, and the only difference is that mmap pages' `write_back` field is set while executable pages' not. When evicted, executable pages will be written back to swap space, and its type will be set to `PAGE_SWAP`, while mmap pages will be written back to its backing file, with its type preserved as `PAGE_FILE` (assuming dirty). There are no differences in the page fault process between mmap pages and executable pages. Both types of pages will be loaded from their corresponding files (executable or mmap) on their first access.

The differences between swap pages and other pages (specifically, zeroed pages and file-backed pages) are:

- File-backed pages and zeroed pages are pages to be lazy-loaded from their corresponding files (file-backed) or lazy-filled with zeroes (zeroed) on their first access, while swap pages are those already loaded into physical memory before but evicted to swap space now. Swap pages will be loaded back from swap space.
- When being loaded back into physical memory, swap pages will be marked as dirty (since if not, they would not be swap pages in the first place). This is the same with zeroed pages, but different from file-backed pages.
- In eviction process, mmap pages will be written back to their backing files, preserving its page type as `PAGE_FILE`, while all other pages (executable pages, zeroed pages, and swap pages) will be written back to swap space, with their page type set to `PAGE_SWAP`, as already explained above.

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

I maintain a list of occupied VM regions for each process. There are three types of VM regions:

- `REGION_EXEC` : Executable region
- `REGION_ZERO` : Zeroed region
- `REGION_MMAP` : Memory-mapped region

On each `mmap` system call, the VM region list will be iterated to see if the memory region to be mapped overlaps with any existing region (including the user stack segment, segments corresponding to the loaded executable, and other mmap regions).

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.

My implementation shares most of the code for mmap and executable pages. The reasons are as follows:

- I enjoy writing clean and elegant code. I hate poorly written code.
- They are essentially the same type of pages, with the only difference being that mmap pages are written back to their backing files while executable pages are written back to swap space. It is unreasonable to treat them as different page types since the way they are installed into the supplemental page table and the way they are loaded are exactly the same.
- The DRY principle in software engineering states that repetitive code will only lead to bugs and maintenance problems. Not sharing the code may, at first glance, seem to be less error-prone as it does not introduce new bugs in existing code. However, it is very likely that after several refactorings, code that should be the same gets diverged, leading to unexpected bugs and poor clarity.