CS 130: Project 3 - Virtual Memory Design Document

Group Information

Please provide the details for your group below, including your group's name and the names and email addresses of all members.

- **Group Name**: *[printf("team")]*
- **Member 1**: Zirui Wang <wangzr2024@shanghaitech.edu.cn>
- **Member 2**: Xing Wu <wuxing2024@shanghaitech.edu.cn>

Preliminaries

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

None.

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.

New Struct Member to `threads/thread.h`

```
struct thread
{
  /* ... */

  #ifdef VM
    struct list page_list; /* Pages a user process owns. */
    void *user_esp; /* Stores esp on the transition from user to kernel mode */
  #endif

  /* ... */
};
```

New Struct to `vm/frame.h`

```
struct frame_owner
{
  void *upage;               /* Address returned to user. */
  struct thread *thread;     /* Owner thread of this frame. */
  struct page *sup_page;     /* The corresponding supplemental page. */
  struct list_elem listelem; /* The list element in owner_list. */
```

```
};

struct frame
{
  void *kpage;               /* Address returned by palloc. */
  bool pinned;               /* Whether this frame can be evicted. */
  struct hash_elem hashelem; /* The hash element in frame_hash. */
  struct list_elem listelem; /* The list element in frame_list. */
  struct list owner_list;    /* List of owners of this frame. */
};
```

New Static Variable to `vm/frame.c`

```
/*Frame table. */
static struct lock frame_lock;      /* A lock to protect frame table. */
static struct hash frame_hash;      /* A hash table to store frames. */
static struct list frame_list;      /* A list to evict frames.*/
static struct list_elem *clock_ptr; /* Still used to evict frames. */
```

New Struct and Enumeration to `vm/page.h`

```
/* Types of a page, see comments of PAGE for more detail. */
enum page_type
{
  PAGE_UNALLOC, /* Unallocated anonymous page. */
  PAGE_ALLOC,   /* Allocated anonymous page. */
  PAGE_FILE     /* Memory mapped page. */
};

/* Page is a kind of abstraction to the memory a user program uses.

   A page can be one of difference types:
   - UNALLOC: unallocated anonymous page. it will be allocated when accessed.
   - ALLOC: allocated anonymous page. it is not backed by any file. it might be
     in the swap slot if evicted.
   - FILE: file backed page. it might be in the filesys if evicted or unloaded.
 */
struct page
{
  enum page_type type; /* Page type. */
  void *kpage;         /* Frame mapped to this page, if it exists. */
  slot_id slot_idx;    /* Slot store this page, if it exists. */

  /* We need some metadata to load the correct data from file. */
  struct file *file;
  off_t ofs;
  void *upage;
  uint32_t read_bytes;
```

```
  uint32_t zero_bytes;
  bool writable;

  struct hash_elem hashelem; /* Used by sup_hash_table. */
  struct list_elem listelem; /* Used by page_list in struct thread. */
  struct thread *owner;      /* The thread that own this page. */
};
```

New Static Variable to `vm/page.c`

```
/* The supplemental page table. */
static struct hash sup_page_table;

/* The lock for the supplemental page table. */
static struct lock sup_page_table_lock;
```

New `typedef` to `vm/swap.h`

```
/* The swap partition is split into small slots. This is the unique identifier
   for each slot. */
typedef size_t slot_id;
```

New Static Variable to `vm/swap.c`

```
/* The swap partition. */
static struct block *swap_device;

/* We use a bitmap to track allocated slots. */
static struct bitmap *swap_bitmap;

/* Use a lock to protect the swap bitmap. */
static struct lock swap_lock;
```

Algorithms

> **A2:** In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

We use a hash table to lookup the data we need. To uniquely determine that page, we need a virtual address and its owner thread.

This instruction has been encapsulated into a function `get_page()`, which first executes `page_round_down()` to normalize the address, then acquires the lock, and finally quires the hash table.

> **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 mainly use user virtual address to avoid confusion.

Synchronization

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

The frame table is protected by `frame_lock`, which makes inserting or deleting atomic.

Rationale

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

We chose a hash table for the supplemental page table because it has advantage in time complexity over list.

The frame list paired with a clock pointer is used for the eviction algorithm.

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.

None.

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.

We implement the clock algorithm by maintaining a circular linked list of all frames and a clock pointer. When selecting a victim, we examine the frame pointed to by the clock pointer: if its accessed bit is 1, we clear the bit and advance the pointer to the next frame; if its accessed bit is 0, we choose that frame as the victim. In this way, we approximate LRU performance at low overhead.

> **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?

The old frame is freed, and it will be unbounded from its corresponding supplemental page. Then a new frame will be allocated with a different owner and be bounded to P's supplemental page.

The two frames have different owner, so they won't be considered the same.

> **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.

If the fault address lies within 8 MB below the top of the stack, no more than 32 bytes below the user stack pointer, and is not bounded to any pages. Then we decides to grow the user stack.

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.)

- To protect a frame across system calls, we raise a `pinned` flag on the frame rather than holding a lock, which breaks the "hold-and-wait"condition for a deadlock.
- If necessary, use static locks instead of global locks. However, `filesys_lock` is global because filesystem instructions naturally span multiple modules. We consider it safer to limit the scope of locks.

> **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?

We use the `frame_lock` to ensure the evict instruction atomic.

> **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, for example, attempting to evict the frame while it is still being read in?

When a page is still being read from the file system or swap, it's frame is set to be "pinned". It won't be evicted until reading is finished.

> **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?

We always try to full load the the page. If it fails, it means that the address is invalid, then it's time to terminate the process.

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. Explain where your design falls along this continuum and why you chose to design it this way.

A single, global VM lock would be easy to implement but would serialize all instructions, severely limiting concurrency. At the opposite extreme, if we protect every data structure or even each page with its own lock, the code might be too complicated to understand.

Our design split the VM system into 3 subsystems, each has its own lock: `frame_lock`, `sup_page_table_lock` and `swap_lock`. In fact, we think the synchronization problem in VM system can be classified by the readers- writers problem, and conditional variables can also be a solution. But locks seems to be a more natural method to control synchronization.

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.

New Struct Member to `threads/thread.h`

```
struct thread
{
  /* ... */

  #ifdef VM
    mapid_t mapid_next; /* Next mapid for this process. */
  #endif

  /* ... */
};
```

New Struct to `userprog/sycall.h`

```
struct mmap_data
{
  mapid_t mapping;           /* Unique identifier within a process */
  struct file *file;         /* File mapped to this memory segment. */
  struct hash_elem hashelem; /* The hash element in mmap_table. */
  tid_t owner;               /* Owner process of this mapping. */
  void *uaddr;               /* Begin of mapped memory address. */
};
```

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.

The memory mapped data can use the supplemental page table to lazy allocate memory. The difference is that the frame needs to be written back to the file system if evicted.

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

A file may use memory in `[addr, addr + file_len)` (page-aligned), we iterate through this range, check wether a page has already used this address.

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

Most of their implementation are shared, such as `page_lazy_load()`, `page_full_load()`, `page_free()`. Hoever, it's true that a page from `load_segment()` and `mmap()` differs, and we use `enum page_type` to identify them.

Survey Questions

> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Any other comments?

None.