CS130 Project 3 Design Document

Group16

- Cunhan You: youch@shanghaitech.edu.cn
- Junda Shen: shenjd@shanghaitech.edu.cn

Reference:

> Pintos Guide by Stephen Tsung-Han Sher

Task 1: Page Table Management

Data Structure

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`

```
struct suppl_page_table_entry
{
  struct file *file;            /* File to load */
  int type;                     /* Type of the file */
  void *addr;                   /* User virtual address, key to he hash table */
  off_t offset;                 /* File offset */
  uint32_t read_bytes;          /* Bytes to read from file after offset */
  uint32_t zero_bytes;          /* Bytes to be zeroed, after read bytes */
  bool writable;                /* Whether the page is writable */
  bool free;                    /* False if the page hasn't been loaded */
  size_t swap_idx;              /* Index on swap bitmap returned by swap_out()
*/
  int64_t timestamp;            /* Creation and accessing time */
  struct lock spte_lock;        /* Lock in case synchronization */
  struct hash_elem hash_elem;   /* Hash table element */
};
```

In `threads/thread.h`

```
struct thread {
  struct hash suppl_page_table; /* The SPT which stores some information about a
page */
}
```

Algorithms

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

In `load_segment()`, we create a suppl page table entry for every page, and store some information in it, the user virtual address, the file pointer, the offset of the file, the number of bytes read, writability, free or not, load/use time(used for LRU_evict). Then we assign it to the suppl page table of the current process. This is called lazy load.

When a page fault occurs, we get the fault address and look for it in the suppl page table of the current process. If we can't find it, exit. Otherwise, we find the suppl page table entry and load the page, allocating a page, reading the file at the offset, and filling the rest part of the page to 0. Then we install this page.

When a page fault occurs on stack growth, we get the fault address and allocate a page for it, and then install this page.

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 this by accessing only through user virtual addresses.

Synchronization

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

In `vm_get_frame`, we use `frame_lock` to ensure that adding new frames is an atomic operation.

Rationale

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

We use a linked list for frame table. Inserting (allocate frame) to or removing (evict frame) from the table takes constant time. However, finding a frame needs $O(n)$ time for a linked list, but we only need to iterate over the whole list when performing eviction and free operations, these iteration will not use too much time.

We use a hash table for supplemental page table, with the user virtual address as the key. Since a page fault only gives a fault address, the hash table is fast to find the SPT entry given the address (takes constant time on average).

Task 2: 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/frame.h`

```
struct list frame_table;

struct frame_tab_entry
{
  void *frame;                              /* Actual frame addr */
  struct suppl_page_table_entry *spte;    /* Corresponding
suppl_page_table_entry */
  pid_t owner;                              /* Used to identify the owner
thread/process */
  struct list_elem elem;
};
```

In `vm/swap.c`

```
static struct block* global_swap_block; /* The block of the swap disk */
static struct bitmap *swap_bitmap;          /* The bitmap, store the info
whether a swap slot is used or not */
static struct lock swap_lock;           /* The lock for synchronization */
```

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.

Whenever a page is changed (used or created), we update the access time (saved as `timestamp` in `frame_tab_entry`) to the current timer ticks.
We use a linked list to record all the frames we allocated.
When a frame need to be evicted:

1. We search all the frame_tab_entry to find an unaccessed page, if it is found successfully, it is chosen to be evicted.
2. If we can't find such an unaccessed page, we choose a least recently used (`LRU`) page by finding the longest access time, which means it is used earliest.

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?

We remove the frame_tab_entry corresponding to process Q, and add a frame_tab_entry corresponding to process P. Then in frame table, it means that this frame belongs to process P instead of process Q.

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 (spte == NULL && fault_addr >= PHYS_BASE - STACK_LIMIT && fault_addr >= f-
>esp - 32)
{
  stack_grow (fault_addr);
}
```

1. If the SPT entry is not found (`NULL`), the page fault could be caused by stack growth.
2. If the fault address is within the stack size, (larger than `PHYS_BASE - STACK_LIMIT`), it is either invalid, or valid `PUSH` or `PUSHA` instructions.
3. We check if this `fault_addr` is larger than `esp - 32` to ensure this is `PUSH` or `PUSHA`. An address that does not appear to be a stack access, meaning the virtual address requested does not to appear to be the next contiguous memory page address of the stack (within 32 bytes of the stack pointer) is rejected.

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.)

In project2, we use a `file_lock` to guarantee that all file operations is atomic.

In project3, we also need file operations, so we use the same `file_lock`.

In `swap_in()` and `swap_out()`, every time we use bitmap, we use a `swap_lock` to ensure that no other process can modify the bitmap at the same time.

When we do operations of the frame table, such as get frame, evict frame, we use a `frame_lock` to ensure that only one process is modifying 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?

When P is evicting one frame, we acquire `spte_lock` to ensure that Q can't modify or read the frame.

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?

When P is causing a page to be read from the file system or swap, we acquire `spte_lock` to ensure that Q can't evict this page.

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?

In syscall, we check the validation of function arguments (which could be buffers or strings) before entering the syscalls functions and accessing them. if not, `exit(-1)`.

Especially in `read()` syscall, when reading to buffer, we should avoid page fault, we deal with it at syscall level, as to handle it before the page fault handler gets it. We also ensure that all the file operations are atomic by using `file_lock`.

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 use `frame_lock` to ensure the frame table can not be modified by multiple threads, especially when evicting frames.

We use `swap_lock` to ensure the operation in bitmap must be atomic, since swap_in/out can't be called by multiple threads at the same time.

We split them instead of using a global lock to make swap operation and evict operation can happen at the same time for high parallelism.

But for `spte_lock`, it should be used both in evicting and getting a frame, since many operations will deal with `suppl_page_table_entry`, we should ensure that these operations will not change/read supplemental page table at the same time.

Task 3: 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/mmap.h`

```
struct mmap_entry
{
  mapid_t mmap_id;         /* The map id */
  void *uvaddr;            /* The user virtual address */
  struct file *file;       /* The pointer of the mapped file */
  unsigned int page_num;   /* Number of pages in this map */
  struct list_elem elem;   /* The hash_elem used to become an elem in a hash
table */
};
```

In `threads/thread.h`

```
struct thread {
  struct list mmap_list;   /* List of mmap file descriptors. */
  int mmap_num;            /* Number of mmap files. */
}
```

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.

When a user program call `mmap`:

1. Check the argument.
2. Reopen a new file pointer.
3. Malloc an `mmap_entry` to store required information. Increase `mmap_num` to ensure every new `mmap_entry` will have a unique `mmap_id`.
4. Call `page_lazy_load()` to create new `suppl_page_table_entry` for this `mmap` file, and insert them to the SPT.
5. Push the `mmap_entry` to `mmap_list` and return the corresponding `mmap_id`.

When a user program call `munmap`:

1. We will iterate the `mmap_list` and free all the related resources then delete it from the `mmap_list`.

For page fault:

1. For swap pages, the data is obtained from swap slots, but for other pages, data is obtained from files.

For eviction:

1. There is no difference between swap pages and other pages when deciding which frame to evict.
2. The type of any evicted page will be changed to `type_swap after` eviction.
3. For `mmap` files, if its pages are dirty, we will write its data back to file.

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

Check if the user address starting from `addr` to `addr + file length` is already mapped by other pages or there is already a page table entry occupying that address space.

If one of the two situations happens, we can determine it overlaps, so we `return -1` to fail.

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.

1. In `suppl_page_table_entry`, they share all the member values except `spte_type`. This slight difference is demanded during eviction, we need to write back mmap file content to their original files. For data demand-paged from executables, they are written to swap area.
2. We share `page_lazy_load()` and `page_load_file()`.

Contributions

Cunhan You:

- mmap, munmap
- swap
- page
    - load file/swap
    - lazy load

Junda Shen:

- frame
    - get frame
    - evict frame
    - Synchronization

- page
    - stack grow