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**: **lorin&lucy**
- **Member 1**: Jintong Luo `<luojt2022@shanghaitech.edu.cn>`
- **Member 2**: Lufei Li `<lilf2024@shanghaitech.edu.cn>`

Preliminaries

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

No.

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.

Your answer here.

In `vm/frame.h`:

```
struct lock frame_lock;              /* Lock to synchronize between
processes on frame table. */
static struct list frame_table;     /* The frame table. */

/* This is the structure of page frame. */
struct frame
{
    void* kpage;                    /* Kernel virtual address. */
    struct thread* holder;          /* The thread that holds the frame. */
    struct page* page;              /* The related page. */
    struct list_elem frame_elem;    /* List element to iterate over. */
    bool pinned;                    /* Record when a page contained in a
frame
                                       must not be evicted. */
};
```

In `vm/page.h`:

```
enum page_type {
    PAGE_DEFAULT,
    PAGE_FILE,
    PAGE_MAPPED
};

/* This is the structure of the supplemental page table entry. */
struct page {
    void* vaddr;                    /* The user virtual address. */
    enum page_type page_type;       /* The type of supplemental page entry.
*/

    bool writable;                  /* Whether the page is writable. */
    bool is_loaded;                 /* Whether the page is loaded in
memory. */
    int swap_index;                 /* Set it to -1 when unswapped. */

    /* The corresponding file. */
    struct file* file;
    off_t offset;
    size_t read_bytes;
    size_t zero_bytes;

    struct hash_elem hash_elem;     /* struct thread `page_table' hash
element. */
};
```

In `threads/thread.h`:

```
struct thread {
    struct hash* page_table;        /* page table. */
};
```

Algorithms

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

1. ***Locate the SPT Entry**: Given a user virtual address that triggered the page fault or is being accessed, using the user virtual address as the key to search the SPT hash table to locate the corresponding entry in the SPT.*
2. ***Check Page Type and Status**: Once the SPT entry is found, the type of the page is checked to determine how the page should be handled.*
3. ***Load the Page into Memory**: The `is_loaded` flag in the SPT entry indicates whether the page is currently loaded into memory, i.e. a frame. If the page is not loaded, we need to load the page into memory. Once the page is loaded into a frame, the `is_loaded` flag is set to `true`, and the frame information is updated in the SPT entry.*

4. ***Update Page Table**: After the page is loaded into a frame, the process's page table is updated to map the user virtual address to the physical frame.*

> **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?

Our design avoids direct manipulation of these bits in the kernel. Instead, it ensures that the kernel always accesses user data through user virtual addresses, leveraging the existing page table mechanisms to maintain consistency.

Synchronization

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

A global lock named `frame_lock` is used to protect access to the frame table. This lock ensures that only one process can allocate or free a frame at any given time. Before any function calls to acquire or modify the frame we try to acquire the lock and release it as soon as the process finishes its tasks.

Rationale

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

The SPT is implemented as a hash table, with each entry corresponding to a user virtual address. This allows for efficient lookups, insertions, and deletions. In addition, hash tables can handle a large number of entries efficiently, which is important for systems with many processes and pages.

The frame table is implemented as a linked list (`struct list`), with each frame represented by a `struct frame` containing associated information. Linked lists allow for easy traversal and ordering, which is useful for implementing eviction policies. Frames can be dynamically added or removed from the list as they are allocated or freed.

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`:

```
/* The number of sectors per page. */
#define SWAP_SECTOR (PGSIZE / BLOCK_SECTOR_SIZE)

struct bitmap* swap_table;      /* A bitmap to identify whether a swap slot
is available. */
struct block* swap_block;       /* The swap block. */
struct lock swap_lock;          /* Lock to synchronize access to the swap
space. */
```

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. ***Select Frame to Evict***:*The `get_frame_evict()` function is called to select a frame to evict. The `lru` pointer starts at the beginning of the frame table and moves to the next frame each time it is called. If the current frame is pinned or has been recently accessed, it skips that frame and continues to the next one. The function continues this process until it finds a frame that is not pinned and has not been recently accessed.*
2. ***Evict the Selected Frame**: Depending on the page type of the associated page, the page content would be written to disk space or to the file system. It then clears the page table entry for the evicted page using `pagedir_clear_page()`. Finally, it frees the kernel page using `palloc_free_page()` and removes the frame from the frame table.*

> **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 eviction process clears the page table entry for the evicted page in process Q's page directory using the `pagedir_clear_page()` function. Then removing the frame from the frame table. The kernel page associated with the evicted frame is freed using `palloc_free_page()`. The SPT entry for the evicted page in process Q is updated to reflect that the page is no longer loaded into memory, i.e. the `is_loaded` flag in the SPT entry is set to `false`. If the page is swapped out, the swap index is stored in the SPT entry for future reference when the page needs to be loaded back into memory.

> **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.

The heuristic checks if the faulting address is within a certain range below `esp`. Specifically, it checks if the faulting address is within 32 bytes of `esp`. If the faulting address is outside this range, it is considered an invalid memory access, and the process is terminated.

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.)

The frame table and swap table are global, whereas the supplemental page table is specific to each process. Our design prevents the three conditions ---- hold and wait, no preemption and circular wait.

- **Lock Ordering**: To prevent deadlocks, the locks are acquired in a consistent order.
- **Minimizing Lock Hold Time**: A lock is released immediately after the associated function is complete.

> **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?

The `frame_lock` is held during the eviction process, ensuring that no other process can allocate or free a frame while the eviction is in progress. If process Q attempts to access the page after it has been evicted, a *page fault will occur. The page fault handler will check the SPT to determine if the page is still valid and should be loaded back into memory.*

> **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?

*The `frame_lock` ensures that only one process can allocate or modify a frame at a time. The **`swap_lock`** ensures that only one process can perform swap operations at a time. When a frame is being used to load a page from the file system or swap, its `pinned` flag is set as `true`, which means that the frame is temporarily fixed in memory and cannot be evicted. This ensures that even if the frame eviction logic is triggered by another process, it will skip over any frames that are currently `pinned`.*

> **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?

When a process makes a system call and accesses a paged-out page, a page fault occurs. The page fault handler checks if the accessed page is valid but not currently in memory. If the page is valid but paged out, the system schedules a disk operation to bring the page back into memory. During the page fault handling process, the frame allocated for the paged-in data is temporarily pinned to prevent it from being evicted while the system call is still in progress.

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.

Insteas of a single lock, our design uses a combination of a few key locks to protect critical sections of the VM system, rather than a single lock or many fine-grained locks. Specifically, it uses:

- *`frame_lock`: Protects access to the frame table.*
- *`swap_lock`: Protects access to the swap table.*
- *`process_lock`: Protects file operations.*

The use of separate locks for the frame table, swap table, and file operations allows multiple threads to perform different types of operations concurrently.

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 `threads/thread.h`:

```
struct thread {
    struct hash* page_table;            /* page table. */
    int mapid;                          /* next mapid to assign to the new
file. */
};
```

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.

`struct list mapped_filelist` is used to track which files are mapped to which pages. The system supports `mmap` and `munmap` system calls to manage memory-mapped files. `sys_mmap` maps a file into the process's address space, while `sys_munmap` unmaps it.

*For **page fault process**:*

- ***swap pages**: When a process accesses a page that is not currently in memory, a page fault occurs. The page fault handler consults the SPT to determine if the page is valid but not present. If the page is backed by swap space, the handler allocates a frame, reads the page from swap space, and updates the page table.*
- ***memory mapped files**: The page is read from the file on disk.*

*For **eviction processes**:*

- ***swap pages**: If the page is dirty, it is written back to swap space. If the page is clean, it can be evicted without writing it back.*
- ***memory mapped files**: If the page is dirty, its contents are written back to the original file on disk instead of to swap space. If the page is not dirty, it can be simply swapped out to disk without writing back to the file.*

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

Before creating a new file mapping, the system iterates through the address range of the new file mapping and checks whether there is an entry in the SPT for the given address or the page is already mapped in the process's page directory.

If either `get_page` or `pagedir_get_page` returns a non-`NULL` value for any page in the range, it indicates that the new mapping overlaps with an existing mapping. In this case, the function returns `-1` to indicate 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 why your implementation either does or does not share much of the code for the two situations.

Our design shares some part of the code for handling page faults and frame management between `mmap` mappings and demand-paged data from executables. This shared code includes the page fault handler, frame allocation, and eviction mechanisms. However, distinct handling is required for differences in write-back behavior and page table updates. Specifically, `mmap` mappings need to write back modified pages to the original file, while demand-paged data writes to swap space. This separation ensures that each type of mapping behaves correctly according to its semantics.

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?

No.