INF333 2024-2025 Spring Semester

Pintocchio
Sude Melis Pilaz <22401992@ogr.gsu.edu.tr>
Ali Burak Saraç <21401932@ogr.gsu.edu.tr>

Homework III
Design Document

Please provide answers inline in a `quote` environment.

1 Preliminaries

Q1: If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

> Answer here

Q2: Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, and lecture notes.

> Answer here

2 Page Table Management

2.1 Data Structures

Q3: 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.

From page.h:

```
enum vm_type { VM_BIN, VM_FILE, VM_ANON, VM_MMAP };
```

Defines the type of virtual memory page (binary, file-backed, anonymous, or memory-mapped).

```
struct vm_entry {
void *vaddr;
bool writable;
enum vm_type type;
struct file *file;
off_t offset;
size_t read_bytes;
size_t zero_bytes;
bool loaded;
int mapid;
int swap_slot;
struct hash_elem helem;
};
```

Represents a virtual page in the supplemental page table, storing metadata for loading, swapping, and file mapping.

```
struct supplemental_page_table {
struct hash pages;
struct lock page_lock;
};
```

Per-process hash table for tracking all virtual pages and synchronizing access.

```
struct mmap_desc {
int mapid;
struct file *file;
void *base_addr;
```

```
size_t page_cnt;
struct list_elem elem;
};
```

Describes a memory-mapped file region for a process.

From frame.h:

```
extern struct bitmap *frame_bitmap;
extern void **frame_kpages;
extern unsigned frame_count;
```

Tracks physical frame allocation status, kernel page pointers, and total frame count.

From thread.h:

```
struct supplemental_page_table spt;
struct list mmap_list;
int next_mapid;
```

Added to each thread to provide a per-process supplemental page table, a list of memory-mapped files, and a counter for the next available mapping ID.

2.2 Algorithms

Q4: In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

To access the SPT, we use a hash table stored in each thread under `thread_current()->spt`. When a page fault happens at address `fault_addr`, we first get the page start with:

```
void *page_addr = pg_round_down(fault_addr);
```

Next, we make a temporary entry and look it up:

```
struct vm_entry key;
key.vaddr = page_addr;
struct hash_elem *e =
    hash_find(&thread_current()->spt, &key.helem);
```

If `e` is not `NULL`, we recover the real entry:

```
struct vm_entry *vme =
    hash_entry(e, struct vm_entry, helem);
```

Then we can read the fields in `vme`:

- `vme->type` shows the page type (VM_BIN, VM_ANON, VM_FILE, VM_MMAP).
- `vme->file`, `vme->offset`, `vme->read_bytes`, `vme->zero_bytes`, `vme->writable`, and `vme->swap_slot` give more info.

Finally, we load or swap in the page and install it with:

```
pagedir_set_page(thread_current()->pagedir,
                 page_addr, kpage, vme->writable);
```

This method lets us find and load any page quickly using the SPT.

Q5: 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?

In our code, we avoid any conflict between kernel and user accessed/dirty bits by only using the user page table entry. We follow these steps:

2.1. Perform all file I/O (loading or swapping) on the frame via its kernel virtual address (`kpage`) *before* installing the user mapping.

2.2. Install the user mapping with

```
pagedir_set_page(cur->pagedir, vaddr, kpage, writable);
```

At this point the user PTE's accessed and dirty bits start cleared.

2.3. For eviction or `munmap`, check and clear bits only on the user mapping:

```
pagedir_is_accessed(cur->pagedir, vaddr);
pagedir_is_dirty(cur->pagedir, vaddr);
pagedir_clear_page(cur->pagedir, vaddr);
```

2.4. Never read or write the direct-mapping (kernel) PTE's bits.

By tracking exactly one set of bits (the user PTE) and ignoring any kernel alias, we completely avoid the aliasing issue.

2.3 Synchronization

Q6: When two user processes both need a new frame at the same time, how are races avoided?

To avoid races when two processes need a new frame at the same time, we use a global lock in our frame allocation code:

2.1. We declare `static struct lock frame_lock;` in `frame.c` and initialize it in `frame_init()` with

```
lock_init(&frame_lock);
```

2.2. In `frame_alloc()`, we acquire the lock before touching the frame table or bitmap:

```
lock_acquire(&frame_lock);
```

2.3. We then scan for a free frame or evict one if needed, update the frame table, and map the page.

2.4. Finally, we release the lock with

```
lock_release(&frame_lock);
```

This simple mutual-exclusion ensures that only one thread at a time can allocate or evict frames, so no two processes ever pick the same frame simultaneously.

2.4 Rationale

Q7: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

We chose a hash table for our supplemental page table (SPT) because it gives us average-case O(1) lookups, which is essential when handling frequent page faults. A page fault must quickly locate the metadata for the faulting virtual address, and the hash table lets us do that with minimal overhead.

Insertion and removal in the SPT are also O(1) on average, so when we load a new page or unmap a file, we can update the table efficiently. We considered other structures:

- A balanced tree would give O(log n) lookups and updates, which is slower in the common case.

- A linked list would be O(n) for lookups, which would degrade performance as the number of pages grows.

By contrast, the hash table in Pintos' `lib/kernel/hash.h` is simple to use and scales well, making it the best fit for our virtual-to-physical mapping needs.

3 Paging to and from Disk

Q8: 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.

swap.h

```
void swap_init(void);
int  swap_out(void *kpage);
bool swap_in(int slot, void *kpage);
void swap_free(int slot);
```

We declare swap functions to initialize swap, write a frame out, read it back, and free the slot.

vm/page.h

```
struct vm_entry {
    ...
    int swap_slot;
    ...
};
```

We add `swap_slot` to record which disk slot holds an evicted page.

swap.c

```
static struct bitmap *swap_bitmap;
static struct block  *swap_device;
```

We use `swap_bitmap` to track free/used slots and `swap_device` to read/write swap pages.

3.1 Algorithms

Q9: When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

Q9

When we need a frame but none is free, we use the clock algorithm in `frame_get_victim()`:

3.1. We keep a global pointer `clock_hand` into the frame table.

3.2. In a loop, we pick the frame at `clock_hand` and get its user virtual address from the reverse map.

3.3. If `pagedir_is_accessed(cur->pagedir, vaddr)` is true, we clear the accessed bit with `pagedir_clear_page(cur->pagedir, vaddr)` and advance `clock_hand`.

3.4. Otherwise, we select this frame to evict.

3.5. If the page is dirty (`pagedir_is_dirty`), we swap it out or write it back to its file. We then update the SPT entry to mark it unloaded.

This method makes one pass through frames and approximates LRU by giving recently used pages a second chance.

Q10: 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?

Answer here

Q11: 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.

Answer here

3.2 Synchronization

Q12: 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.)

Answer here

Q13: 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?

Answer here

Q14: 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?

Answer here

Q15: 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?

Answer here

3.3 Rationale

Q16: 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.

Answer here

4 Memory Mapped Files

4.1 Data Structures

Q17: 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.

Answer here

4.2 Algorithms

Q18: 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.

Answer here

Q19: Explain how you determine whether a new file mapping overlaps any existing segment.

Answer here

4.3 Rationale

Q20: 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.

Answer here

5 Survey Questions

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want–these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

Q1: In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

Answer here

Q2: Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Answer here

Q3: Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Answer here

Q4: Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

Answer here

Q5: Any other comments?

Answer here