Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Jianan Ji 2000012995@stu.pku.edu.cn

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

> 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.

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.

```
struct sup_page_entry{
    void *upage; /* User virtual address */
    void *kpage; /* Kernel virtual address */
    struct frame_entry *frame_entry; /* Frame entry */
    struct file *file; /* File */
    off_t ofs; /* Offset in file */
    uint32_t read_bytes; /* Number of bytes to read from file */
    uint32_t zero_bytes; /* Number of bytes to set to zero */
    bool writable; /* Page is writable */
    bool loaded; /* Page is loaded to memory */
    bool pinned; /* Page is pinned */
    struct thread *thread; /* Thread that owns the page */
    block_sector_t sector; /* Sector number in swap */
    struct hash_elem hash_elem; /* Hash element */
};
```

ALGORITHMS

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

The `sup_page_lookup` function takes a user virtual address `upage` as input and searches for the corresponding page entry in the Supplemental Page Table. It uses the `hash_find` function to perform the lookup in the SPT hash table of the current thread.

If the page entry is found, the function returns a pointer to the `struct sup_page_entry` associated with the page. If the page entry is not found, the function returns `NULL`.

The function includes optional debug print statements to provide additional information about the success or failure of the lookup operation.

> 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?

I exclusively access memory through the Page Table Entry or the user virtual memory address present in the frame structure. I avoid accessing memory directly from the kernel virtual memory within the user pool.

SYNCHRONIZATION

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

When two user processes simultaneously require a new frame, races are avoided in this code by using locks.

Firstly, the code acquires a scanning lock (`lock_for_scan`) to ensure that only one thread is performing the scanning operation for selecting a frame. This ensures that multiple threads do not simultaneously choose the same available frame.

Next, it iterates through the frame array (`frames`) to find an available frame. For each frame, it attempts to acquire the frame lock (`frame_lock`). If it successfully acquires the lock and the `page` field of the frame is NULL, indicating the frame is available, it releases the scanning lock and returns the frame. This ensures that only one thread can successfully acquire and occupy an available frame, while other threads continue to attempt to acquire other frames.

RATIONALE

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

Hash tables offer constant-time lookup, handle collisions, and have lower memory overhead compared to other structures. They can efficiently handle dynamic updates and scale well with large address spaces.

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.

```
static struct block *swap_device; //swap device, used to store pages

static struct bitmap *swap_bitmap;  //swap bitmap, used to manage swap
space

static struct lock swap_lock; //swap lock, used to protect swap device
static struct lock bitmap_lock; //bitmap lock, used to protect swap bitmap
```

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.

The code implements the Clock algorithm for choosing a frame to evict when no free frames are available. Here's a concise explanation of the code:

1. The code initializes a variable called "hand" to keep track of the current position in the frame list.
2. It enters a loop that iterates through the frames twice the number of frame entries (frame_cnt * 2).
3. In each iteration, the "hand" variable is incremented and wrapped around to ensure it stays within the valid range of frame indices.
4. The code attempts to acquire the lock for the frame at the current "hand" position.
5. If the lock acquisition is successful, it checks if the frame is not associated with any page (indicating it is free).
    - If the frame is free, it is chosen as the eviction candidate, and the loop is exited.
6. If the frame is associated with a page, it checks if the page has been accessed by the process.
    - If the page has been accessed, the accessed bit is cleared to indicate that it has been recently used, and the loop continues to the next iteration.
7. If the page has not been accessed, it considers the frame for eviction.
    - It attempts to swap out the page using the "page_out" function.
    - If the page-out operation is successful, the frame is marked as free and chosen as the eviction candidate, and the loop is exited.
8. The lock for the frame is released before proceeding to the next iteration.

> 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?

1. The page table entry corresponding to the frame's virtual address (upage) is cleared using pagedir_clear_page.

2. The dirty status of the page is determined using pagedir_is_dirty.

3. If necessary, the page is swapped out to secondary storage using swap_out.

4. Update data structures: If the swap out operation is successful, the frame entry's page field is set to NULL, and the frame_entry field in the sup_page_entry structure is also set to NULL.

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.)

1. The code follows a strict lock hierarchy where locks are acquired in a specific order to prevent circular dependencies. This order ensures that a thread can never acquire a lock that is already held by a lower-level lock, preventing the formation of a deadlock cycle.

2. To avoid potential deadlocks, the code ensures that locks are always acquired in the same predetermined order. This consistent lock ordering prevents situations where different threads acquire locks in different orders, which can lead to a circular waiting pattern and deadlock.

> 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?

Individual locks are used for each frame to prevent concurrent access and modifications. This ensures that Process Q cannot access or modify the page while it is being evicted due to a page fault in Process P. The lock held by Process P throughout the eviction process avoids a race condition with Process Q faulting in the page. This design enables parallelism and maintains data consistency during evictions.

> 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?

A lock is used to synchronize access to the frame. The lock is acquired before the read operation begins and is released once the page has been successfully read into the frame. This prevents any concurrent attempts by process Q to evict the frame while it is still being read in, ensuring data consistency and avoiding potential conflicts.

> 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?

I use page faults to bring in pages. When a page fault occurs for a paged-out page, the page fault handler is triggered, which then calls the load_page function to bring the page back into physical memory. The load_page function, in turn, invokes the swap_in mechanism to retrieve the page from the swap space.

If an attempted access is made to an invalid virtual address, it will trigger a page fault, which is gracefully handled by the page fault handler. The page fault handler identifies the invalid address and return error code.

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.

I have chosen to use a combination of locks for different components of the VM system: page, frame, and swap. Additionally, each frame has its own individual lock, and there is a lock specifically for the swap bitmap. In total, there are five different locks used in the design.

By using multiple locks, I aim to strike a balance between synchronization simplicity and parallelism. Having separate locks for different components allows for concurrent access and modification to those components, which can increase parallelism and improve performance in a multi-threaded environment.

However, using multiple locks does introduce complexity in synchronization and raises the possibility of deadlocks. To mitigate these issues, I have carefully designed the lock hierarchy and ensure that locks are acquired and released in a consistent and predefined order to prevent circular dependencies and minimize the risk of deadlocks.