PROJECT 3: VIRTUAL MEMORY DESIGN DOCUMENT

Zhang Yichi zhangych6@shanghaitech.edu.cn
Hu Aibo huab@shanghaitech.edu.cn

PRELIMINARIES

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

```
// Supplemental page
struct page
{
  void *uaddr; // user virtual address for this page
  struct frame *frame; // corresponding frame. if null, page is not in memory
  bool rw;       // is writable
  bool is_stack; // used to check stack growth

  struct hash_elem page_table_elem; // for page_table
};

struct frame
{
```

```
  struct hash_elem frame_table_elem; // for frame_table
  void *kpage; // kernel virtual address for this frame
  struct page *upage; // corresponding page.
  struct thread *owner; // thread that owns this frame

  int second_change; // for Second Change algorithm
};

static struct hash frame_table; // The frame table
struct lock frame_global_lock; // Ensure synchronization

struct thread
{
    // ... Omit other members
    struct hash page_table; // Supplemental page table
    void *esp; // User stack pointer. Saved when interrupt occurs.
};
```

ALGORITHMS

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

Our SPT is a hash table.

First, we find the page in the SPT.

If the page is in memory, i.e. `page->frame != NULL`, we can directly access the data in the page.

If the page is in swap, we load the page from swap disk to memory, and then access the data in the page.

If the page is a mmap and not in memory. We load the page from file to memory, and then access the data in the 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?

In our implementation, we do not allow aliasing. We avoid accessing the memory using kernel virtual address. Kernel virtual address is only used to initialize the page.

SYNCHRONIZATION

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

We use a global lock `frame_global_lock` to ensure synchronization.

Before a thread operates on the frame table, it must acquire the lock. After the operation, it must release the lock.

RATIONALE

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

We use a hash table. So that given a virtual address, we can find the corresponding physical address in $O(1)$ time.

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.

```
struct page
{
    // ... Omit other members
  size_t swap_index; // where is this page in swap disk. when this page is not
                     // in swap, swap_index=BITMAP_ERROR
};
```

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 use Second Change Algorithm to choose a frame.

All pages are considered in a round robin matter, but one page be relaced only when visit it at second time. When accessing a page, the flag will be reset.

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?

When a frame is not belong to a process Q, we use `frame_free` to clean this frame, and set `page->frame = NULL`

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 address is not lower than PYHS_BASE and not higher than `esp-32`, and the stack is not larger than 8MB, we think it is a valid stack address, and extend the stack. Otherwise it not cause a stack growth.

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.)

"Wait while holding" is a necessary condition for deadlock. We use only one lock, so we do not meet this condition. Therefore, we do not have 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?

We use a global lock `frame_global_lock` to ensure synchronization.

Before a thread operates on the frame table, it must acquire the lock. After the operation, it must release the lock.

So P and Q won't have a race.

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?

We use a global lock `frame_global_lock` to ensure synchronization.

Before a thread operates on the frame table, it must acquire the lock. After the operation, it must release the lock.

So Q will wait until P finishes reading file.

> 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 use the page fault to bring in pages. When a page fault occurs, in the interrupt handler, we first check if the page is in the SPT. If it is, we load the page from file or swap disk to memory. Then we return to the interrupted instruction. Otherwise, we think user gives syscall with invalid virtual address.

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 a single lock. Because Pintos is a single-core system.

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.

```
struct thread
{
    // ... Omit other members
    struct list mmap_list; // List of mmaped files
    mapid_t next_mapid;
};

struct page
{
  // ... Omit other members
```

```
  struct file *file; // Indicate whether the page is a mmap page
  off_t file_offset;
  size_t file_size; // [file_offset, file_offset + file_size) is mapped to this
                    // page
};
```

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.

In lazy creation, we check `file` attribute to determine whether the page is a mmap page. If yes, we use `fread` to load the content of the file and use `memset` to set the rest of the page to zero.

In swap, we check `file` attribute to determine whether the page is a mmap page. If yes, instead of write it to the swap disk. We write it directly to the file it maps. So the `swap_index` of a mmap page is always `BITMAP_ERROR`.

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

We use a for loop to check every address. If it overlaps a existing page in spte, mmap syscall is terminated.

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.

We implement stack growth and swap first. Adding mmap doesn't modify the code much. Because in mmap syscall, we just create many mmap pages in spte without reading the content of the file. Mmap pages share the same page fault handler with the stack growth. The only thing we need to do is to add some if else. Swap is similar.