Project 3: Virtual Memory

author: 冯宇凡 fengyf2@shanghaitech.edu.cn

杨 斌 yangbin@shanghaitech.edu.cn

date: 2021/12/06

Consulted Sources:

- Paste bin

  https://pastebin.com/zw1SSumP

- CSCI 350: Pintos Guide

  https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

- Other guy's implementation

  https://github.com/guliashvili/CS140-Operating-Systems-pintos/tree/90d8251ff6976a7481c34cc9566181abe5642d1d

  https://github.com/codyjack/OS-pintos

PAGE TABLE MANAGEMENT

DATA STRUCTURES

> 1. 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`:

```
/* Frame table that stores all frames globally.  */
struct list frame_table;

/* Lock of frame table. */
struct lock frame_table_lock;

/* Frame table entry in frame_table. */
struct frame_table_entry {
    uint32_t* frame;                            /* Kernel virtual address of this frame. */
    struct sup_page_table_entry* vpage;         /* The corresponding page entry. */

    struct list_elem elem;                      /* List elem in frame_table. */
};
```

In `vm/page.h`:

```
/* Page entry status number. */
enum sup_page_table_entry_status{
    FRAME,
    SWAP,
    EMPTY
};

/* Supplemental page table entry. */
struct sup_page_table_entry {
    struct thread* owner;                       /* Owner thread of this page. */
```

```
    void* vadd;                                 /* Virtual address of this page. */
    enum palloc_flags flag;                     /* Palloc flag when we want to create a frame
                                                   that associate with this page. */

    union entry_data
    {
        struct frame_table_entry * frame;       /* The associated frame entry under FRAME
status. */
        size_t swap_index;                      /* The corresponding index on swap table under
                                                   SWAP status. */
    }value;                                     /* Data according to different status. */

    enum sup_page_table_entry_status status;    /* Status of this page. */

    bool writable;                              /* Writable bit. */
    struct lock page_lock;                      /* Lock of this page. */

    int fd;                                     /* File descriptor if this page associates with
a file. */
    int file_start;                             /* The start position of file that stored in
this page. */
    int file_end;                               /* The end position of file. */
    struct list_elem elem;                      /* List elem for `sup_page_table` */
};

/* Supplemental page table. */
struct sup_page_table
{
    struct list table;              /* List that stores all entries. */
    struct lock table_lock;         /* Lock of this table */
};
```

New struct member in `threads/thread.h`

```
struct thread{
    struct sup_page_table *sup_page_table;  /* Supplemental page table of this thread. */
}
```

ALGORITHMS

> 2. In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

This is implemented in `vm/page.c:sup_page_table_look_up (struct sup_page_table *table, void *vaddr)`. The input of this function is a supplemental page table and an user virtual address. In this function, we iterate through the table and find if there is an entry whose `vadd == vaddr`. Then this is the entry we a looking for. If we didn't find the page, return a NULL pointer.

> 3. 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, a frame will only be associated with one unique single user virtual address.

We will get a frame only when we activate a virtual page, then this frame will be linked with this unique virtual page. In the next when we doing swap, we will use `pagedir_clear_page` to remove the virtual page in directory but keep our supplemental page table entry. In this way, we ensure that only one virtual page is linked with this frame at the same time.

SYNCHRONIZATION

> 4. When two user processes both need a new frame at the same time, how are races avoided?

When a process need a new frame, it will call `sup_page_activate()` in `vm/page.c`. In this function, we will call `frame_get_page` in `vm/frame.c`. In this function, we get a frame directly by `palloc_get_page`. This funtion was provided and it uses a lock while trying allocate a frame from a pool. After that, we initialize a frame structure in our frame table with a lock of table. In this way, we ensure that both "allocate a frame from pool" and "make changes to our global frame table" are protected by locks.

RATIONALE

> 5. Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

Our design is that we have a global frame table based on list to track all allocated frame, and a supplemental page table based on list for every thread. This is because frame should be allocated globally, but a user process should have its own stack. We implement tables based on list, since we are familiar with list and it's easier to implement.

A frame entry must have a pointer to a page, and if a page's status is "FRAME", then it will also have a pointer to a frame entry. A frame entry has its own kernel virtual address, and a page entry also has its own user virtual address. We choose this design so that all information with an address will be encapsure into entries and matching every entries is easy and clear.

PAGING TO AND FROM DISK

DATA STRUCTURES

> 1. 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`:

```
/* Sector number to store a single page. */
#define PAGE_SECTOR_NUM (PGSIZE/BLOCK_SECTOR_SIZE)

/* Bitmap to track the usage of swap disk. */
struct bitmap* swap_map;
/* Lock of swap_map. */
struct lock swap_map_lock;
```

New struct variable in `vm/frame.h:struct frame_table_entry`:

```
/* Frame table entry in frame_table. */
struct frame_table_entry {
    bool swap_able;                             /* Able to swap. */
    bool last_used;                             /* Last used bit in second chance algorithm. */
};
```

New global variable in `vm/frame.h`:

```
/* Pointer of next frame to evict. */
struct list_elem* frame_table_evict_ptr;
```

In `vm/page.h:struct sup_page_table_entry`:

```
struct sup_page_table_entry {
    union entry_data
    {
        struct frame_table_entry * frame;       /* The associated frame entry under FRAME
status. */
        size_t swap_index;                      /* The corresponding index on swap table under
                                                   SWAP status. */
    }value;                                     /* Data according to different status. */
};
```

ALGORITHMS

> 2. 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 chance algorithm to choose a frame. We newly define a global pointer `frame_table_evict_ptr`.

Every time we want to choose a frame to evict, we continue to move the pointer along the frame table. When we encounter a frame entry with `swap_able == true`, we set `last_used` to 0 if `last_used == 1` and choose the entry to evict if `last_used == 0`.

> 3. 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 we evict a frame, we call `pagedir_clear_page` to remove the virtual page in its `pagedir`, so that the `kaddr` is not match to this `vaddr` anymore. Then, we set the status of the corresponding page table entry from "FRAME" to "SWAP". The `value` is set to the swap index. So that this page has nothing to do with previous frame.

Also, we will free the frame page along with frame entry when evict the frame. So that we can newly create a frame. This frame is connected with new page.

> 4. 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 invalid virtual address should satisfy that:

- It is not NULL and it is an user virtual address.
- We can not find this address in our supplemental page table.
- `fault address > f->esp - 33` and `fault address < f->esp + PGSIZE*100`. That it should not be lower than esp for 32 bytes and it should not be too large.

Then we can grow the stack.

SYNCHRONIZATION

> 5. 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.)

There is a lock for frame table, a lock for swap table, a lock for each page table and a lock for each page. All of the locks are only used in their section. And they does not interact with other parts of lock acquiring. Therefore, we won't have the situation like holding and waiting. So, deadlock is prevented.

> 6. 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 have set a lock for every page. When we are doing eviction, we have to acquire the lock of the evicted page, so that another thread cannot access or modify this page during eviction process.

> 7. 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?

This is also realize by acquiring lock of the certain page when we want to read, write or modify this page.

> 8. 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 handled this in our page fault handler.

Any access to invalid virtual address will enter into page fault handler. In the handler, we check our own supplemental page table to see that if we have allocated this fault address before. If we can find the page in our page table, then we can brink back the page from swap or file. In this case we only have to activate the page again and return. If we cannot find the page, then this is really a page fault and should exit(-1).

RATIONALE

> 9. 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.

As mentioned in q5 before, there is a lock for frame table, a lock for swap table, a lock for each page table and a lock for each page. We assure that a lock is only responsible for a certain resource but not for some global events. So that a certain level of parallelism is permitted. All of the locks are only used in their section. And they does not interact with other parts of lock acquiring. Therefore, we won't have the situation like holding and waiting. So, deadlock is prevented.

MEMORY MAPPED FILES

DATA STRUCTURES

> 1. 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.

New struct variable in `vm/page.h:struct sup_page_table_entry`:

```
/* Supplemental page table entry. */
struct sup_page_table_entry {
    int fd;                                 /* File descriptor if this page associates with
a file. */
    int file_start;                         /* The start position of file that stored in
this page. */
    int file_end;                           /* The end position of file. */
    struct list_elem elem;                  /* List elem for `sup_page_table` */
};
```

In `userprog/mmap.h`:

```
/* Necessary info for a mmap */
struct mmap_file
{
    void* vadd;                     /* The starting address of this mapping */
    int id;                         /* The id of this mapping which is unique */
    int page_count;                 /* The numbers of pages of this file hold */
    int fd;                         /* The fd of the mapping file */
    struct list_elem elem;          /* List elem */
};
```

ALGORITHMS

> 2. 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 mmap was called, we will create some page table entries corresponding to the demanding virtual address and add them into page table. So that we "tag" this addresses of files. When we want to use the file, we will activate the page, which means load page content to memory. So, we will allocate a frame for this page and load the file, this is similiar with previous operations on pages.

The only difference is that when we are evicting frames, we will write the frame back to file if the page is linked with file. And when we are activating pages, we will read from file to newly created frame.

> 3. Explain how you determine whether a new file mapping overlaps any existing segment.

When we are mapping the file on the memory, we check that if the virtual address is already been used, to be specific, that the virtual address is already allocated an entry in our page table. If the virtual address has not been used before, we should not get any pages after look_up and then we can create new virtual pages for the file.

RATIONALE

> 4. 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.

The only difference is that when we are evicting frames, we will write the frame back to file if the page is linked with file. Otherwise, we will do swap and save it on swap disk. We only differenciate this because other operations, e.g. finding a frame to evict, second chance algorithm and freeing the frame are quite same.

Contribution

- Figure out data structures and function interfaces together.
- For frame table, page table and stack growth, we implemented different functions separately and equally.
- Swap was implemented by Feng Yufan, and mmap was implemented by Yang Bin.
- Debugging together.