Project3: Memory

PintOS

陈震雄

2025 年 3 月 21 日

武汉大学

Table of contents

Introduction

Introduction

By now you should have some familiarity with the inner workings of Pintos.

Your OS can properly handle multiple threads of execution with proper synchronization, and can load multiple user programs at once.

However, **the number and size of programs that can run is limited by the machine's main memory size. In this assignment, you will remove that limitation.**

Preparation

What should we complete in preparation?

Frame table, supt, swap table

init, allocate, free

Frame data structure

```
1  // frame.c
2  /* A global lock, to ensure critical sections on frame operations. */
3  static struct lock frame_lock;
4
5  /* A mapping from physical address to frame table entry. */
6  static struct hash frame_map;
7
8  /* A (circular) list of frames for the clock eviction algorithm. */
9  static struct list frame_list;      /**< the list */
10 #ifdef LRU
11 /* Global counter for LRU */
12 static size_t lru_counter;
13 #else
14 static struct list_elem *clock_ptr; /**< the pointer in clock algorithm */
15 #endif
16 /* Frame Table Entry */
17 struct frame_table_entry
18   {
19     void *kpage;               /**< Kernel page, mapped to physical address */
20
21     struct hash_elem helem;    /**< see ::frame_map */
22     struct list_elem lelem;    /**< see ::frame_list */
23
24     void *upage;               /**< User (Virtual Memory) Address, pointer to page */
25     struct thread *t;          /**< The associated thread. */
26
27     bool pinned;               /**< Used to prevent a frame from being evicted,
28                                     while it is acquiring some resources.
29                                   If it is true, it is never evicted. */
30 #ifdef LRU
31     size_t last_used;          /**< LRU timestamp */
32 #endif
33   };
```

Overview

```c
#ifndef VM_FRAME_H
#define VM_FRAME_H

#include <hash.h>
#include "lib/kernel/hash.h"

#include "threads/synch.h"
#include "threads/palloc.h"


/* Functions for Frame manipulation. */
void vm_frame_init (void);
void* vm_frame_allocate (enum palloc_flags flags, void *upage);

void vm_frame_free (void*);
void vm_frame_remove_entry (void*);

void vm_frame_pin (void* kpage);
void vm_frame_unpin (void* kpage);

#endif /**< vm/frame.h */
```

Allocate frame

```
void*
vm_frame_allocate (enum palloc_flags flags, void *upage)
{
  lock_acquire (&frame_lock);

  void *frame_page = palloc_get_page (PAL_USER | flags);
  if (frame_page == NULL)
  {
    /* page allocation failed. */
    /* first, swap out the page */
    struct frame_table_entry *f_evicted =
    pick_frame_to_evict( thread_current()
    ->pagedir );

    ASSERT (f_evicted != NULL && f_evicted
    ->t != NULL);

    /* clear the page mapping, and replace it
    with swap */
    ASSERT (f_evicted->t->pagedir !=
    (void*) 0xcccccccc);
    pagedir_clear_page(f_evicted->t
    ->pagedir, f_evicted->upage);

    bool is_dirty =
pagedir_is_dirty(f_evicted->t->pagedir,
    f_evicted->upage)
    || pagedir_is_dirty(f_evicted->t
    ->pagedir, f_evicted->kpage);
    swap_index_t swap_idx = vm_swap_out(
    f_evicted->kpage );
    vm_supt_set_swap(f_evicted->t->supt,
    f_evicted->upage, swap_idx);
    vm_supt_set_dirty(f_evicted->t->supt,
    f_evicted->upage, is_dirty);
    vm_frame_do_free(f_evicted->kpage, true); /**<
    f_evicted is also invalidated. */

    frame_page = palloc_get_page (PAL_USER |
    flags);
    ASSERT (frame_page != NULL); /**< should
    success in this chance. */
  }
  struct frame_table_entry *frame =
  malloc(sizeof(struct frame_table_entry));
    if(frame == NULL)
    {
      /* frame allocation failed. a critical
      state or panic? */
      lock_release (&frame_lock);
      return NULL;
    }
    frame->t = thread_current ();
    frame->upage = upage;
    frame->kpage = frame_page;
    frame->pinned = true;
  /**< can't be evicted yet */

    /* insert into hash table */
    hash_insert (&frame_map, &frame->helem);
    list_push_back (&frame_list, &frame->lelem);

    lock_release (&frame_lock);
    return frame_page;
  }
```

Free frame

```
1   void
2   vm_frame_do_free (void *kpage, bool free_page)
3   {
4     ASSERT (lock_held_by_current_thread(&frame_lock) == true);
5     ASSERT (is_kernel_vaddr(kpage));
6     ASSERT (pg_ofs (kpage) == 0); /**< should be aligned. */
7
8     /* hash lookup : a temporary entry */
9     struct frame_table_entry f_tmp;
10    f_tmp.kpage = kpage;
11
12    struct hash_elem *h = hash_find (&frame_map, &(f_tmp.helem));
13    if (h == NULL)
14    {
15      PANIC ("The page to be freed is not stored in the table");
16    }
17
18    struct frame_table_entry *f;
19    f = hash_entry (h, struct frame_table_entry, helem);
20
21    hash_delete (&frame_map, &f->helem);
22    list_remove (&f->lelem);
23
24    /* Free resources. */
25    if(free_page) palloc_free_page(kpage);
26    free(f);
27  }
```

Select one frame to evict-LRU?

```
1  static struct frame_table_entry*
2  pick_frame_to_evict(uint32_t* pagedir)
3  {
4    struct list_elem *e;
5    struct frame_table_entry *victim = NULL;
6    size_t min_last_used = (size_t)-1;
7
8    /* Iterate over all frames to find LRU */
9    for (e = list_begin(&frame_list); e != list_end(&frame_list); e = list_next(e))
10   {
11       struct frame_table_entry *f = list_entry(e, struct frame_table_entry, lelem);
12       if (f->pinned)
13           continue;
14
15       /* Check and update access bit */
16       bool accessed = pagedir_is_accessed(f->t->pagedir, f->upage) ||
17                       pagedir_is_accessed(f->t->pagedir, f->kpage);
18       if (accessed)
19       {
20           f->last_used = lru_counter++;
21           pagedir_set_accessed(f->t->pagedir, f->upage, false);
22           pagedir_set_accessed(f->t->pagedir, f->kpage, false);
23       }
24
25       /* Track the least recently used */
26       if (f->last_used < min_last_used)
27       {
28           min_last_used = f->last_used;
29           victim = f;
30       }
31   }
32   ASSERT(victim != NULL);
33   return victim;
34 }
```

Select one frame to evict-Clock

```
1  struct frame_table_entry* pick_frame_to_evict( uint32_t *pagedir )
2  {
3    size_t n = hash_size(&frame_map);
4    if(n == 0) PANIC("Frame␣table␣is␣empty,␣can't␣happen␣-␣there␣is␣a␣leak␣somewhere");
5
6    size_t it;
7    for(it = 0; it <= n + n; ++ it) /**< prevent infinite loop. 2n iterations is enough. */
8    {
9      struct frame_table_entry *e = clock_frame_next();
10     /* if pinned, continue */
11     if(e->pinned) continue;
12     /* if referenced, give a second chance. */
13     else if( pagedir_is_accessed(pagedir, e->upage))
14     {
15       pagedir_set_accessed(pagedir, e->upage, false);
16       continue;
17     }
18
19     /* OK, here is the victim : unreferenced since its last chance. */
20     return e;
21   }
22
23   PANIC ("Can't␣evict␣any␣frame␣--␣Not␣enough␣memory!\n");
24 }
25 struct frame_table_entry* clock_frame_next(void) {
26   if (list_empty(&frame_list))
27     PANIC("Frame␣table␣is␣empty,␣can't␣happen␣-␣there␣is␣a␣leak␣somewhere");
28   if (clock_ptr == NULL || clock_ptr == list_end(&frame_list))
29     clock_ptr = list_begin (&frame_list);
30   else
31     clock_ptr = list_next (clock_ptr);
32   struct frame_table_entry *e = list_entry(clock_ptr, struct frame_table_entry, lelem);
33   return e;
34 }
```

Page data structure

```
1  /** Indicates a state of page. */
2  enum page_status
3  {
4    ALL_ZERO,         /**< All zeros */
5    ON_FRAME,         /**< Actively in memory */
6    ON_SWAP,          /**< Swapped (on swap slot) */
7    FROM_FILESYS      /**< from filesystem (or executable) */
8  };
9  /**
10   Supplemental page table. The scope is per-process.
11  */
12 struct supplemental_page_table
13   {
14     /* The hash table, page -> spte */
15     struct hash page_map;
16   };
17 struct supplemental_page_table_entry {
18     void *upage;              /**< Virtual address of the page (the key) */
19     void *kpage;              /**< Kernel page (frame) associated to it.
20                                  Only effective when status == ON_FRAME.
21                                  If the page is not on the frame, should be NULL. */
22     struct hash_elem elem;
23     enum page_status status;
24     bool dirty;               /**< Dirty bit. */
25     /* for ON_SWAP */
26     swap_index_t swap_index;  /**< Stores the swap index if the page is swapped out.
27                                  Only effective when status == ON_SWAP */
28     /* for FROM_FILESYS */
29     struct file *file;
30     off_t file_offset;
31     uint32_t read_bytes, zero_bytes;
32     bool writable;
33   };
```

Overview

```
/**
  Methods for manipulating supplemental page tables.
 */
struct supplemental_page_table* vm_supt_create (void);
void vm_supt_destroy (struct supplemental_page_table *);

bool vm_supt_install_frame (struct supplemental_page_table *supt, void *upage, void *kpage);
bool vm_supt_install_zeropage (struct supplemental_page_table *supt, void *);
bool vm_supt_set_swap (struct supplemental_page_table *supt, void *, swap_index_t);
bool vm_supt_install_filesys (struct supplemental_page_table *supt, void *page,
    struct file * file, off_t offset, uint32_t read_bytes, uint32_t zero_bytes, bool writable);

struct supplemental_page_table_entry* vm_supt_lookup (struct supplemental_page_table *supt, void *);
bool vm_supt_has_entry (struct supplemental_page_table *, void *page);

bool vm_supt_set_dirty (struct supplemental_page_table *supt, void *, bool);

static bool vm_load_page_from_filesys(struct supplemental_page_table_entry *, void *);
bool vm_load_page(struct supplemental_page_table *supt, uint32_t *pagedir, void *upage);

bool vm_supt_mm_unmap(struct supplemental_page_table *supt, uint32_t *pagedir,
    void *page, struct file *f, off_t offset, size_t bytes);

void vm_pin_page(struct supplemental_page_table *supt, void *page);
void vm_unpin_page(struct supplemental_page_table *supt, void *page);
```

vm_supt_install_frame

Four cases: **frame, zero page, filesys, swap**

```
1  bool
2  vm_supt_install_frame (struct supplemental_page_table *supt, void *upage, void *kpage)
3  {
4    struct supplemental_page_table_entry *spte;
5    spte = (struct supplemental_page_table_entry *) malloc(sizeof(struct supplemental_page_table_entry));
6
7    spte->upage = upage;
8    spte->kpage = kpage;
9    spte->status = ON_FRAME;
10   spte->dirty = false;
11   spte->swap_index = -1;
12
13   struct hash_elem *prev_elem;
14   prev_elem = hash_insert (&supt->page_map, &spte->elem);
15   if (prev_elem == NULL)
16     /* successfully inserted into the supplemental page table. */
17     return true;
18   else
19     {
20       /* failed. there is already an entry. */
21       free (spte);
22       return false;
23     }
24 }
```

vm_supt_install_zeropage

```
1   bool
2   vm_supt_install_zeropage (struct supplemental_page_table *supt, void *upage)
3   {
4     struct supplemental_page_table_entry *spte;
5     spte = (struct supplemental_page_table_entry *) malloc(sizeof(struct supplemental_page_table_entry));
6
7     spte->upage = upage;
8     spte->kpage = NULL;
9     spte->status = ALL_ZERO;
10    spte->dirty = false;
11
12    struct hash_elem *prev_elem;
13    prev_elem = hash_insert (&supt->page_map, &spte->elem);
14    if (prev_elem == NULL) return true;
15
16    /* there is already an entry -- impossible state */
17    PANIC("Duplicated␣SUPT␣entry␣for␣zeropage");
18    return false;
19  }
```

vm_supt_install_filesys

```
1  bool
2  vm_supt_install_filesys (struct supplemental_page_table *supt, void *upage,
3      struct file * file, off_t offset, uint32_t read_bytes, uint32_t zero_bytes, bool writable)
4  {
5    struct supplemental_page_table_entry *spte;
6    spte = (struct supplemental_page_table_entry *) malloc(sizeof(struct supplemental_page_table_entry));
7
8    spte->upage = upage;
9    spte->kpage = NULL;
10   spte->status = FROM_FILESYS;
11   spte->dirty = false;
12   spte->file = file;
13   spte->file_offset = offset;
14   spte->read_bytes = read_bytes;
15   spte->zero_bytes = zero_bytes;
16   spte->writable = writable;
17
18   struct hash_elem *prev_elem;
19   prev_elem = hash_insert (&supt->page_map, &spte->elem);
20   if (prev_elem == NULL) return true;
21
22   /* there is already an entry -- impossible state */
23   PANIC("Duplicated␣SUPT␣entry␣for␣filesys-page");
24   return false;
25 }
```

vm_supt_set_swap

```
1   bool
2   vm_supt_set_swap (struct supplemental_page_table *supt, void *page, swap_index_t swap_index)
3   {
4     struct supplemental_page_table_entry *spte;
5     spte = vm_supt_lookup(supt, page);
6     if(spte == NULL) return false;
7
8     spte->status = ON_SWAP;
9     spte->kpage = NULL;
10    spte->swap_index = swap_index;
11    return true;
12  }
```

Load data from src to frame

```
bool
vm_load_page(struct supplemental_page_table
*supt, uint32_t *pagedir, void *upage)
{
  /* see also userprog/exception.c */
  /* 1. Check if the memory reference is valid */
  struct supplemental_page_table_entry *spte;
  spte = vm_supt_lookup(supt, upage);
  if(spte == NULL) {
    return false;
  }
  if(spte->status == ON_FRAME) {
    /* already loaded */
    return true;
  }
  /* 2. Obtain a frame to store the page */
  void *frame_page = vm_frame_allocate(PAL_USER, upage);
  if(frame_page == NULL) {
    return false;
  }
  /* 3. Fetch the data into the frame */
  bool writable = true;
  switch (spte->status)
    {
      case ALL_ZERO:
        memset (frame_page, 0, PGSIZE);
        break;
      case ON_FRAME:
        /* nothing to do */
        break;
      case ON_SWAP:
        /* Swap in: load the data from the swap disc */
        vm_swap_in (spte->swap_index, frame_page);
        break;
      case FROM_FILESYS:
        if( vm_load_page_from_filesys(spte,
        frame_page) == false)
          {
            vm_frame_free(frame_page);
            return false;
          }
        writable = spte->writable;
        break;

      default:
        PANIC ("unreachable state");
    }

  /* 4. Point the page table entry for
  the faulting virtual address to the physical page. */
  if(!pagedir_set_page (pagedir, upage,
  frame_page, writable))
    {
      vm_frame_free(frame_page);
      return false;
    }
  /* Make SURE to mapped kpage is stored in the SPTE. */
  spte->kpage = frame_page;
  spte->status = ON_FRAME;
  pagedir_set_dirty (pagedir, frame_page, false);
  /* unpin frame */
  vm_frame_unpin(frame_page);
  return true;
}
```

Load from file

```
1   static bool vm_load_page_from_filesys(struct supplemental_page_table_entry *spte, void *kpage)
2   {
3     file_seek (spte->file, spte->file_offset);
4
5     /* read bytes from the file */
6     int n_read = file_read (spte->file, kpage, spte->read_bytes);
7     if(n_read != (int)spte->read_bytes)
8       return false;
9
10    /* remain bytes are just zero */
11    ASSERT (spte->read_bytes + spte->zero_bytes == PGSIZE);
12    memset (kpage + n_read, 0, spte->zero_bytes);
13    return true;
14  }
```

Clear map in frame, upage and kpage.

```
bool
vm_supt_mm_unmap(
    struct supplemental_page_table *supt, uint32_t *pagedir,
    void *page, struct file *f, off_t offset, size_t bytes)
{
  struct supplemental_page_table_entry *spte
  = vm_supt_lookup(supt, page);
  if(spte == NULL) {
    PANIC ("munmap - some page is missing; can't happen!");
  }
  /* Pin the associated frame if loaded
    otherwise, a page fault could occur while
    swapping in (reading the swap disk) */
  if (spte->status == ON_FRAME) {
    ASSERT (spte->kpage != NULL);
    vm_frame_pin (spte->kpage);
  }

  /* see also, vm_load_page() */
  switch (spte->status)
  {
    case ON_FRAME:
      ASSERT (spte->kpage != NULL);

      /* Dirty frame handling (write into file)
      Check if the upage or mapped frame is dirty.
      If so, write to file. */
      bool is_dirty = spte->dirty;
      is_dirty = is_dirty ||
      pagedir_is_dirty(pagedir, spte->upage);
      is_dirty = is_dirty ||
      pagedir_is_dirty(pagedir, spte->kpage);
      if(is_dirty)
        file_write_at (f, spte->upage, bytes, offset);
      /* clear the page mapping, and release the frame */
      vm_frame_free (spte->kpage);
      pagedir_clear_page (pagedir, spte->upage);
      break;

    case ON_SWAP: {
        bool is_dirty = spte->dirty;
        is_dirty = is_dirty
        || pagedir_is_dirty(pagedir, spte->upage);
        if (is_dirty) {
            /* load from swap, and write back to file */
            void *tmp_page = palloc_get_page(0);
            vm_swap_in (spte->swap_index, tmp_page);
            file_write_at (f, tmp_page, PGSIZE, offset);
            palloc_free_page(tmp_page);
          }
        else {
            /* just throw away the swap. */
            vm_swap_free (spte->swap_index);
          }
      }
      break;
    case FROM_FILESYS:
      /* do nothing. */
      break;

    default:
      /* Impossible, such as ALL_ZERO */
    PANIC ("unreachable state");
  }
    hash_delete(& supt->page_map, &spte->elem);
  return true;
}
```

SPTE destroy

```
1   static void
2   spte_destroy_func(struct hash_elem *elem, void *aux UNUSED)
3   {
4     struct supplemental_page_table_entry *entry = hash_entry(elem, struct supplemental_page_table_entry, elem);
5
6     /* Clean up the associated frame */
7     if (entry->kpage != NULL)
8       {
9         ASSERT (entry->status == ON_FRAME);
10        vm_frame_remove_entry (entry->kpage);
11      }
12    else if(entry->status == ON_SWAP)
13      {
14        vm_swap_free (entry->swap_index);
15      }
16
17    /* Clean up SPTE entry. */
18    free (entry);
19  }
```

Data structure and overview

```c
// swap.c
static struct block *swap_block;
static struct bitmap *swap_available;

static const size_t SECTORS_PER_PAGE = PGSIZE / BLOCK_SECTOR_SIZE;
// swap.h
/* the number of possible (swapped) pages. */
static size_t swap_size;
#ifndef VM_SWAP_H
#define VM_SWAP_H

typedef uint32_t swap_index_t;


/* Functions for Swap Table manipulation. */

/**
  Initialize the swap. Must be called ONLY ONCE at the initializtion phase.
 */
void vm_swap_init (void);

/**
  Swap Out: write the content of `page` into the swap disk,
  and return the index of swap region in which it is placed.
 */
swap_index_t vm_swap_out (void *page);

void vm_swap_in (swap_index_t swap_index, void *page);

void vm_swap_free (swap_index_t swap_index);

#endif /**< vm/swap.h */
```

Init

The swap block size can be assigned by cmd arg ”**–swap-size=4**”

The units is MB

```
1  // swap.c
2  static const size_t SECTORS_PER_PAGE = PGSIZE / BLOCK_SECTOR_SIZE;
3  void
4  vm_swap_init ()
5  {
6    ASSERT (SECTORS_PER_PAGE > 0); /* 4096/512 = 8? */
7
8    /* Initialize the swap disk */
9    swap_block = block_get_role(BLOCK_SWAP);
10   if(swap_block == NULL)
11   {
12     PANIC ("Error:␣Can't␣initialize␣swap␣block");
13     NOT_REACHED ();
14   }
15
16   /* Initialize swap_available, with all entry true
17    each single bit of `swap_available` corresponds to a block region,
18    which consists of contiguous [SECTORS_PER_PAGE] sectors,
19    their total size being equal to PGSIZE. */
20   swap_size = block_size(swap_block) / SECTORS_PER_PAGE;
21   swap_available = bitmap_create(swap_size);
22   bitmap_set_all(swap_available, true);
23 }
```

Swap in

```
1   /** Swap from swap slot to kernel virtual address. */
2   void
3   vm_swap_in (swap_index_t swap_index, void *page)
4   {
5     /* Ensure that the page is on kernel's virtual memory. */
6     ASSERT (page >= PHYS_BASE);
7
8     /* check the swap region */
9     ASSERT (swap_index < swap_size);
10    if (bitmap_test(swap_available, swap_index) == true) {
11      /* still available slot, error */
12      PANIC ("Error, invalid read access to unassigned swap block");
13    }
14
15    size_t i;
16    for (i = 0; i < SECTORS_PER_PAGE; ++ i)
17    {
18      block_read (swap_block,
19          /* sector number */  swap_index * SECTORS_PER_PAGE + i,
20          /* target address */ page + (BLOCK_SECTOR_SIZE * i)
21          );
22    }
23
24    bitmap_set(swap_available, swap_index, true);
25  }
```

Swap free

```
1  void
2  vm_swap_free (swap_index_t swap_index)
3  {
4    /* check the swap region */
5    ASSERT (swap_index < swap_size);
6    if (bitmap_test(swap_available, swap_index) == true)
7    {
8      PANIC ("Error, invalid free request to unassigned swap block");
9    }
10   bitmap_set(swap_available, swap_index, true);
11 }
```

Makefile.build

We should add vm files to Makefile.build

```
...
# Virtual memory code.
vm_SRC  = vm/frame.c        # Frame tables.
vm_SRC += vm/page.c         # Page tables.
vm_SRC += vm/swap.c         # Swap tables.
...
```

Task 1: Paging

Exercise 1.1

Implement paging for segments loaded from executables.

All of these pages should be loaded **lazily**, that is, only as the kernel intercepts page faults for them.

Upon eviction:

Pages **modified** since load (e.g. as indicated by the ”dirty bit”) should **be written to swap.**

Unmodified pages, including read-only pages, should never **be written to swap** because they can always be read back from the executable.

Data structure

```
1   struct thread
2   {
3       ...
4       uint8_t *current_esp; /**< The current value of the user program' s stack pointer.
5                                A  page fault might occur in the kernel, so we might
6                               need to store esp on transition to kernel mode. (4.3.3) */
7       ...
8   #ifdef VM
9      /* Project 3: Supplemental page table. */
10     struct supplemental_page_table *supt;   /**< Supplemental Page Table. */
11  #endif
12      ...
13  }
```

Init

```
1   // init.c
2   int
3   pintos_init (void) {
4       ...
5   #ifdef VM
6     /* Initialize Virtual memory system. (Project 3) */
7     vm_frame_init();
8   #endif
9       ...
10  #ifdef VM
11    /* Initialnize swap system (Project3). */
12    vm_swap_init ();
13  #endif
14      ...
15  }
16  // process.c
17  bool load (const char *file_name, void (**eip) (void), void **esp) {
18      ...
19  #ifdef VM
20    t->supt = vm_supt_create ();
21  #endif
22      ...
23  }
```

Free

```
1   void
2   process_exit (void)
3   {
4       ...
5   #ifdef VM
6       /* Destroy the SUPT, its all SPTEs, all the frames, and swaps.
7        Important: All the frames held by this thread should ALSO be freed
8        (see the destructor of SPTE). Otherwise an access to frame with
9        its owner thread had been died will result in fault. */
10      vm_supt_destroy (cur->supt);
11      cur->supt = NULL;
12  #endif
13      ...
14  }
```

Solution to Exercise 1.1

```c
// process.c
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
    ...
#ifdef VM
      /* Lazy load */
      struct thread *curr = thread_current ();
      ASSERT (pagedir_get_page(curr->pagedir,
      upage) == NULL); /**< no virtual page yet? */

      if (! vm_supt_install_filesys(curr->supt, upage,
            file, ofs, page_read_bytes,
            page_zero_bytes, writable) ) {
        return false;
      }
#else
      /* Get a page of memory. */
      uint8_t *kpage = vm_frame_allocate
      (PAL_USER, upage);
      if (kpage == NULL)
        return false;
      /* Load this page. */
      if (file_read (file, kpage, page_read_bytes) !=
      (int) page_read_bytes)
        {
          vm_frame_free (kpage);
          return false;
        }
      memset (kpage + page_read_bytes, 0,
      page_zero_bytes);

      /* Add the page to the process's address space. */
      if (!install_page (upage, kpage, writable))
        {
          palloc_free_page (kpage);
          return false;
        }
#endif
      /* Advance. */
      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      upage += PGSIZE;
#ifdef VM
      ofs += PGSIZE;
#endif
    }
  return true;
}
```

Solution to Exercise 1.1

```c
// exception.c
#define MAX_STACK_SIZE 0x800000
static void
page_fault (struct intr_frame *f)
{
    ...
#if VM

  struct thread *curr = thread_current();
  void* fault_page = (void*) pg_round_down(fault_addr);

  if (!not_present) {
    /* attempt to write to a read-only
    region is always killed. */
    goto PAGE_FAULT_VIOLATED_ACCESS;
  }

  void* esp = user ? f -> esp : curr -> current_esp;

  /* Stack Growth */
  bool on_stack_frame, is_stack_addr;
  on_stack_frame = (esp <= fault_addr ||
                fault_addr == f->esp - 4 || /**< PUSH */
                fault_addr == f->esp - 32); /**< PUSHA */
  is_stack_addr = (PHYS_BASE - MAX_STACK_SIZE
  <= fault_addr && fault_addr < PHYS_BASE);
  if (on_stack_frame && is_stack_addr)
  {
    if (vm_supt_has_entry(curr->supt, fault_page) == false)
      vm_supt_install_zeropage (curr->supt, fault_page);
  }
```

```c
  if(! vm_load_page(curr->supt, curr->pagedir, fault_page) )
      goto PAGE_FAULT_VIOLATED_ACCESS;

    /* success */
    return;

  PAGE_FAULT_VIOLATED_ACCESS:
  #endif
    /* (3.1.5) a page fault in the kernel merely sets
      eax to 0xffffffff
      and copies its former value into eip */
     if(!user) {
        f->eip = (void *) f->eax;
        f->eax = 0xffffffff;
        return;
     }
    printf ("Page␣fault␣at␣%p:␣%s␣error␣%s
  ␣␣page␣in␣%s␣context.\n",
            fault_addr,
            not_present ? "not␣present" :
            "rights␣violation",
            write ? "writing" : "reading",
            user ? "user" : "kernel");
    kill (f);
  }
```

Exercise 1.2

Implement a global page replacement algorithm that approximates LRU.

Your algorithm should perform **at least as well as** the simple variant of the ”second chance” or ”clock” algorithm.

Solution to Exercise 1.2

See **pick_frame_to_evict ()** in frame.c

Task 2: Accessing User Memory

Exercise 2.1

Adjust user memory access code in system call handling to deal with potential page faults.

You will need to adapt your code to access user memory (see section 3 Accessing User Memory in project 2) **while handling a system call.**

While accessing user memory, your kernel must either be prepared to handle such page faults, or it must prevent them from occurring. The kernel must prevent such page faults while it is **holding resources** it would need to acquire to handle these faults.

Solution to Exercise 2.1

```
1  // syscall.c
2  static void
3  syscall_handler (struct intr_frame *f UNUSED) {
4      ...
5      /* Store the esp, which is needed in the page fault handler.
6     refer to exception.c:page_fault() (see manual 4.3.3) */
7    thread_current() -> current_esp = f -> esp;
8    ...
9  }
10 int
11 sys_read(int fd, void *buffer, unsigned size) {
12     ...
13     #ifdef VM
14           preload_and_pin_pages(buffer, size);
15 #endif
16           ret = file_read(file_d->file, buffer, size);
17 #ifdef VM
18           unpin_preloaded_pages(buffer, size);
19 #endif
20     ...
21 }
22 int sys_write(int fd, const void *buffer, unsigned size) {
23     ...
24     #ifdef VM
25           preload_and_pin_pages(buffer, size);
26 #endif
27
28           ret = file_write(file_d->file, buffer, size);
29
30 #ifdef VM
31           unpin_preloaded_pages(buffer, size);
32 #endif
33     ...
34 }
```

Auxiliary funcs

```
1  void preload_and_pin_pages(const void *buffer, size_t size)
2  {
3    struct supplemental_page_table *supt = thread_current()->supt;
4    uint32_t *pagedir = thread_current()->pagedir;
5
6    void *upage;
7    for(upage = pg_round_down(buffer); upage < buffer + size; upage += PGSIZE)
8    {
9      vm_load_page (supt, pagedir, upage);
10     vm_pin_page (supt, upage);
11   }
12 }
13
14 void unpin_preloaded_pages(const void *buffer, size_t size)
15 {
16   struct supplemental_page_table *supt = thread_current()->supt;
17
18   void *upage;
19   for(upage = pg_round_down(buffer); upage < buffer + size; upage += PGSIZE)
20   {
21     vm_unpin_page (supt, upage);
22   }
23 }
```

Task 3: Stack Growth

Exercise 3.1

Implement stack growth.

In project 2, the stack was **a single page** at the top of the user virtual address space, and programs were limited to that much stack.

Now, **if the stack grows past its current size, allocate additional pages as necessary.**

PUSH and PUSHA

You should impose some absolute limit on stack size, as do most OSes.

The first stack page need not be allocated lazily.

Solution to Exercise 3.1

stack growth and stack size: See page_fault () in exception.c

first stack page: load () -> setup_stack (esp) -> install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);

```
1  static bool
2  install_page (void *upage, void *kpage, bool writable)
3  {
4    struct thread *t = thread_current ();
5
6    /* Verify that there's not already a page at that virtual
7       address, then map our page there. */
8       bool success = (pagedir_get_page (t->pagedir, upage) == NULL)
9                       && pagedir_set_page (t->pagedir, upage, kpage, writable);
10    #ifdef VM
11      success = success && vm_supt_install_frame (t->supt, upage, kpage);
12      if(success) vm_frame_unpin(kpage);
13    #endif
14      return success;
15 }
```

Task4: Memory Mapped Files

Exercise 4.1

Implement memory mapped files, including the following system calls.

mapid_t **mmap** (int fd, void *addr)

void **munmap** (mapid_t mapping)

If two or more processes map the same file, there is no requirement that they see consistent data.

Data structure, init and clear

```
1  // process.h
2  #ifdef VM
3  typedef int mmapid_t;
4  struct mmap_desc {
5    mmapid_t id;
6    struct list_elem elem;
7    struct file* file;
8    void *addr;   /**< where it is mapped to? store the user virtual address. */
9    size_t size;  /**< file size */ };
10 #endif
11 // thread.h
12 struct thread {
13     ...
14     struct list mmap_list;                  /**< List of struct mmap_desc. */
15     ... };
16 // thread.c
17 static void init_thread (struct thread *t, const char *name, int priority) {
18 ...
19 #ifdef VM
20   list_init(&t->mmap_list);
21 #endif }
22 void process_exit (void) {
23 ...
24 #ifdef VM
25   /* mmap descriptors */
26   struct list *mmlist = &cur->mmap_list;
27   while (!list_empty(mmlist)) {
28     struct list_elem *e = list_begin (mmlist);
29     struct mmap_desc *desc = list_entry(e, struct mmap_desc, elem);
30     ASSERT( sys_munmap (desc->id) == true );
31   }
32 #endif
33 ...
34 }
```

syscall_handler

```
1   static void
2   syscall_handler (struct intr_frame *f UNUSED) {
3       ...
4   #ifdef VM
5       case SYS_MMAP:
6         {
7           int fd;
8           void *addr;
9           memread_user(f->esp + 4, &fd, sizeof(fd));
10          memread_user(f->esp + 8, &addr, sizeof(addr));
11
12          mmapid_t ret = sys_mmap (fd, addr);
13          f->eax = ret;
14          break;
15        }
16
17      case SYS_MUNMAP: // 14
18        {
19          mmapid_t mid;
20          memread_user(f->esp + 4, &mid, sizeof(mid));
21
22          sys_munmap(mid);
23          break;
24        }
25  #endif
26  ...
27  }
```

sys_mmap

```
mmapid_t sys_mmap(int fd, void *upage) {
  if (upage == NULL || pg_ofs(upage) != 0) return -1;
  if (fd <= 1) return -1;
  struct thread *curr = thread_current();
  lock_acquire (&filesys_lock);
  struct file *f = NULL;
  struct file_desc* file_d =
  find_file_desc(thread_current(), fd);
  if(file_d && file_d->file) {
      f = file_reopen (file_d->file);
    }
  if(f == NULL) {
      lock_release (&filesys_lock);
      return -1;
    }
  size_t file_size = file_length(f);
  if(file_size == 0) {
      lock_release (&filesys_lock);
      return -1;
    }
  size_t offset;
  for (offset = 0; offset <
  file_size; offset += PGSIZE) {
      void *addr = upage + offset;
      if (vm_supt_has_entry(curr->supt, addr))
        {
          lock_release (&filesys_lock);
          return -1;
        }
    }
```

```
  for (offset = 0; offset < file_size; offset += PGSIZE)
   {
      void *addr = upage + offset;

      size_t read_bytes = (offset + PGSIZE < file_size ?
      PGSIZE : file_size - offset);
      size_t zero_bytes = PGSIZE - read_bytes;

      vm_supt_install_filesys(curr->supt, addr,
          f, offset, read_bytes, zero_bytes,
          true/**< writable */);
    }
  mmapid_t mid;
  if (! list_empty(&curr->mmap_list)) {
    mid = list_entry(list_back(&curr->mmap_list),
    struct mmap_desc, elem)->id + 1;
  }
  else mid = 1;
  struct mmap_desc *mmap_d = (struct mmap_desc*)
  malloc(sizeof(struct mmap_desc));
  mmap_d->id = mid;
  mmap_d->file = f;
  mmap_d->addr = upage;
  mmap_d->size = file_size;
  list_push_back (&curr->mmap_list, &mmap_d->elem);
  lock_release (&filesys_lock);
  return mid;
}
```

sys_munmap

```
1   bool sys_munmap(mmapid_t mid)
2   {
3     struct thread *curr = thread_current();
4     struct mmap_desc *mmap_d = find_mmap_desc(curr, mid);
5
6     if(mmap_d == NULL) { /**< not found such mid. */
7       return false;
8     }
9
10    lock_acquire (&filesys_lock);
11    {
12      /* Iterate through each page */
13      size_t offset, file_size = mmap_d->size;
14      for(offset = 0; offset < file_size; offset += PGSIZE) {
15        void *addr = mmap_d->addr + offset;
16        size_t bytes = (offset + PGSIZE < file_size ? PGSIZE : file_size - offset);
17        vm_supt_mm_unmap (curr->supt, curr->pagedir, addr, mmap_d->file, offset, bytes);
18      }
19
20      /* Free resources, and remove from the list. */
21      list_remove (& mmap_d->elem);
22      file_close (mmap_d->file);
23      free (mmap_d);
24    }
25    lock_release (&filesys_lock);
26
27    return true;
28  }
```

Auxiliary func

```
1   static struct mmap_desc*
2   find_mmap_desc(struct thread *t, mmapid_t mid)
3   {
4     ASSERT (t != NULL);
5
6     struct list_elem *e;
7
8     if (! list_empty(&t->mmap_list)) {
9       for(e = list_begin(&t->mmap_list);
10          e != list_end(&t->mmap_list); e = list_next(e))
11      {
12        struct mmap_desc *desc = list_entry(e, struct mmap_desc, elem);
13        if(desc->id == mid) {
14          return desc;
15        }
16      }
17    }
18
19    return NULL; // not found
20  }
```

Results

A strange problem

pintos -v -k -T 60 –qemu –filesys-size=2 -p tests/vm/page-parallel -a page-parallel -p tests/vm/child-linear -a child-linear –swap-size=4 – -q -f run page-parallel < /dev/null 2> tests/vm/page-parallel.errors > tests/vm/page-parallel.output

```
FAIL tests/vm/page-parallel
```

But if we try again, it would **pass**.

Results

```
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 113 tests passed.
```

```
u2204@ubuntu2204:~/pintos_lab/pintos$ git diff --stat fea7ba6bc4a3188b6560c1d7
1f97bd256fbfda9e156c0ab8d361
 src/Makefile.build        |   6 +-
 src/threads/init.c        |  13 +++
 src/threads/thread.c      |   3 +
 src/threads/thread.h      |  16 +++
 src/userprog/exception.c  |  57 ++++++++++-
 src/userprog/process.c    |  68 ++++++++++--
 src/userprog/process.h    |  13 +++
 src/userprog/syscall.c    | 207 ++++++++++++++++++++++++++++++++++++++-
 src/userprog/syscall.h    |   6 +-
 src/vm/frame.c            | 294 ++++++++++++++++++++++++++++++++++++++++++++++
 src/vm/frame.h            |  21 ++++
 src/vm/page.c             | 404 ++++++++++++++++++++++++++++++++++++++++++++++++
 src/vm/page.h             |  76 +++++++++++
 src/vm/swap.c             |  95 +++++++++++++++
 src/vm/swap.h             |  31 +++++
 15 files changed, 1296 insertions(+), 14 deletions(-)
```

Thank you!
Questions?