OS Project 4 - Team 4

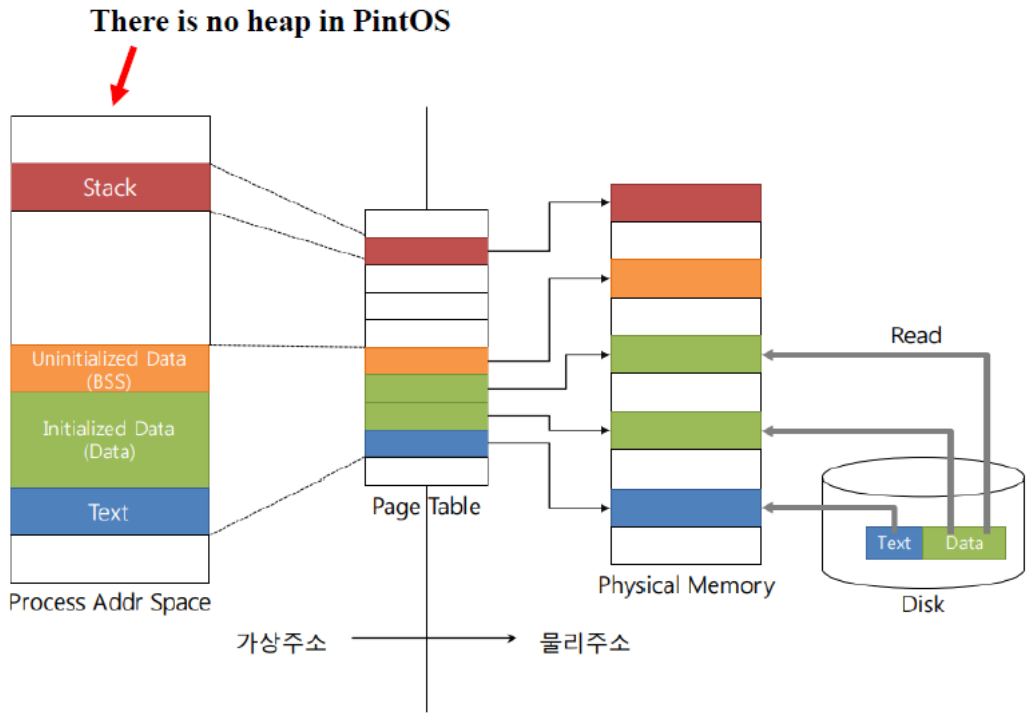
20185083 서영석, 20185141 이준명, 20185010 권강민

1. Introduction

Project 4 requires the implementation of various features related to virtual memory. Currently, Pintos uses a direct mapping approach, where memory addresses are directly mapped to physical memory addresses when allocating memory for processes. However, this method can lead to unsafe memory accesses and lacks of support for swapping, resulting in limitations on physical memory space during process execution. Additionally, it does not allow for memory sharing through pages. To address these issues related to memory usage, the goal of this project is to implement functionalities such as demand paging, stack growth, and mmap, which are all associated with virtual memory.

2. Overview

A. Paging (Problem 1)



Pintos has a basic VM implementation using palloc and pagedir. However, this VM can only store pages that are currently loaded in memory. This means that when memory becomes full, the program cannot continue running.

So, we should implement demand paging, which is a virtual memory management technique. With demand paging, only the necessary pages of a program are loaded into physical memory when they are required. Instead of loading the entire code and data of a program into memory at once, it is divided into pages and loaded on-demand.

We have two challenges for paging.

A-1. Page&Frame management

To implement virtual memory, the first step is to map physical memory to virtual pages.

A page is a unit of virtual memory that represents a fixed-size portion of the program's address space. The program's address space is divided into pages, and each page has a unique page number. In Pintos, each page is 4kB (4096 bytes) in size.

A frame is a unit of physical memory where pages are stored. Frames have the same size as pages and are one-to-one mapped with pages. In a real operating system, the mapping relationship between virtual pages and physical frames is

maintained in the page table. In this project, a hash table for `vm_entry` is used per process as a replacement for the functionality of the page table.

The main objectives of this section are as follows:

1. Implement data structure for proper page management
2. Implement demand paging logic for handling page fault

A-2. Swapping

The goal of this task is to implement a page replacement algorithm in the Pintos operating system that approximates the Least Recently Used (LRU) algorithm. The page replacement algorithm is responsible for determining which pages should be swapped out from physical memory to the disk's swap area and which pages should be swapped in from the disk to physical memory. We implemented “second chance” swapping algorithm which are commonly used page replacement algorithms

The main objectives of this section are as follows:

1. Implement page swapping with disk (swap in&out with swapping partition of PintOS)
2. Implement page swapping algorithm (w. second-chance algorithm)

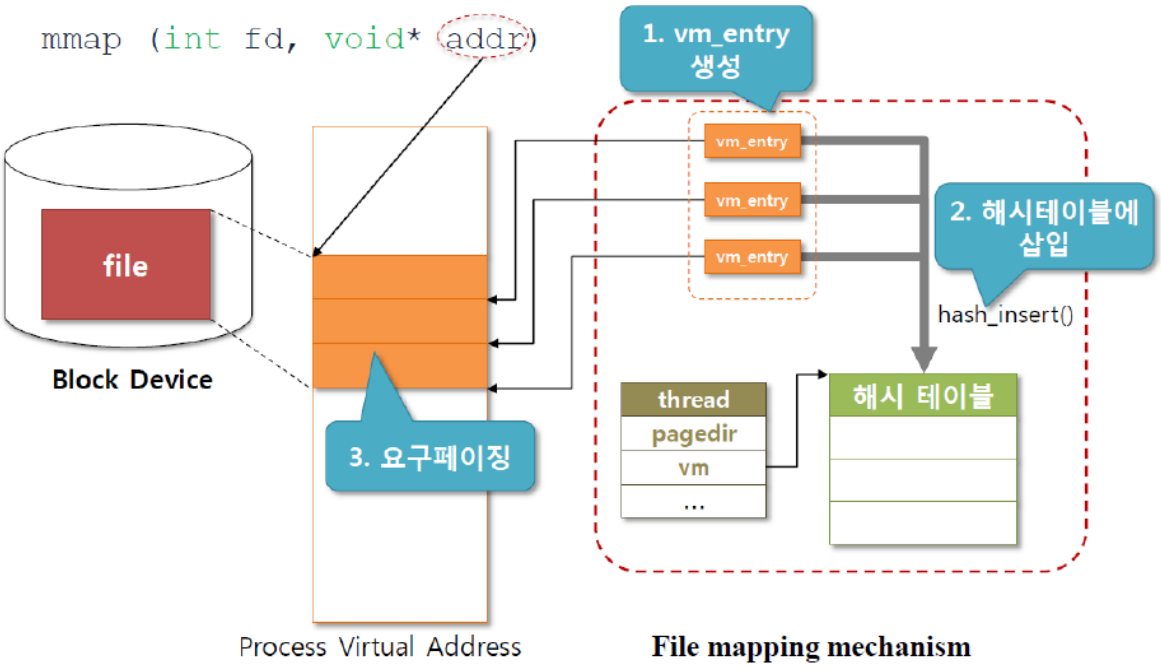
B. Stack growth (Problem 2)

In the previous project, the stack in the Pintos operating system was limited to a single page at the top of the user virtual address space. To address this limitation, we need to allocate additional pages to the stack when it exceeds its current size. Therefore, our goal is to modify the stack handling mechanism in Pintos using the previously implemented paging system to support stack expansion.

The maximum stack size in Pintos is 8MB. Hence, we aim to dynamically expand the stack from its initial size of 4KB up to a maximum of 8MB. This improvement aims to prevent stack overflow errors and allow the stack to adjust dynamically to the requirements of the running program, enabling more efficient memory allocation.

Therefore, when a stack access exceeds the current stack size, appropriate handling should be performed within the page fault function based on the address. If the access falls within a valid range, the stack size should be expanded to accommodate the access. However, if the access is invalid and exceeds the allowed stack range or stack pointer, a segmentation fault should be triggered to notify the error.

C. Memeory-mapped file(Problem 3)



In the current state of the Pintos operating system, only read/write system calls are supported for loading files. In order to leverage the benefits of virtual memory, it is necessary to implement the usage of memory-mapped files.

The mmap operation is responsible for mapping a file, opened with a file descriptor (fd), into the virtual address space of a process. Pages within the mmap region should be lazily loaded, and when the mapped file is swapped out, any modifications should be checked and reflected back to the file on disk.

The munmap operation is used to unmap a mapping identified by the mapping ID returned by a previous mmap call. Therefore, in addition to implementing the mmap system call, additional data structures need to be implemented.

D. Accessing User Memory(Problem 4)

Currently, in our Pintos project, we encounter a problem where page faults occur when kernel code accesses non-resident user pages. To address this issue, we need to modify our kernel code to handle the access to user memory while handling system calls. The kernel must be prepared to handle such page faults or prevent them from occurring when accessing user memory. Our objective is to adapt our code to effectively handle or prevent page faults during user memory access while handling system calls.

3. Code-level development

A-1. Page management

In real operating systems, a Page Table is used to manage the mapping between pages and frames. However, in our project, instead of using a global page table, we implemented page management using a `vm_entry` structure corresponding to the page table entry and a hash table that contains these structures.

In our implementation, pages are managed locally on a per-process basis. We included a hash table `vm_list` in the thread structure that holds `vm_entry` as an element.

- `src/vm/page.h`

```
enum page_status {
  PAGE_STACK_UNINIT,
  PAGE_STACK_INMEM,
  PAGE_STACK_SWAPPED,
  PAGE_FILE_INDISK,
  PAGE_FILE_INMEM,
  PAGE_FILE_SWAPPED,
  PAGE_MMAP_INDISK,
  PAGE_MMAP_INMEM,
  PAGE_UNEXPECTED
};

/*...other page codes...*/

struct vm_entry{
    uint32_t vm_address;
    struct hash_elem vm_list_elem;
    enum page_status entry_status;
    struct swap_pool * stored_swap_pool;
    block_sector_t stored_swap_sector;
    struct file * swap_file;
    off_t swap_file_offset;
    off_t file_left_size;
    bool is_file_writable;
};
```

- `src/threads/thread.h`

```
 struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
/*...other thread codes...*/
//added part
    struct hash vm_list;
    struct hash_iterator vm_LRU_iterator;
}
```

B. Stack Growth

As mentioned earlier, the initial stack size of a process is one page, which is 4KB. Therefore, as the process runs, the stack area needed for execution can grow. Thus, it is necessary to allocate new pages for accesses beyond the originally allocated page range.

In PintOS, when a process attempts to access a page that has not been allocated to it, a page fault is triggered, and the execution context is trapped. When a page fault error occurs, the execution context is transferred to the `page_fault()` function defined in `exception.c`, where the allocation of a new page for the stack takes place.

During page allocation, it is important to verify whether the process is accessing a valid region. Only when the access is valid, a new page should be allocated.

The following conditions can lead to incorrect accesses:

1. If the requested address falls within the maximum stack size range (PHYS_BASE - MAX_STACK_SIZE), the `vm_check_stack_vm_entry()` function is called to verify if the address is a valid stack access by checking target address in larger than stack pointer. If it is valid, it allocate new `vm_entry` with status `PAGE_STACK_UNINIT`. If it is not a valid stack access, the system will exit with a status value of -1.
    - `vm_check_stack_vm_entry()` function:
2. If a kernel thread triggers a fault for a user-space address.
3. If the requested address is not in the user space and exceeds PHYS_BASE (kernel space).
4. If the page is not writable, but a write operation is attempted.
5. If the virtual address is not assigned to the current thread.

After `vm_check_stack_vm_entry` produces uninitialized `vm_entry`, the actual stack growth occurs through the `vm_handle_syscall_alloc()` function.

- `userprog/exception.c`

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  bool isHandled = false;
  void *fault_addr;  /* Fault address. */

....
 struct thread * cur_thre = thread_current();
  uint32_t *pd = cur_thre->pagedir;
  bool is_writable_page =pagedir_is_writable(pd,fault_addr);
  struct vm_entry * found_entry ;//= find_vm_entry_from(cur_thre,fault_addr);
  if(user && fault_addr >= PHYS_BASE - MAX_STACK_SIZE){
    if(!vm_check_stack_vm_entry(cur_thre,f,fault_addr,1)){
      sys_exit(-1);
    }
  }
  if(!user && !is_user_page) { // kernel mode fault
    f->eip = (void *) f->eax;
    f->eax = 0xffffffff;
    return;
  }
  else if(!user && is_user_page){
    sys_exit(-1);
  }
  else if(user && fault_addr >= PHYS_BASE){
    sys_exit(-1);
  }
  else if(!not_present && !is_writable_page && write){// instruction (null part) try write pt-write-code2
    sys_exit(-1);
  }
  else if(!(found_entry = find_vm_entry_from(cur_thre,fault_addr))){ //invalid access
    sys_exit(-1);
  }
  bool alloc_result;
  bool retry_alloc_result;
  bool is_upper_stack = fault_addr >= f->esp -32;
  switch(found_entry->entry_status){
    case PAGE_STACK_UNINIT:
      if(is_upper_stack){
        //print_entry_info(found_entry);
        alloc_result = vm_handle_syscall_alloc(thread_current(),f,fault_addr,1);
        if(alloc_result == false){
          printf("stack_alloc_fail at pagefault_PAGE_STACK_UNINIT\N");
          //print_entry_info(found_entry);
```

```
          sys_exit(-1);
        }
        isHandled = true;
      }
/*...other codes*/
```

- `vm/page.c`

```
bool vm_check_stack_vm_entry(struct thread * target_thread, struct intr_frame * f, uint8_t* addr, uint32_t byte_to_handle){
  uint8_t * target_addr = pg_round_down(addr);
  uint8_t * max_addr = pg_round_down(addr + byte_to_handle - 1);
  bool is_upper_stack = (addr >= (f->esp - 32));
  if(!is_upper_stack && max_addr >= PHYS_BASE){
    return false;
  }
  while(target_addr <= max_addr){
    struct vm_entry * found_entry = find_vm_entry_from(target_thread,target_addr);
    if(!found_entry){
        bool is_alloc = add_new_vm_entry_at(target_thread,PAGE_STACK_UNINIT,target_addr);
        if(!is_alloc){
          printf("add_new_vm_entry failed at vm_check_stack_vm_entry\n");
          return false;
        }
    }
    target_addr += PGSIZE;
  }
  return true;
}

bool vm_handle_syscall_alloc(struct thread * target_thread, struct intr_frame *f, uint8_t* addr, uint32_t byte_to_handle){//TODO need
  vm_check_stack_vm_entry(target_thread,f,addr,byte_to_handle);
  uint8_t* target_addr = pg_round_down(addr);
  uint8_t* max_addr = pg_round_down(addr + byte_to_handle-1);
  while(target_addr <= max_addr){
    struct vm_entry * found_entry = find_vm_entry_from(target_thread,target_addr);
    if(found_entry == NULL){
      printf("vm_handle_stack_alloc_failed: reason - cannot find vm_entry from vm_list : addr - %x",addr);
      return false;
    }
    switch(found_entry->entry_status){
      case PAGE_STACK_UNINIT:{
        bool is_upper_stack  = (addr >= (f->esp - 32));
        if(is_upper_stack){
          vm_install_new_stack(target_thread,found_entry);
        }
        else{
          return false;
        }
        break;
      }
      case PAGE_STACK_SWAPPED:
 ...
 return true;
}
```

A-2. Swapping

To implement swapping, we defined the `entry_status` to allow `vm_entry` to have multiple statuses.

The core logic for swapping is included in `vm/page.c`.

- `src/userprog/exception.c`

```
static void
page_fault (struct intr_frame *f)
{
...
  bool alloc_result;
  bool retry_alloc_result;
  bool is_upper_stack = fault_addr >= f->esp -32;
  switch(found_entry->entry_status){
    case PAGE_STACK_UNINIT:
      if(is_upper_stack){
        //print_entry_info(found_entry);
        alloc_result = vm_handle_syscall_alloc(thread_current(),f,fault_addr,1);
        if(alloc_result == false){
          printf("stack_alloc_fail at pagefault_PAGE_STACK_UNINIT\N");
          //print_entry_info(found_entry);
          sys_exit(-1);
        }
```

```
        isHandled = true;
      }
      break;
    case PAGE_STACK_SWAPPED:
    case PAGE_MMAP_INDISK:
    case PAGE_FILE_INDISK:
    case PAGE_FILE_SWAPPED:
      alloc_result =vm_swap_in_page(thread_current(),found_entry);
      if(alloc_result == false){
        vm_swap_out_LRU(thread_current());
        retry_alloc_result = vm_swap_in_page(thread_current(),found_entry);
        if(retry_alloc_result == false){
          sys_exit(-1);
        }
      }
      isHandled = true;
      break;
    default:
      printf("unexpected status at page fault\n");
      //print_entry_info(found_entry);
      break;
  }
...
}
```

- `vm/page.c`

```
...

bool vm_swap_out_page(struct thread * target_thread, struct vm_entry * target_entry){
  ASSERT(is_entry_inmem(target_entry));
  uint32_t * pd = target_thread->pagedir;
  uint32_t targetAddr = target_entry->vm_address;
  uint32_t * target_paddr = pagedir_get_page(pd,targetAddr);
  if(target_paddr == (uint32_t *)NULL){
    printf("In vm swap out page : finding physical address of target in pagedirectory is failed\n");
    return false;
  }
  ASSERT(target_entry);
  struct swap_pool * target_pool = NULL;
  switch(target_entry->entry_status){
    case PAGE_STACK_INMEM:
      vm_swap_init();
      target_pool = page_swap_pool;
      ASSERT(target_pool);
      block_sector_t need_block_size = PGSIZE/ BLOCK_SECTOR_SIZE;
      lock_acquire(&target_pool->swap_lock);
      block_sector_t block_index = bitmap_scan_and_flip(target_pool->used_map,0,need_block_size,0);
      lock_release(&target_pool->swap_lock);
      if(block_index == BITMAP_ERROR){
        return false;
      }
      else {
        int i;
        for(i = 0; i < need_block_size;i++){
          block_write(target_pool->target_block,block_index + i,targetAddr + BLOCK_SECTOR_SIZE * i);
        }
        target_entry->stored_swap_pool = page_swap_pool;
        target_entry->stored_swap_sector = block_index;
      }
      target_entry->entry_status = PAGE_STACK_SWAPPED;
      break;
    case PAGE_FILE_INMEM:
      if(target_entry->is_file_writable){
        vm_swap_init();
        target_pool = page_swap_pool;
        block_sector_t need_block_size = PGSIZE/ BLOCK_SECTOR_SIZE;
        lock_acquire(&target_pool->swap_lock);
        block_sector_t block_index = bitmap_scan_and_flip(target_pool->used_map,0,need_block_size,0);
        lock_release(&target_pool->swap_lock);
        if(block_index == BITMAP_ERROR){
          return false;
        }
        else {
          int i;
          for(i = 0; i < need_block_size;i++){
            block_write(target_pool->target_block,block_index + i,targetAddr + BLOCK_SECTOR_SIZE * i);
          }
          target_entry->stored_swap_pool = page_swap_pool;
          target_entry->stored_swap_sector = block_index;
        }
        target_entry->entry_status = PAGE_FILE_SWAPPED;
      }
      else{
```

```
        target_entry->entry_status = PAGE_FILE_INDISK;
      }
      break;
    case PAGE_MMAP_INMEM:
      if(pagedir_is_dirty(target_thread->pagedir,target_entry->vm_address)){
        file_write_at(target_entry->swap_file,(void *)target_entry->vm_address,target_entry->file_left_size,target_entry->swap_file_of
      }
      target_entry->entry_status = PAGE_MMAP_INDISK;
      break;
    default:
     printf("unexpected entry status at swap out- ");
     print_status(target_entry->entry_status);
     return false;
     break;

  }
  pagedir_clear_page(target_thread->pagedir,target_entry->vm_address);
  palloc_free_page(target_paddr);
  return true;
}

bool vm_swap_in_page(struct thread * target_thread, struct vm_entry * target_entry){
  ASSERT(is_entry_inblock(target_entry));//assert target entry is anon entry that once loaded or it came from file.
  uint8_t * ppage = palloc_get_page(PAL_USER|PAL_ZERO);
  if(ppage == NULL){
    return false;
  }

  bool writable;
  int need_block_size = PGSIZE/BLOCK_SECTOR_SIZE;
  switch(target_entry->entry_status){
    case PAGE_FILE_INDISK:{
      size_t read_length = file_read_at(target_entry->swap_file,ppage,target_entry->file_left_size,target_entry->swap_file_offset);
      ASSERT(read_length == target_entry->file_left_size);
      size_t page_zero_bytes = PGSIZE - target_entry->file_left_size;
      memset (ppage + target_entry->file_left_size, 0, page_zero_bytes);
      target_entry->entry_status = PAGE_FILE_INMEM;
      writable = target_entry->is_file_writable;
      break;
    }
    case PAGE_FILE_SWAPPED:{
      struct swap_pool * target_pool = target_entry->stored_swap_pool;
      block_sector_t target_block_sector = target_entry->stored_swap_sector;
      struct block * target_block = target_pool->target_block;
      int i = 0;
      for(i = 0; i < need_block_size; i ++){
        block_read(target_block,target_block_sector + i, ppage+BLOCK_SECTOR_SIZE * i);
      }
      lock_acquire(&target_pool->swap_lock);
      int need_block_size = PGSIZE/BLOCK_SECTOR_SIZE;
      bitmap_set_multiple(target_pool->used_map,target_block_sector,need_block_size,0);
      lock_release(&target_pool->swap_lock);
      target_entry->entry_status = PAGE_FILE_INMEM;
      writable = target_entry->is_file_writable;
      break;
    }
    case PAGE_STACK_SWAPPED:{
      struct swap_pool * target_pool = target_entry->stored_swap_pool;
      block_sector_t target_block_sector = target_entry->stored_swap_sector;
      struct block * target_block = target_pool->target_block;
      int i = 0;
      for(i = 0; i < need_block_size; i ++){
        block_read(target_block,target_block_sector + i, ppage+BLOCK_SECTOR_SIZE * i);
      }
      lock_acquire(&target_pool->swap_lock);
      int need_block_size = PGSIZE/BLOCK_SECTOR_SIZE;
      bitmap_set_multiple(target_pool->used_map,target_block_sector,need_block_size,0);
      lock_release(&target_pool->swap_lock);
      target_entry->entry_status = PAGE_STACK_INMEM;
      writable = true;
      break;
    }
    case PAGE_MMAP_INDISK:{
      size_t read_length = file_read_at(target_entry->swap_file,ppage,target_entry->file_left_size,target_entry->swap_file_offset);
      ASSERT(read_length == target_entry->file_left_size);
      size_t page_zero_bytes = PGSIZE - target_entry->file_left_size;
      memset (ppage + target_entry->file_left_size, 0, page_zero_bytes);
      target_entry->entry_status = PAGE_MMAP_INMEM;
      writable = true;
      break;
    }
    default:
      printf("unexpected state at swap_in\n");
      return false;
      break;
  }
      pagedir_set_page(target_thread->pagedir,target_entry->vm_address,ppage,writable);
```

```
  return true;
}
...
```

1. Swap In

The logic for swapping in a page is implemented in the `vm_swap_in_page` function. This function swaps the page from disk into physical memory.

The `vm_swap_in_page` function handles various cases based on the `entry_status` of the `vm_entry`:

1. `PAGE_FILE_INDISK`: If the swap-in data is located in the file system partition.
2. `PAGE_FILE_SWAPPED`: If the page is swapped out and stored in the swap partition.
3. `PAGE_STACK_SWAPPED`: If the page allocated for the stack is swapped out.
4. `PAGE_MMAP_INDISK`: If the page needs to be loaded from a file specified by the mmap system call.

The `default` case is triggered when an unexpected state occurs. It prints an error and returns `false`.

2. Swap Out

The logic for swapping out a page is implemented in the `vm_swap_out_page` function. Similar to the swap-in process, it updates the bitmap that represents the usage of blocks after swapping the page to the swap pool.

The `vm_swap_out_page` function handles the following cases based on the `entry_status` of the `vm_entry`:

1. `PAGE_STACK_INMEM`: If a page allocated for the stack is selected.
2. `PAGE_FILE_INMEM`: If a page allocated outside of the stack area is selected.
3. `PAGE_MMAP_INMEM`: If a page allocated for a file specified by the mmap system call needs to be swapped out.

By checking the status of the `vm_entry`, the appropriate swap-in/out operation is performed.

When performing swap out, the approximately LRU algorithm called the one-chance algorithm is used to find the victim page. It involves traversing the hash data structure and iterating through all the `vm_entry` structures to check if a page has been referenced. If the access flag is 0, it is selected as the victim page. The check for whether a page has been referenced is done using the `pagedir_is_accessed` function.

- `userprog/pagedir.c`

```
bool
pagedir_is_accessed (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_A) != 0;
}
```

- `vm/page.h`

```
bool vm_swap_out_LRU_Global(){
    struct list_elem *e;
    for (e = list_begin (&all_list); e != list_end (&all_list);e = list_next (e)){
      struct thread *target_thread = list_entry (e, struct thread, allelem);
      if(vm_swap_out_LRU(target_thread)){
        return true;
      }
    }
    return false;
}

bool vm_swap_out_LRU(struct thread * target_thread){
  struct hash_elem * iter_elem;
  if(hash_size(&target_thread->vm_list) == 0){
    return false;
  }
  if(target_thread->vm_LRU_iterator.hash ==NULL){
    hash_first(&target_thread->vm_LRU_iterator,&target_thread->vm_list);
    iter_elem = hash_cur(&target_thread->vm_LRU_iterator);
  }
  else{
    iter_elem = hash_next(&target_thread->vm_LRU_iterator);
```

```
  }
  if(iter_elem == NULL){
    hash_first(&target_thread->vm_LRU_iterator,&target_thread->vm_list);
    iter_elem = hash_cur(&target_thread->vm_LRU_iterator);
  }
  int roll_index = 0;
  while(roll_index < 2){
    struct vm_entry * target_entry = hash_entry(iter_elem,struct vm_entry, vm_list_elem);
    if(is_entry_inmem(target_entry)){
      if(pagedir_is_accessed(target_thread->pagedir,target_entry->vm_address)){
        pagedir_set_accessed(target_thread->pagedir,target_entry->vm_address,false);
      }
      else{
        return vm_swap_out_page(target_thread,target_entry);
      }
    }
    iter_elem = hash_next(&target_thread->vm_LRU_iterator);
    if(iter_elem == NULL){
      hash_first(&target_thread->vm_LRU_iterator,&target_thread->vm_list);
      iter_elem = hash_cur(&target_thread->vm_LRU_iterator);
      roll_index += 1;
    }
  }
  return false;
}
```

3. Swap Pool

The swap pool utilizes a similar implementation to palloc. It consists of a bitmap to track the in-use sectors on the swap disk, a lock to prevent concurrent access by multiple threads to the bitmap, and swap disk information. The swap pool is lazily initialized when a swap out is required.

To initialize the swap pool, the kernel retrieves the blocks registered as swap disk using the `block_get_role` function. It then determines the available number of sectors by dividing the usable disk size by the sector size. A new bitmap, equal to the number of available sectors, is allocated and stored in the swap pool.

When there is a need to swap out due to limited user space, the swap pool bitmap allocates a consequtive bitmap, size equal to the page size divided by the sector size. The page information is then stored in the corresponding sectors indicated by the allocated bitmap, and the block location where the page information is stored is saved in the `vm_entry`. The virtual page is then freed. When a swap in occurs, the page information is loaded into memory from the stored block location. Finally, the allocated bitmap is released.

- `vm/page.h`

```
struct swap_pool{
  struct lock * swap_lock;
  struct bitmap * used_map;
  struct block * target_block;
};
```

C. MMAP implementation

To implement mmap, we have created three page types: mmap_indisk, mmap_inmem, and mmap_swapout. We have also implemented the `sys_mmap` function. Additionally, we have introduced the `mmap_entry` structure to store information about the mapped file and the current thread.

When mmap is called, it searches for the corresponding file descriptor in the current thread's file descriptor list. Depending on the size of the found file, it creates `vm_entry` structures starting from the given virtual address. The necessary file information for swapping in the `vm_entry` is also stored within the `vm_entry` itself. Then, the required thread, file, and `vm_address` information are stored in the `mmap_entry` structure.

When a page fault occurs, the file information stored in the `vm_entry` is used to retrieve the data. For swap-out cases, the `pagedir_is_dirty` function is used to check if write operations have been applied to the page. If the page has been written to, the page information is saved in the file.

Mmap also requires handling of the `exit` and `close` system calls. When an `exit` system call occurs, the thread needs to save all memory mapped through mmap back to the file. To achieve this, a `mmap_list` is created to manage all `mmap_entry` structures. When `exit` is called, the vm pages created through mmap that are still in memory are swapped out to the file. Additionally, since mmap is lazily loaded, if the user does not access the mmap and the corresponding file is closed, the mmap information cannot be accessed by the user. Therefore, when the user closes the file, any mmap virtual pages associated with that file descriptor need to be swapped in.

- `userprog/syscall.c`

```
...
static struct lock mmap_lock;
static struct list mmap_list;

struct mmap_entry{
  int mapid;
  uint32_t vm_address;
  size_t mmap_size;
  struct list_elem mmap_elem;
  struct thread * mmap_thread;
  int fd;
};
...
int sys_mmap(int fd, void * addr){
  struct thread * current_thread = thread_current();
  struct list * fd_list = &current_thread->file_descriptors;
  if(fd <3){
    return -1;
  }
  if(!addr || addr >= PHYS_BASE){
    return -1;
  }
  if(!(addr == pg_round_down(addr))){
    return -1;
  }
  struct file_desc * target_file_desc = find_file_desc(current_thread,fd);
  if(!target_file_desc){
    return -1;
  }

  size_t size_of_file = file_length(target_file_desc->file);
  size_t left_size = size_of_file;
  size_t written_file_size = 0;
  if(size_of_file == 0){
    return -1;
  }

  void * target_addr = addr;
  while(left_size > 0){
    struct vm_entry * found_entry = find_vm_entry_from(current_thread,target_addr);
    if(found_entry || target_addr >= PHYS_BASE - MAX_STACK_SIZE){
      return -1;
    }
    struct vm_entry * target_entry = add_new_vm_entry_at(current_thread,PAGE_MMAP_INDISK,target_addr);
    size_t current_page_write_size = left_size > PGSIZE ? PGSIZE : left_size;
    target_entry->file_left_size = current_page_write_size;
    target_entry->swap_file = target_file_desc->file;
    target_entry->swap_file_offset = written_file_size;

    target_addr += PGSIZE;
    left_size -= current_page_write_size;
    written_file_size -= current_page_write_size;
  }
  struct mmap_entry * map_target_entry = malloc(sizeof(struct mmap_entry));
  map_target_entry->mapid = allocate_mapid_t();
  map_target_entry->mmap_size = size_of_file;
  map_target_entry->vm_address = addr;
  map_target_entry->mmap_thread = current_thread;
  map_target_entry->fd = fd;
  list_push_front(&mmap_list,&map_target_entry->mmap_elem);
  return map_target_entry->mapid;
}

void sys_munmap(int mapping){
  struct mmap_entry * found_mmap_entry = find_mmap_entry(mapping, thread_current());
  if(!found_mmap_entry){
    sys_exit(-1);
  }
  struct thread * current = thread_current();
  size_t mmap_size = found_mmap_entry->mmap_size;
  size_t left_size = mmap_size;
  void * target_addr = (void *)found_mmap_entry->vm_address;
  while(left_size > 0){
    struct vm_entry * found_vm_entry = find_vm_entry_from(current,target_addr);
    if(!found_vm_entry){
      printf("MUMMAP failed\n");
      sys_exit(-1);
    }
    if(is_entry_inmem(found_vm_entry)){
      vm_swap_out_page(current,found_vm_entry);
    }
    hash_delete(&current->vm_list,&found_vm_entry->vm_list_elem);
    vm_destroy(&found_vm_entry->vm_list_elem,NULL);
```

```
    hash_first(&current->vm_LRU_iterator,&current->vm_list);
    target_addr += PGSIZE;
    left_size -= left_size > PGSIZE ? PGSIZE : left_size;
  }
  list_remove(&found_mmap_entry->mmap_elem);
}
...
```

D. Accessing User Memory

When a page fault occurs, an `intr_frame` structure is passed as an argument. We can use this `intr_frame` structure to determine which memory the user was trying to access and where the stack pointer is pointing. However, when a page fault occurs in the kernel space, the `intr_frame` information of the kernel thread is provided, and we cannot access information such as user's stack pointer that is necessary for page fault recovery. Therefore, when accessing the user space from the kernel space, we need to preload the required user virtual pages. To achieve this, we have created the `vm_handle_syscall_handle` function in `page.c`, which checks if the user-specified memory access is valid and loads the pages if the access is valid, thus preventing page faults.

- `userprog/syscall.c`

```
static void
syscall_handler (struct intr_frame *f)
{
  int syscall_number;
  int intsize = 4;
  int ptrsize = 4;
  int fd;
  ASSERT( sizeof(syscall_number) == 4 ); // assuming x86

  // The system call number is in the 32-bit word at the caller's stack pointer.
  memread_user(f->esp, &syscall_number, intsize);
...

  // Dispatch w.r.t system call number
    case SYS_READ: // 8
    {
      uint32_t return_code;
      void *buffer;
      unsigned size;
      int sizek =sizeof(size);
      memread_user(f->esp + 4, &fd, intsize);
      memread_user(f->esp + 8, &buffer, ptrsize);
      memread_user(f->esp + 12, &size, sizek);
      bool handle_result = vm_handle_syscall_alloc(thread_current(),f,buffer,size);
      if(!handle_result){
        sys_exit(-1);
      }

      return_code = sys_read(fd, buffer, size);
      f->eax = return_code;
      break;
    }
...
```

- `vm/page.c`

```
/* above is region for swap handling*/
bool vm_handle_syscall_alloc(struct thread * target_thread, struct intr_frame *f, uint8_t* addr, uint32_t byte_to_handle){//TODO need
  vm_check_stack_vm_entry(target_thread,f,addr,byte_to_handle);
  uint8_t* target_addr = pg_round_down(addr);
  uint8_t* max_addr = pg_round_down(addr + byte_to_handle-1);
  while(target_addr <= max_addr){
    struct vm_entry * found_entry = find_vm_entry_from(target_thread,target_addr);
    if(found_entry == NULL){
      printf("vm_handle_stack_alloc_failed: reason - cannot find vm_entry from vm_list : addr - %x",addr);
      return false;
    }
    switch(found_entry->entry_status){
      case PAGE_STACK_UNINIT:{
        bool is_upper_stack  = (addr >= (f->esp - 32));
        if(is_upper_stack){
          vm_install_new_stack(target_thread,found_entry);
        }
        else{
          return false;
        }
        break;
```

```
      }
      case PAGE_STACK_SWAPPED:
        vm_swap_in_page(target_thread,found_entry);
        break;
      case PAGE_STACK_INMEM:
        break;
      case PAGE_FILE_INDISK:
        vm_swap_in_page(target_thread,found_entry);
        break;
      case PAGE_FILE_INMEM:
        //do nothing;
        break;
      default:
        return false;
        break;
    }
    target_addr += PGSIZE;
  }
  return true;
}
```

4.Results

```
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
pass tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
pass tests/vm/mmap-exit
FAIL tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
pass tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
pass tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
pass tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
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
3 of 109 tests failed.
```

Through problem analysis and code implementation, we have successfully passed all but few of the 109 tests provided by PintOS. Our code addresses the majority of the outlined problems, effectively implementing paging, stack growth, and mmap, as well as handling user memory access. In conclusion, we have achieved efficient memory utilization through VM implementation.