PintOS Part3 : Virtual Memory
Appendix

CONTENTS

Notice

- All implementations in this slides are examples only, and other approaches are possible

- Feel free to add any features or functionality you need for your implementation

Extra: export the container

- Export container as tar file
  - $ docker export -o [file name].tar [container name]

- Testing your file
  - $ docker import [file name].tar [image name]:[tag name]
  - $ docker run –it –name [container name] [image name]:[tag name] /bin/bash

1

Demand Paging with Swapping

Demand Paging with Swapping

- Current pintos
  - All files are loaded into memory at first startup → Inefficient because not everything in the file is necessary
  - All physical addresses are fixed on each page at the beginning of process execution → Problems such as memory overhead or swapout can occur
  - These problem can be solved by introducing a demand paging method.
- Need to implement
  - Demand paging
    - Virtual page: Virtual Page number (20 bit) + page offset (12bit)
    - Page frame: physical frame number (20 bit) + page offset (12 bit)
    - Page table:
      - VPN → PFN
      - It is hardware.
    - Swap space: array of page sized blocks

Demand Paging with Swapping

- **Page_fault**
  - page fault occurs when you try to access a virtual address because the address is not mapped to physical memory**.**

1. Obtain the starting address of the page:

- Get the starting address (**fault_page**) of the page by descending the virtual address(**fault_addr)** that caused the page fault to the page boundary.
  - Using **pg_round_down**() : Drops the given address to the nearest lower page boundary (already implemented)
  - For example, if **fault_addr** is **0x12345**, it is reduced to a multiple of the nearest page boundary(e.g. 4KB, 0x1000) to be 0x12000, which is used to find the starting address of a page when managing memory on a page-by-page basis.

2. Page fault cause analysis:

- The cause of a page fault can be determined by the **error_code** set in the interrupt frame :
  - **Not_present**: **true** if the page does not exist in physical memory.
  - **Write**: **true** if the fault occurred during a write operation, **false** if it occurred during a read operation.
  - **User**: **true** if the fault occurred in user mode, **false** if it occurred in kernel mode.

Demand Paging with Swapping

- **Page_fault**
  - page fault occurs when you try to access a virtual address because the address is not mapped to physical memory.

3. Load the page:

- **Call vm_load_page** function to load the required pages into physical memory. .

4. Exception handling:

- When the page is a fault (**if (!not_present)**) due to a write attempt in a read-only area or when the **vm_load_page** process fails, ,
  - If page fault in kernel mode, set the **eax** register to **0xffffffff**로 to notify the caller that a system call or other function has failed. Copy the previous **eax** value to **eip**so that the system call can return the error status when it returns to user mode.
  - If it occurred in user mode, the process is considered to be in an unrecoverable state and the **kill()** function is called. This safely terminates the process and releases system resources.

Demand Paging with Swapping

- **Bool vm_load_page(struct supplemental_page_table *supt, uint32_t *pagedir, void *upage)**
  - Load the page, specified by the address `upage`, back into the memory.

1. Check if the memory reference is valid

- Using **vm_supt_lookup(supt, upage)** : Looks up the supplemental page table entry (spte) corresponding to the virtual address upage. If the spte is NULL, it returns false because there is no information for that virtual address.

2. Obtain a frame to store the page

- Using **vm_frame_allocate(**PAL_USER, upage), : The virtual memory management system is responsible for allocating physical frames (pages in physical memory). This function allocates the requested physical pages and, if necessary, swaps out existing pages to free up space.

Demand Paging with Swapping

- **Bool vm_load_page(struct supplemental_page_table *supt, uint32_t *pagedir, void *upage)**

3. Fetch the data into the frame

- Status of page == ALL_ZERO:
  - Initialize frames to zero
- Status of page == ON_FRAME :
  - Nothing to do
- Status of page == ON_SWAP:
  - Using **vm_swap_in (spte->swap_index, frame_page)**, load the data from the swap disc
- Status of page == FROM_FILESYS :
  - Using **vm_load_page_from_filesys(spte, frame_page)**, reads a specific portion of a specific file from the file system and loads it into the specified kernel page.
  - If the load fails, Using vm_frame_free(frame_page), obtain frame_lock(you need to declare),free the page from the frame table that manages the kernel page, and unlock it

Demand Paging with Swapping

- **Bool vm_load_page(struct supplemental_page_table *supt, uint32_t *pagedir, void *upage)**

4. Point the page table entry for the faulting virtual address to the physical page

- **Using pagedir_set_page (pagedir, upage, frame_page, writable),**in the page directory (pagedir) of a specific process, associates a specific virtual page address (upage) with a corresponding physical memory address (frame_page). This process links virtual addresses to real physical memory addresses so that the process can access data.
- If the mapping fails, **Using vm_frame_free(frame_page),** free the page from the frame table that manages kernel pages

5. Update the supplemental page table entry(spte), indicating that the page has been loaded into the frame

6. Using pagedir_set_dirty (pagedir, frame_page, false) (already implemented), **Set the dirty bit to DIRTY in the page table entry for virtual page virtual page in the page directory**

- 'dirty bit' indicates that the page has been modified, used by the system to determine if the page needs to reflect its latest state when it is swapped out or written to disk.

7. Using vm_frame_set_pinned (frame_page, false),: unpin the given kernel page..

Demand Paging with Swapping

- **You need to modify thread structure to dealing virtual memory space.**

```
struct thread{
…
#ifdef VM
    struct supplemental_page_table *supt;
#endif
…
}
```

supt

- supplemental_page_table is used to manage the virtual address space of the thread. This table contains mapping information of the virtual address and its corresponding physical page.
- It is necessary for each thread to manage its virtual address space and to independently maintain mapping information from virtual address to physical address.

Demand Paging with Swapping

- **struct supplemental_page_table_entry* vm_supt_lookup (struct supplemental_page_table *supt, void *page)**
  - This function looks up information about a specific virtual address within the process's virtual memory system.
  - It looks for an entry (**supplemental_page_table_entry**) in the process's supplemental page table (**supplemental_page_table**) that contains information associated with that virtual address.

Demand Paging with Swapping

- **struct supplemental_page_table :**
  - Contains the hash table **page_map** for managing the virtual address of the process.
  - This hash table maps each virtual page address to **supplemental_page_table_entry** to manage the status and location information of that page.

- **struct supplemental_page_table_entry :**
  - Store information about individual virtual pages, including the following variables.
    - **void *upage;**: The virtual address of the page.
    - **void *kpage;**: Indicates the physical address of the page if the page is loaded into physical memory; **NULL** if the page is not in memory.
    - **struct hash_elem elem;**: A hash element for inserting this structure into the hash table.
    - **enum page_status status;**: Ithe current status of the page, ex) **ALL_ZERO**, **ON_FRAME**, **ON_SWAP**, **FROM_FILESYS**
    - **swap_index_t swap_index;**: The index that points to the swap location when the page is in the swap zone.
    - **bool dirty;**: A dirty bit that indicates whether the page has been modified.
    - **struct file *file;**: Indicates the file if the page is mapped in the file system..
    - **off_t file_offset;**: The offset from which the page starts within the file..
    - **uint32_t read_bytes, zero_bytes;**: The number of bytes to read from the file and the number of bytes to fill the rest of the page with zeros.
    - **bool writable;**: Whether the page is writable.

Demand Paging with Swapping

void*vm_frame_allocate (enum palloc_flags flags, void *upage)

- This function allocates physical frames (physical memory pages) from the virtual memory management system. Assign the requested physical pages and, if necessary, swap out the existing pages to free up space.

1. lock aquire

- Acquire **frame_lock(you need to declare)** locks to manage the allocation and release of memory frames..

2. Attempt to allocate page frames

- **Use thepalloc_get_page(PAL_USER | flags)** function to allocate physical pages. If page allocation fails, there is not enough memory, so space is reserved through the swapout process.

3. swapout

- **pick_frame_to_evict:** Select the frame to swap out through the function
- **Using pagedir_clear_page** function to verify that the page has been modified, and update the dirty status
- **Using pagedir_isdirty**() function to verify that the page has been modified, and update the dirty status.
- **Using vm_swap_out** to swap out the frame and update the swap index.
- **Using vm_supt_Set_swap,** after the page is swapped out, reflects its status in the secondary page table.
- **Using vm_supt_set_dirty,** updates the status of 'dirty' for a particular page in the supplemental page table to indicate if the page has changed.
- **Using vm_frame_do_free** to physically release the frame.

Demand Paging with Swapping

void*vm_frame_allocate (enum palloc_flags flags, void *upage)

4. Call palloc_get_page

- Call the **paolloc_get_page** (PAL_USER | flags) again to get the page frame assigned.

  .

5. Creating a Frame Table Entry

- **Using malloc** create and initialize the **frame_table_entry**. This structure stores the metadata of the allocated frames.
- Set a pin on the new frame to prevent swapping out.

6. Set a pin on the new frame to prevent swapping out.

- **Use hash_insert to add a new entry to the frame map.**

7. Unlock and return frame address

- After completing all tasks, unlock frame_lock and return the address of the assigned physical page.

Demand Paging with Swapping

struct frame_table_entry

- Used to store information for each allocated frame.
- This structure contains the following fields:
  - **void *kpage;**: The physical address of the frame.
  - **struct hash_elem helem;**: A hash element used when included in the frame map hash table.
  - **struct list_elem lelem;**: List element used when included in a frame list.
  - **void *upage;**: The virtual address mapped to this frame.
  - **struct thread *t;**: The pointer to the thread that owns this frame..
  - **bool pinned;**: frame is pinned, this means that it will not be swapped out. Used to prevent swapping while securing or releasing resources.

Demand Paging with Swapping

- **Need to implement**

```
void vm_swap_init () {
  ASSERT (SECTORS_PER_PAGE > 0);
  swap_block = block_get_role(BLOCK_SWAP);
  if(swap_block == NULL) {
      PANIC ("Error: Can't initialize swap block");
      NOT_REACHED ();
  }
  swap_size = block_size(swap_block)
  swap_available = bitmap_create(swap_size);
  bitmap_set_all(swap_available, true);
}
```

- This function is part of the virtual memory management system, which is called once when the operating system boots and is used to initialize the swap zone.
- Establish a basic structure for setting up and managing swap space on your system..

Demand Paging with Swapping

- **Need to implement**

```
void vm_frame_init ()
{
  lock_init (&frame_lock);
  hash_init (&frame_map, frame_hash_func, frame_less_func, NULL);
  list_init (&frame_list);
  clock_ptr = NULL;
}
```

- Perform initialization operations to manage physical memory frames within the operating system's virtual memory management system.
- It runs once when the system boots, setting up essential data structures for frame management.
- Initialize the pointer **clock_ptr** used for the frame table, **frame_lock**, **frame_map, frame_list,** and clock algorithms, which are locks used to prevent simultaneous access with other threads when accessing and modifying the frame table.

Demand Paging with Swapping

struct frame_table_entry* pick_frame_to_evict(uint32_t *pagedir)

- When there is not enough physical memory available, select a frame (page) from memory to replace
- This function uses the "clock algorithm" to determine the best replacement candidate.

1.Check frame map size:

- Check the total number of allocated frames. If the frame table is empty, output an error message and enter a panic state (**PANIC**).

2.Traverse frames to find evict :

- Traverses the list of frames, checking the selection criteria of the frames..
- If a frame is pinned, it is skipped. Pinned frames cannot be swapped out..
- Check to see if the page for that frame has been accessed recently. If it has been accessed, clear the access flag and move on to the next iteration.
- If you find a frame that is unflagged and not pinned, select it as a evict to swap out..

3. **Return a pointer to the frame to be swapped out.**

- If it finds no evict because all frames are accessed or pinned, it raises a panic condition indicating that it is out of memory and exits.

Demand Paging with Swapping

swap_index_t vm_swap_out (void *page)

- Writes the given page to a swap area and returns the swap index of that page. .
- Saves a page from physical memory to a swap area on disk,and returns the index of that swap area.
- This process is used when moving pages that are not currently in use to disk for memory management.

1. Validate the page

- **Using bitmap_scan (swap_availableASSERT (page >= PHYS_BASE)** verify that the address (page) of the page you want to swap out belongs to the user's virtual memory region.

2. Find an available swap slot

- **Using bitmap_scan (swap_available, 0, 1, true)** find one free slot in the bitmap that manages the available swap area (**swap_available**) and return its index. In other words, find the index of an available swap slot.

3. Write the page data to the swap area

- Repeat as many times as the number of disk sectors that make up a page **(SECTORS_PER_PAGE)** and store the entire contents of the page in successive sectors on the disk.

- **Using the block_write(swap_block, swap_index * SECTORS_PER_PAGE + i, page + (BLOCK_SECTOR_SIZE * i))** function, write the data of the sector from memory **(page + (BLOCK_SECTOR_SIZE * i))** to the calculated disk sector position **(swap_index * SECTORS_PER_PAGE + i).**

Demand Paging with Swapping

bool vm_supt_set_swap (struct supplemental_page_table *supt, void *page, swap_index_t swap_index)

- When a page corresponding to a specific virtual address is moved to the swap zone, reflect the status in the supplemental page table.

1. **Update page status**:
   - Using **vm_supt_lookup()** function, find the page table entry (**spte**) corresponding to the given virtual address in the supplemental page table.
   - It sets the status of the found entry to ON_SWAP to indicate that the page has been moved from physical memory to the swap area.

2. **Set the swap index**:
   - Stores a **swap_index** in the **spte** that indicates the location of the swapped-out page. This index is needed when the page is later reloaded.
3. **Initialize physical page information**:
   - **spte->kpage** to **NULL**, making it clear that this page does not currently exist in physical memory..
   - **spte->kpage** field serves to store the physical memory address of this page within the supplemental page table entry **(struct supplemental_page_table_entry)**.

4. It returns **true** if the page table entry was successfully updated, or **false** if the page was not found..

Demand Paging with Swapping

bool vm_supt_set_dirty (struct supplemental_page_table *supt, void *page, bool value)

- Updates the 'dirty' status for a specific page in the given supplemental page table.

1. Looking up page entries:

- **Using vm_supt_lookup** look up the page table entry (**spte**) corresponding to a given virtual address.

2. Update Dirty Status:

- Updates the 'dirty' flag of the looked up page entry with the given value. If value is **true** and the 'dirty' flag was already set, the flag remains **true**. If **value** is **false**, the original 'dirty' status remains unchanged if it was **true**.
  - Value is the value to set the page's 'dirty' status to: **true** indicates that the page has been modified, **false** indicates that it has not been modified.

Demand Paging with Swapping

void vm_frame_do_free (void *kpage, bool free_page)

- This function free a page allocated in kernel space and cleaning up the associated resources.
- Removes entries from the frame table, frees pages of physical memory if necessary, and frees the memory of the frame table entries themselves.

1.Validation:
- To ensure data consistency and system stability, it validates that the **frame_lock** is held by the current thread, that the entered **kpage** address belongs to the kernel address space, and that the page addresses are correctly aligned.

2.Hash table lookup:
- Given a kernel page address, locates the corresponding frame in the frame table (**frame_map**).
- If the corresponding frame is not found, the **PANIC** function is called to signal the error and stop the system.

3.Remove the frame table entry:
- After obtaining the actual frame table entry from the hash element looked up, we remove it from the hash table and list.
- This means that the page is no longer managed through the frame table.

4.Freeing resources:
- If the parameter **free_page** is true, the **palloc_free_page** function is called to free a page allocated in physical memory.
  - This allows the physical memory to be reused for other purposes.
- Frees the memory of the allocated frame table entry itself.

Demand Paging with Swapping

void vm_swap_in (swap_index_t swap_index, void *page)

- Performs an operation to reload a page from the swap area into memory.

1.Validation:

- Validates that the starting address of the memory to load data from (**page**) is within the user virtual memory area.
- Verify that the index **(swap_index)** indicating where the page data is stored in the swap area is within the swap area.
- Using **bitmap_test(swap_available, swap_index),** verify that the swap slot has data allocated for it. A read attempt to a slot that does not have data allocated is considered an error.

2.Load data:

- **Using block_read(swap_block,swap_index * SECTORS_PER_PAGE + i, page + (BLOCK_SECTOR_SIZE * i)** read data from the swap area as many times as the page size, starting with the sector corresponding to **swap_index**.
- The data from each sector is stored sequentially at the **page** address using the calculated offset.

3.Updating the swap slots:

- After loading data using **bitmap_set(swap_available, swap_index, true)** set the usage status of the corresponding swap slot back to 'available'.

Demand Paging with Swapping

static bool vm_load_page_from_filesys(struct supplemental_page_table_entry *spte, void *kpage)

- Reads a specific portion of a specific file from the filesystem and loads it into the specified kernel page (**kpage**).

1.Set the file location:

- **Using file_seek (spte->file, spte->file_offset)** set the exact location to start the read operation by moving the file pointer provided by **spte** to a specific offset in the file (**spte->file_offset**).

2.Reading data from a file:

•**Using file_read (spte->file, kpage, spte->read_bytes)** load the requested number of bytes (**spte->read_bytes)** of data from the file into the kernel page (**kpage**).

•If the number of bytes actually read from the file does not match the number of bytes requested, the function returns **false**, signaling a load failure.

3.Fill the rest of the page::

•**Using memset (kpage + n_read, 0, spte->zero_bytes)** fill the remaining portion of the kernel page (**kpage**) with zeroes.

4. Return true if the load process completes successfully.

Demand Paging with Swapping

static void vm_frame_set_pinned (void *kpage, bool new_value)

- This function is used by the page replacement algorithm to protect a page from being replaced (pin) or make it available for replacement (unpin)

1. Access control

- Prevents other threads from accessing the frame table at the same time by acquiring a **frame_lock.**

2. Hash table lookup

- Using **kpage ,** locates the entry for that frame within the **frame_map** hash table. .
- This process looks up the hash table based on the frame's memory address to identify the entry that contains the metadata for that frame.

3. Change the pin state of the frame table entry

- After converting the looked-up hash element into a **struct frame_table_entry** structure, change the pin state by setting the pinned field of this structure to **new_value**.
- If **new_value == true** means pin, **new_value == false** means unpin.

4. Lock release

- Release the lock to allow other threads to access the frame table.

Demand Paging with Swapping

- **Memory mapping**
    - Previously, when a system call is made, it typically accesses physical memory directly to process data. But after implement virtual memory, system calls also need to access and manage data through a virtual address space.
    - So, modifications are required, such as mapping system calls, such as file system access, to virtual memory using **mmap** and unmapping them with **munmap** as needed.
- **mmap (Memory Mapping)**
    - mmap is used when you need to map data from a file or device to the virtual address space of a process.
        - Used when you need to map data from a file or device to the virtual address space of a process..
        - Mapped memory regions can be read and written like normal memory accesses, and the system automatically loads only the necessary portions into physical memory..
- **munmap**
    - Required when a program no longer uses the mapped memory, or to clean up resources before the program exits.
        - Unmaps memory in the specified virtual address space, clearing the associated page table entries and, if necessary, rewriting the changed data to disk.
- **Association with Demand Paging**
    - Memory mappings created through **mmap** utilize the demand paging mechanism, and do not load pages into physical memory until they are actually accessed.
    - **munmap** is used to unmap these, swapping out changed pages and reclaiming physical memory if necessary

Demand Paging with Swapping

- **You need to modify thread structure to dealing virtual memory space.**

```
struct thread{
…
#ifdef VM
    ….
    struct list mmap_list;              /* List of struct mmap_desc. */
#endif
…
}
```

mmap_list

- Manage a list of files mapped to memory by these threads, each of which contains an instance of the **structure mmap_desc** structure, each of which represents an individual file mapping

Demand Paging with Swapping

- **You need to add thread structure to dealing virtual memory space.**

```
typedef int mmapid_t;
struct mmap_desc {
  mmapid_t id; // Each mapping has a unique ID, which allows you to reference or manipulate specific mappings.
  struct list_elem elem; // mmap_desc are connected to a list to efficiently traverse and access multiple mappings..
  struct file* file; // refers to a mapped file.
  void *addr;   // store the user virtual address.
  size_t size;  //  total number of bytes of files mapped to memory
};
```

- This structure stores the details of memory mapped files.
- This structure is primarily used to map a file or device's data to memory through **mmap** system calls.
- When you call the **munmap** system, this structure is used to remove the file mapping from memory and manages the cleanup and release of related resources.

Demand Paging with Swapping

- **Helper functions on memory access**
  - You can use the following helper functions to modify the system call function
    - static int32_t get_user (const uint8_t *uaddr)
    - static bool put_user (uint8_t *udst, uint8_t byte)
    - Static void check_user (const uint8_t *uaddr)
    - static void fail_invalid_access(void)
    - static int memread_user (void *src, void *dst, size_t bytes)
    - static struct mmap_desc* find_mmap_desc(struct thread *t, mmapid_t mid)
    - void preload_and_pin_pages(const void *buffer, size_t size)
    - void unpin_preloaded_pages(const void *buffer, size_t size)

Demand Paging with Swapping

```
static int32_t get_user (const uint8_t *uaddr)
{
if (! ((void*)uaddr < PHYS_BASE)) {
    return -1;
  }
int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
      : "=&a" (result) : "m" (*uaddr));
  return result;
}
```

- This function checks if the user address is within a valid range, and only reads a single byte from the given user memory address(**uaddr**) if it is valid.
- ' uaddr ' must be below PHYS_BASE.
- Returns the byte value if successful (extract the least significant byte), or -1 in case of error (a segfault occurred or invalid uaddr)

Demand Paging with Swapping

```
static bool put_user (uint8_t *udst, uint8_t byte)
{
    if (! ((void*)udst < PHYS_BASE)) {
        return false;
    }
    int error_code;
    asm ("movl $1f, %0; movb %b2, %1; 1:“ : "=&a" (error_code), "=m" (*udst) : "q" (byte));
    return error_code != -1;
}
```

- This function performs checks to ensure safe memory access and prevent write attempts to invalid memory addresses.
- It stores the given bytes(**byte**) at the specified user address (**udst**).
- '**udst**' must be below PHYS_BASE.
- Returns true if successful, false if a segfault occurred.

Demand Paging with Swapping

```
static void check_user (const uint8_t *uaddr)
{
  if (get_user (uaddr) == -1)
      fail_invalid_access();
}
```

- This function is mainly used to verify the validity of user memory access in the system
- Checks if the given user memory address (**uaddr**) is valid, and if an invalid memory access is detected, calls the **fail_invalid_access** function to perform error handling

Demand Paging with Swapping

```
bool lock_held_by_current_thread (const struct lock *lock)
{
  ASSERT (lock != NULL);
  return lock->holder == thread_current ();
}
```

- Returns true if the current thread holds LOCK, false otherwise.

```
static void fail_invalid_access(void) {
  if (lock_held_by_current_thread(&filesys_lock))
      lock_release (&filesys_lock);
  sys_exit (-1);
  NOT_REACHED(); }
```

- Called when an invalid memory access is detected, it is responsible for safely shutting down the system.
- This function is mainly used to immediately abort the user program when an error occurs during memory access validation, and to free up necessary resources.

Demand Paging with Swapping

```
static int memread_user (void *src, void *dst, size_t bytes)
{
  int32_t value;
  size_t i;
  for(i=0; i<bytes; i++) {
      value = get_user(src + i);
      if(value == -1) // segfault or invalid memory access
          fail_invalid_access();
      *(char*)(dst + i) = value & 0xff;
  }
  return (int)bytes;
}
```

- This function is mainly used by the os or system calls to securely read data from user space
- Reads a consecutive \`bytes\` bytes of user memory with the starting address \`src\` (uaddr), and writes to dst. And returns the number of bytes read.
- In case of invalid memory access, exit() is called and consequently the process is terminated with return code -1.

Demand Paging with Swapping

static struct mmap_desc* find_mmap_desc(struct thread *t, mmapid_t mid)

```
{
  ASSERT (t != NULL);
  struct list_elem *e;
  if (! list_empty(&t->mmap_list)) {
        for(e = list_begin(&t->mmap_list); e != list_end(&t->mmap_list); e = list_next(e))
        {
              struct mmap_desc *desc = list_entry(e, struct mmap_desc, elem);
              if(desc->id == mid) {
                    return desc;
               }
         }
    }
  return NULL;
}
```

- Finds the memory mapping descriptor (**mmap_desc**) corresponding to a specific ID (**mid**) in the memory mapping list of a given thread.
    1. Using a pointer that acts as an iterator to traverse the thread's memory mapping list, traversing from the beginning to the end of the memory mapping list.
    2. It checks if the ID of the current memory mapping descriptor matches the **mid** it is looking for.
    3. If a matching memory mapping descriptor is found, it returns a pointer to that descriptor.
    4. If it has not found a matching descriptor, it returns NULL to indicate that it has not found one.

Demand Paging with Swapping

```
void preload_and_pin_pages(const void *buffer, size_t size)
{
  struct supplemental_page_table *supt = thread_current()->supt;
  uint32_t *pagedir = thread_current()->pagedir;
  void *upage;
  for(upage = pg_round_down(buffer); upage < buffer + size; upage += PGSIZE)
  {
     vm_load_page (supt, pagedir, upage);
     vm_pin_page (supt, upage);
  }
}
```

- Preloads the contents of the given buffer into virtual memory and pins the page.
- This function is often called before an I/O operation to ensure that the required memory pages are not swapped out.

Demand Paging with Swapping

```
void unpin_preloaded_pages(const void *buffer, size_t size)
{
  struct supplemental_page_table *supt = thread_current()->supt;
  void *upage;
  for(upage = pg_round_down(buffer); upage < buffer + size; upage += PGSIZE){
       vm_unpin_page (supt, upage);
  }
}
```

- This function is primarily used to unpin memory pages after an I/O operation.
- The primary purpose of this function is to optimize memory management. By unpinning pages, the system is free to swap out those pages as needed.
- pg_round_down(buffer): This function rounds the start address of the given buffer down to the nearest page boundary, obtaining a page-aligned start address. This is used for the purpose of starting an operation at a page-aligned address.

Demand Paging with Swapping

```
static void syscall_handler (struct intr_frame *f) {
        ...
 #ifdef VM
     case SYS_MMAP:
     {
         int fd;
         void *addr;
         memread_user(f->esp + 4, &fd, sizeof(fd));
         memread_user(f->esp + 8, &addr, sizeof(addr));
         mmapid_t ret = sys_mmap (fd, addr);
         f->eax = ret;
         break;
     }
 ...
 }
```

- The system call maps file data into memory at the specified address (**addr**) using a specific file descriptor (**fd**), and returns a mapping ID if the mapping was successful.
- This ID is used later when referencing or releasing the mapping.

Demand Paging with Swapping

mmapid_t sys_mmap(int fd, void *upage)

- Load file data into memory by demand paging.
- mmap()'ed page is swapped out to its original location in the file.
- **fd** is the file descriptor of the file to be mapped, **upage** is the start address of the virtual address to which the file contents will be mapped.

1. **Argument checking:**
   - Returns an error (-1) if **upage** (user page start address) is NULL, or if the page offset is non-zero. This means that the mapping start address must be aligned with the page boundary.
2. **Open file**:
   - Use the **find_file_desc** function to find the file descriptor corresponding to **fd** in the current thread.
   - If that file descriptor is valid, reopen the file with **file_reopen** function. In this way, you get an independent file handle without affecting the original file.
3. **Memory page mapping:**
   - Use the **vm_supt_lookup** function to check if the page address is already in use. If it is already in use, code moves to error handling to avoid conflicts.
   - Call the **vm_supt_install_filesys** function to install the page into the virtual memory structure. This function installs a filesystem-based page, and sets it as readable.

Demand Paging with Swapping

mmapid_t sys_mmap(int fd, void *upage)

- Load file data into memory by demand paging.
- mmap()'ed page is swapped out to its original location in the file.
- **fd** is the file descriptor of the file to be mapped, **upage** is the start address of the virtual address to which the file contents will be mapped.

4. **Assign mapping ID**:
   - Finds the ID of the last mapping in the current thread's mapping list (**mmap_list**) and sets the new mapping ID. If the list is empty, start the ID with 1.
   - Allocate a new **mmap_desc** structure, initialize it, and add it to the mapping list.
5. **Return**
   - If all goes well, release the file system lock and return the newly assigned mapping ID.
   - If an error occurred, release the file system lock, and return -1.

- Fails if
  - File size is 0.
  - Addr is not page aligned.
  - Address is already in use.
  - Addr is 0.
  - STDIN and STDOUT are not mappable..

Demand Paging with Swapping

bool vm_supt_install_filesys(struct supplemental_page_table *supt, void *upage, struct file * file, off_t offset, uint32_t read_bytes, uint32_t zero_bytes, bool writable)

- It maps a portion of a specific file to a process's virtual address space, which serves to link some data from a file stored on the system to an area of memory that the process can access directly.

1.Creating a supoplemental page table entry:

- Use **malloc** to allocate memory for a new **supplemental_page_table_entry(spte)** structure.
- Store information about file mappings in the allocated spte structure.
  - The initial state of the page is not yet dirty (dirty=false), indicating that it is being loaded from the file system.

2.Inserting entry into the hash table:

- Inserts the supplemental page table entry you created into the process's supplemental page table hashmap.
- Use the **hash_insert** function to insert the newly created spte into the hashmap (page_map) of the process' supt.
  - If the hash table does not already have an entry for the same address, it returns true to indicate a successful mapping.
  - If there is already an entry for the same address in the hash table, it means that a duplicate entry exists, in which case the program will immediately abort with an error message via the **PANIC** macro.

Demand Paging with Swapping

static void syscall_handler (struct intr_frame *f) {

```
...
#ifdef VM
...
           case SYS_MUNMAP:
             {
               mmapid_t mid;
               memread_user(f->esp + 4, &mid, sizeof(mid));
               sys_munmap(mid);
               break;
             }
         #endif
         ...
 }
```

- SYS_MUNMAP system call takes a mapping identifier from the process, finds the corresponding memory mapping, frees it, and returns the used resources.
- Examples of when this system call is called :
    - When a program no longer needs a memory mapping created by memory mapping
    - At program termination, all memory mappings that were open before termination are freed to ensure that the system does not have memory leaks.
    - When a program needs to change its memory usage structure while running, releasing existing memory mappings and setting up a new mapping structure.
    - When a problem occurs after an **mmap** call or an unexpected error prevents memory mapping from occurring normally

Demand Paging with Swapping

bool sys_munmap(mmapid_t mid)

- This function takes a **mid** (memory mapping ID) as an argument, and finds and frees the memory mapping corresponding to that ID.

1. **Find memory mapping**
   - Finds the memory mapping structure (mmap_desc) corresponding to the given **mid** in the list of memory mappings of the currently running thread.
   - If not found, the function returns false and exits.
2. **Memory unmapping**
   - Acquire **filesys_lock** to synchronize file system access..
   - Depending on the size of the mapped file, it iterates in pages, computes a virtual address for each page, and processes it for the size of the page or the remainder of the file.
     - In the process, utilizes the **vm_supt_mm_unmap(curr->supt, curr->pagedir, addr, mmap_d->file, offset, bytes)** function, releasing the page from the supplemental page table and, if necessary, reflecting the changed page contents in the file.
   - Removes the memory mapping structure from the mapping list, closes the associated file, and frees the memory allocated for the memory mapping structure.
   - Release **filesys_lock**
3. **Return**

- Returns **true** if all the freeing processes have completed successfully, indicating that the memory unmapping was successful.

Demand Paging with Swapping

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

- Unmaps a specific page from the virtual address space and cleaning up its associated resources.
- This function uses the process's supplemental page table (**supplemental_page_table**) to manage the status and information about a specific **page**, and releases it from physical storage or swap storage.

1. **Lookup page entry**
    - Looks up the page table entry corresponding to a given virtual address (**page**) in the supplemental page table.
    - If the lookup result is **NULL**, this indicates a program error, and the system goes into a **PANIC** state. This handles cases where the page should exist in the supplemental page table but is missing.

Demand Paging with Swapping

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

- Unmaps a specific page from the virtual address space and cleaning up its associated resources.
- This function uses the process's supplemental page table (**supplemental_page_table**) to manage the status and information about a specific **page**, and releases it from physical storage or swap storage.

2. **Handling based on page status**
   - Depending on the status of the page (**spte->status**), handle it as follows :
     - **ON_FRAME**: The page is loaded into a memory frame.
       - Pin that frame to prevent page faults from occurring.
       - This is important when safely swapping out data or writing to a file.
       - Check the dirty bit of the page to determine if the page has been modified, and if so, write the modified content to the file.
       - After that, it unframes it and removes its page mapping from the page directory.
     - **ON_SWAP**: The page is in the swap zone.
       - If the page has changed by checking the dirty bits of the page, load the page from the swap and write the changes to the file.
       - If it hasn't changed, just release the swap area.
     - **FROM_FILESYS**: For pages mapped directly from the file system, there is no need to do anything special.
3. **Delete page table entry**
   - Delete the corresponding page entry from the supplemental page table.
   - Returns true if all operations complete successfully.

Demand Paging with Swapping

bool vm_supt_install_frame(struct supplemental_page_table *supt, void *upage, void *kpage)

- Adds a new page table entry to the process's supplemental page table (supplemental_page_table).
- This function assists with virtual memory management by registering the physical memory page (**kpage**) associated with a specific user virtual address (**upage**)

1. **Create and initialize the entry**
   - Allocate memory for a new supplemental_page_table_entry structure using **malloc**
   - Set the allocated spte structure with **upage**, **kpage**, and set status to **ON_FRAME**, indicating that this page is loaded into a memory frame.
   - Set the dirty flag to **false**, and initialize **swap_index** to **-1**, indicating that it is not currently swapped out.
2. **Insert an entry into the hash table**
   - Use the **hash_insert** function to insert the supplemental page table entry you created into the hashmap (page_map) of the process's supplemental page table.
   - If the entry is successfully inserted into the hash table, return true to indicate success.
   - If an entry already exists for that **upage**, free the newly allocated spte memory (**free(SPTE)**) and return false to indicate failure.
     - This is because each memory address must uniquely reference data (page, data in a file, etc.), and duplicate addresses can cause confusion in data management or unexpected errors.

Demand Paging with Swapping

- Code that performs tasks related to virtual memory management needs to conditionally **insert virtual memory specific logic**.

- Example1

```
int sys_read(int fd, void *buffer, unsigned size) {
     if(file_d && file_d->file) {
          #ifdef VM
               preload_and_pin_pages(buffer, size);
          #endif
               ret = file_read(file_d->file, buffer, size);
          #ifdef VM
               unpin_preloaded_pages(buffer, size);
          #endif
     }
}
```

Demand Paging with Swapping

- Code that performs tasks related to virtual memory management needs to conditionally **insert virtual memory specific logic**.

- Example2

```
int sys_write(int fd, const void *buffer, unsigned size) {
    #ifdef VM
        preload_and_pin_pages(buffer, size);
    #endif
        ret = file_write(file_d->file, buffer, size);
    #ifdef VM
        unpin_preloaded_pages(buffer, size);
    #endif
}
```

Demand Paging with Swapping

- Code that performs tasks related to virtual memory management needs to conditionally **insert virtual memory specific logic**.

- Example3

```
static void init_thread (struct thread *t, const char *name, int priority)
{
…
#ifdef VM
  list_init(&t->mmap_list);
#endif
…
}
```

- If necessary, you should also conditionally insert virtual memory related logic like this in other code except for the example

2

Stack Growth

Stack Growth

- Until Part 2, the stack we used was a single page, starting with USER_STACK, and the execution of the program was limited to this size.
- Implementing Stack Growth allocates additional pages when the stack exceeds its current size.
- If a page fault occurs because there is no frame mapped to the accessed virtual address, and the accessed virtual address exists within the stack region, the page fault can be resolved by allocating additional pages.

Stack Growth

- **Implementing stack growth within the page fault function**

1. **Obtaining the stack pointer**
    - Determine the appropriate stack pointer value (**esp**) based on whether the context in which the current page fault occurred is user mode or kernel mode.
        - If the page fault occurred in user mode, take the **esp** value from the interrupt frame; if it occurred in kernel mode, take the thread’s **current_esp** value.
        - The esp value in the interrupt frame is a store of the **esp** value when the interrupt (e.g., system call, page fault) occurred in user mode. This value reflects the state of the user stack at the time of the page fault, so you can use it to determine the current stack position.
        - When switching to kernel mode due to a system call or other kernel service call, the value of the user mode esp is saved in **current_esp**. This value provides the necessary user stack information in kernel mode.

SEOULTECH

Stack Growth

- **Implementing stack growth within the page fault function**

2. **Checks for stack growth conditions**
   1. **Accessed in stack frame or not**
      - Checks whether the virtual address accessed when a page fault occurs (**fault_addr**) is an address higher than the current stack pointer (**esp**).
        - The stack grows from high to low addresses, so an access that occurs at an address higher than the stack pointer can be considered a valid stack expansion.
      - Examine the location of page faults expected by the **PUSH** instruction.
        - The PUSH instruction stores data at the current location of the stack pointer and then decrements the stack pointer by the size of the data (4 bytes).
        - When performing a PUSH command, handle the case where a page fault occurs at the location where you want to store data before the stack pointer has yet to be updated.
      - Examine the location of the page fault expected by the **PUSHA** instruction.
        - The PUSHA command pushes 32 bytes onto the stack at a time.
   2. **Whether the stack address is in range**
      - Checks if **fault_addr** is within the maximum allowed range of the stack (**between PHYS_BASE - MAX_STACK_SIZE and PHYS_BASE**)
3. **Execute stack expansion**
   – If both conditions 2.1 and 2.2 above are true, it is considered a valid request that requires stack growth, and the processing to expand the stack is performed.
   – If there is no supplemental page table entry for that **fault_page**, a new zero page (a page whose contents are initialized to zero) is allocated and added to the supplemental page table using **vm_supt_install_zeropage(curr->supt, fault_page)**

Stack Growth

Bool vm_supt_install_zeropage (struct supplemental_page_table *supt, void *upage)

- Install an initialized zero-page entry corresponding to a specific virtual page address (**upage**) in the supplemental page table for virtual memory management of process.
- You can use zero-pages during page fault handling or initial configuration of the virtual address space to minimize memory usage until memory allocation and data writes are made based on actual need.

1. **Create and initialize the entry**
   - Allocate a new instance of the supplemental_page_table_entry structure into memory (using **malloc**).
   - Initialize each field of the supte:
     - upage: assign the virtual page address passed to the function
     - kpage: Set to **NULL** because this page has not yet been mapped to real physical memory
     - status: Set to **ALL_ZERO** to indicate that this is a zero page that contains no data yet
     - dirty: Set to **FALSE** to indicate that the page has not been modified.
2. **Insert entry into the secondary page table**
   - Using **hash_insert**, insert the generated **spte** into the hash table (**page_map**) of supplemental page table.
     - If the insertion was successful, return **true** and exit the function.
     - If the insertion failed because an entry already exists at that location, call the **PANIC** function to notify the error and stop the system

Reference

- **Stanford pintos project**
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos.html#SEC_Contents
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC39 : 3.1.5 accessing user memory
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos_4.html#SEC68 : 4.3.4 memory mapped files

Thank you