OS lab3 report - Team 16

1. Page Table Management

1.1. Data Structures

list & state purpose for your new or modified item

Ans:

- page.h

```
struct page_info
{
 uint8_t type;
 uint8_t writable;
 uint32_t *pd;
 const void *upage;
 bool swapped;
 struct frame *frame;
 union
 {
   struct file_info file_info;
   block_sector_t swap_sector;
   const void *kpage;
 } data;
 struct list_elem elem;
};
```

- Fields:
  - type: Indicates the type of the page (e.g., zero, kernel, file-backed).
  - writable: Specifies if the page is writable and where it can be written (file or swap).
  - pd: The page directory mapping the page.
  - upage: The user virtual address of the page.
  - swapped: Indicates if the page is swapped out.
  - frame: Pointer to the frame backing the page.
  - data: Union containing file info, swap sector, or kernel page address.
  - elem: List element for managing the page in the frame's page list.

- frame.h

```
struct frame
{
 void *kpage;
 struct list page_info_list;
 unsigned short lock;
 bool io;
 struct condition io_done;
 struct hash_elem hash_elem;
 struct list_elem list_elem;
};
```

- Fields:
  - kpage: Kernel virtual address for the frame.
  - page_info_list: List of page_info structures sharing this frame.
  - lock: Lock count to prevent eviction.
  - io: Indicates if I/O is in progress for the frame.
  - io_done: Condition variable for I/O completion.
  - hash_elem: Insert into the read-only cache
  - list_elem: Insert into the global frame list
- Global Structures:
  - frame_list : A global list of all frames in use.
  - read_only_frames: A hash table for caching shared, read-only file-backed pages.

1.2. Algorithms

SPT Lookup / Update – how your code accesses and updates SPT entries for a given page.
Accessed & Dirty-Bit – how you keep A/D bits consistent for aliased kernel/user mappings.

Ans:

- SPT Lookup / Update:
  - Lookup:
    Use pagedir_get_info(pd, upage) to retrieve the page_info associated

with a given virtual address (upage) in the page directory (pd).
If the page_info is NULL, the page is not currently mapped.
  - Update:
    Create a new page_info using pageinfo_create().
    Set the necessary fields (e.g., upage, type, writable, etc.) using the pageinfo_set_* functions.
    Use pagedir_set_info(pd, upage, page_info) to associate the page_info with the virtual address in the page directory.
- Accessed & Dirty-Bit Management:
  - Accessed Bit:
    Use pagedir_is_accessed(pd, upage) to check if the page has been accessed.
    Use pagedir_set_accessed(pd, upage, false) to clear the accessed bit when necessary.
  - Dirty Bit:
    Use pagedir_is_dirty(pd, upage) to check if the page has been modified.
    If the page is file-backed and writable, write the page back to the file if the dirty bit is set.
    Use pagedir_set_dirty(pd, upage, false) to clear the dirty bit after handling it.

1.3. Synchronization

how races are prevented when two user processes simultaneously request

Ans:
To prevent races, we implemented the methods below.
- Frame Table Lock:
  A global lock (frame_lock) is used to synchronize access to the frame table.
  Before modifying or accessing the frame table, acquire the lock using lock_acquire(&frame_lock).
  Release the lock after the operation using lock_release(&frame_lock).

- Per-Frame I/O Synchronization:
  Each frame has an io flag and a condition variable (io_done) to manage concurrent I/O operations.

Before performing I/O, check if io is set. If it is, wait on the io_done condition variable.
Signal the io_done condition variable after completing I/O.

- Page Directory Synchronization:
  Use atomic operations or locks to ensure that updates to the page directory are consistent when multiple processes access the same page.

- Swap Space Synchronization:
  The swap map is protected by internal synchronization mechanisms (e.g., bitmap_scan_and_flip ensures atomic allocation of swap sectors).
  Ensure that only one process can allocate or release a swap sector at a time.
  By combining these synchronization mechanisms, the system ensures that concurrent requests for the same resources (e.g., frames, swap space, or page directory entries) are handled safely and consistently.

2. Paging between Disk

2.1. Data Structures

As above, but only items related to paging and swap.

Ans:

struct page_info

• Contains a **swapped** flag plus the swap sector in data.swap_sector.
• Tracks the frame backing the page (if any).

```
bool swapped;
struct frame *frame;
```

struct frame

• Holds a kernel virtual address, a list of its page_info owners, and an I/O flag for ongoing reads/writes.

```
void *kpage;
struct list page_info_list;
bool io;
struct condition io_done;
```

Swap Bitmap

• Tracks which slots in the swap device are free or occupied.

```
static struct bitmap *swap_map;
```

2.2. Algorithms

Frame Eviction Policy – how you select a victim frame.

Frame Reuse Adjustment – when process P reuses a frame that belonged to process Q.

Ans:

- **Frame Eviction Policy**

  Uses a clock or second-chance approach by walking the global frame_list. Considers the accessed/dirty bits and picks a frame that is neither locked nor recently accessed.

```
static void * get_frame_to_evict (void)
{
 struct frame *cur, *start, *victim = NULL;
 struct page_info *pi;
 struct list_elem *e;
 bool accessed;

 ASSERT (!list_empty (&frame_list));
 start = list_entry (clock_hand, struct frame, list_elem);
 cur = start;

 do {
   accessed = false;
   ASSERT (!list_empty (&cur->page_info_list));

   for (e = list_begin (&cur->page_info_list);
        e != list_end (&cur->page_info_list); e = list_next (e))
     {
       pi = list_entry (e, struct page_info, elem);
       accessed |= pagedir_is_accessed (pi->pd, pi->upage);
       pagedir_set_accessed (pi->pd, pi->upage, false);
     }

   if (!accessed && cur->lock == 0)
```

```
      victim = cur;

    clock_hand = list_next (clock_hand);
    if (clock_hand == list_end (&frame_list))
      clock_hand = list_begin (&frame_list);
    cur = list_entry (clock_hand, struct frame, list_elem);

  } while (!victim && cur != start);

  if (victim == NULL)
    {
      ASSERT (cur == start);
      if (cur->lock > 0)
        PANIC ("no frame available for eviction");

      victim = cur;
      clock_hand = list_next (clock_hand);
      if (clock_hand == list_end (&frame_list))
        clock_hand = list_begin (&frame_list);
    }

  return victim;
}
```

- **Frame Reuse Adjustment**
  When a new process wants a frame used by another process, eviction writes modified data to swap or file, then reassigns the frame to the new user.
  The process is explained below.
  First, choose the frame to evict and prepare the frame for reuse (dissociation from process Q) by **get_frame_to_evict**(called in evict_frame) & **evict_frame** functions
  Then, allocate the frame to process P with **allocate_frame** function.
  Finally, map the frame to process P with **map_page** function.

evict_frame

```
static void * evict_frame (void)
{
  struct frame *f;
  struct page_info *pi;
```

```c
struct file_info *fi;
off_t written;
block_sector_t sector;
struct list_elem *e;
bool is_dirty = false;

f = get_frame_to_evict();

// Unmap and check dirty bits
for (e = list_begin (&f->page_info_list);
     e != list_end (&f->page_info_list); e = list_next (e))
  {
    pi = list_entry (e, struct page_info, elem);
    is_dirty |= pagedir_is_dirty (pi->pd, pi->upage);
    pagedir_clear_page (pi->pd, pi->upage);  // Force page fault
on next access
  }

// Handle dirty page or swap-only
if (is_dirty || (pi->writable & WRITABLE_TO_SWAP))
  {
    ASSERT (pi->writable != 0);
    f->io = true;
    f->lock++;

    if (pi->writable & WRITABLE_TO_FILE)
      {
        fi = &pi->data.file_info;
        lock_release (&frame_lock);
        written = process_file_write_at (fi->file, f->kpage,
                                         size (fi->end_offset),
                                         offset
(fi->end_offset));
        ASSERT (written == size (fi->end_offset));
      }
    else
      {
        lock_release (&frame_lock);
        sector = swap_write (f->kpage);
        }

      lock_acquire (&frame_lock);
      f->lock--;
      f->io = false;
      cond_broadcast (&f->io_done, &frame_lock);
    }
 else if ((pi->type & PAGE_TYPE_FILE) && pi->writable == 0)
    {
      ASSERT (hash_find (&read_only_frames, &f->hash_elem) !=
NULL);
      hash_delete (&read_only_frames, &f->hash_elem);
    }

 // Finalize eviction
 for (e = list_begin (&f->page_info_list); e != list_end
(&f->page_info_list); )
    {
      pi = list_entry (list_front (&f->page_info_list), struct
page_info, elem);
      pi->frame = NULL;
      if (pi->writable & WRITABLE_TO_SWAP)
        {
          pi->swapped = true;
          pi->data.swap_sector = sector;
        }
      e = list_remove (e);
    }

 memset (f->kpage, 0, PGSIZE);
 return f;
}
```

allocate_frame

```
static struct frame *allocate_frame (void)
{
 struct frame *frame;
 void *kpage;
  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
 if (kpage != NULL)
   {
     frame = calloc (1, sizeof *frame);
     if (frame != NULL)
       {
         frame_init (frame);
         frame->kpage = kpage;
         /* Add the frame to the end of the list so it becomes
eligible
            for eviction. */
         if (!list_empty (&frame_list))
           list_insert (clock_hand, &frame->list_elem);
         else
           {
             list_push_front (&frame_list, &frame->list_elem);
             clock_hand = list_begin (&frame_list);
           }
       }
     else
       palloc_free_page (kpage);
   }
 else
   frame = evict_frame ();
 return frame;
}
```

map_page

```
static void map_page (struct page_info *page_info, struct frame
*frame, const void *upage){
 page_info->frame = frame;
 list_push_back (&frame->page_info_list, &page_info->elem);
 pagedir_set_page (page_info->pd, upage, frame->kpage,
page_info->writable != 0);
 pagedir_set_dirty (page_info->pd, upage, false);
 pagedir_set_accessed (page_info->pd, upage, true);
}
```

2.3. Synchronization

VM-Wide Locking Strategy – basic design and how it avoids deadlock.
Cross-Process Eviction Safety – protecting Q while P evicts Q's page.
I/O In-Flight Protection – shielding a frame that is currently being read in.
Syscall-Time Page Access – handling page faults or "locked" frames during kernel system-call code.

Ans:

- **VM-Wide Locking Strategy**
  Basic Design:
  We used a global **frame_lock** to ensure mutual exclusion on frame-table modifications. This lock protects critical shared data structures related to the frame table, including
    - The global **frame_list** which tracks all physical frames
    - The **clock_hand** used by the eviction algorithm
    - The **page_info_list** within each struct frame, which lists all page_info structures currently mapped to that physical frame
    - The **lock** count and **io** status within each struct frame
    - The **read_only_frames** hash table.

  Any operation that modifies these structures or needs a consistent view of them must acquire frame_lock. This includes allocating a frame, evicting a frame, mapping a page to a frame, unmapping a page from a frame, and updating a frame's lock count or I/O status.

  Deadlock Avoidance:
  To avoid deadlock, as previously mentioned, we use frame_lock for frame table changes. This keeps things simple and avoids complicated deadlocks from multiple locks. When working with file system or swap, our system will not hold frame_lock during file I/O since it might block the system. Instead, lock just long enough to update frame state, then release it before calling blocking functions like swap_write(). Re-lock after I/O if needed. It is also crucial for our design to make sure that file I/O code should not try to lock frame_lock again — otherwise it could deadlock itself.

  As for I/O waits, threads use condition variables (like **io_done**) to wait for I/O. **cond_wait()** will release frame_lock while waiting, so other threads can continue VM work.

- **Cross-Process Eviction Safety**
  In our implementation, we use frame_lock to make the whole eviction atomic. While holding the lock, we will first pick the victim frame. Then loop through all page_info entries attached to it. After that, we will clear the page table of any process using it (like Q), and mark the page as not mapped. If it's dirty, we write it to swap or file.
  Since we hold the lock the whole time, Q can't mess with the frame while we're evicting it. If Q tries to use that page later, it gets a page fault and reloads it from the right source.

- **I/O In-Flight Protection**
  To implement I/O in-flight protection, we make sure that we don't touch a frame while it's being loaded from disk or written to swap.
  In our design, each frame has an io flag and a condition variable. When we start I/O, first we set **io = true**, lock the frame, and do the actual read/write. After I/O is done, we set **io = false** and wake up any waiting threads.
  Other threads that want to use the frame check the io flag first. If it's true, they wait on **io_done**.
  This way, we avoid race conditions and make sure no one evicts or uses a half-loaded frame. Also, since lock++ is used during I/O, the clock algorithm will skip that frame for eviction.

- **Syscall-Time Page Access**
  When a syscall accesses user memory, it might trigger a page fault if the page is not loaded or is waiting for I/O.
  We solve it by the following implementation. In the page fault handler, we call **frametable_load_frame()**, which waits for I/O if needed, loads the page from disk or swap, and maps it back to memory.
  If the kernel wants to safely access a user page multiple times, we use **frametable_lock_frame()** to "pin" it in memory so it won't be evicted. After we're done, we unlock it with **frametable_unlock_frame()**.

  This makes sure syscall code can safely access user buffers without race conditions or surprises during eviction or I/O.

frametable_load_frame()

```
bool frametable_load_frame(uint32_t *pd, const void *upage, bool
write) {
 return load_frame (pd, upage, write, false);
}
```

frametable_lock_frame()

```
bool frametable_lock_frame(uint32_t *pd, const void *upage, bool
write)
{
 return load_frame (pd, upage, write, true);
}
```

load_frame()

```
static bool load_frame (uint32_t *pd, const void *upage, bool
write, bool keep_locked)
{
 struct page_info *pi;
 struct file_info *fi;
 struct frame *f = NULL;
 void *src_kpage;
 off_t read_bytes;
 bool ok = false;

 ASSERT (is_user_vaddr (upage));
 pi = pagedir_get_info (pd, upage);
 if (pi == NULL || (write && !pi->writable))
   return false;

 lock_acquire (&frame_lock);
 wait_for_io_done (&pi->frame);

 ASSERT (pi->frame == NULL || keep_locked);
 if (pi->frame != NULL) {
   if (keep_locked)
     pi->frame->lock++;
   lock_release (&frame_lock);
   return true;
 }
```

```
if ((pi->type & PAGE_TYPE_FILE) && !pi->writable) {
  f = lookup_read_only_frame (pi);
  if (f != NULL) {
    map_page (pi, f, upage);
    f->lock++;
    wait_for_io_done (&f);
    f->lock--;
    ok = true;
  }
}

if (f == NULL) {
  f = allocate_frame ();
  if (f != NULL) {
    map_page (pi, f, upage);
    if (pi->swapped || (pi->type & PAGE_TYPE_FILE)) {
      f->io = true;
      f->lock++;
      if (pi->swapped) {
        lock_release (&frame_lock);
        swap_read (pi->data.swap_sector, f->kpage);
        pi->swapped = false;
      } else {
        if (!pi->writable)
          hash_insert (&read_only_frames, &f->hash_elem);
        fi = &pi->data.file_info;
        lock_release (&frame_lock);
        read_bytes = process_file_read_at (fi->file, f->kpage,
                                            size
(fi->end_offset),
                                            offset
(fi->end_offset));
        ASSERT (read_bytes == size (fi->end_offset));
      }
      lock_acquire (&frame_lock);
      f->lock--;
      f->io = false;
      cond_broadcast (&f->io_done, &frame_lock);
```

```
     } else if (pi->type & PAGE_TYPE_KERNEL) {
       src_kpage = (void *) pi->data.kpage;
       ASSERT (src_kpage != NULL);
       memcpy (f->kpage, src_kpage, PGSIZE);
       palloc_free_page (src_kpage);
       pi->data.kpage = NULL;
       pi->type = PAGE_TYPE_ZERO;
     }
     ok = true;
   }
 }

 if (ok && keep_locked)
   f->lock++;

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

frametable_unlock_frame()

```
void frametable_unlock_frame(uint32_t *pd, const void *upage)
{
 struct page_info *page_info;
  ASSERT (is_user_vaddr (upage));
 page_info = pagedir_get_info (pd, upage);
 if (page_info == NULL)
   return;
 ASSERT (page_info->frame != NULL);
 lock_acquire (&frame_lock);
 page_info->frame->lock--;
 lock_release (&frame_lock);
}
```

3. Stack Growth

3.1. Stack-Fault Heuristic

the conditions under which a page fault on an invalid address triggers automatic stack growth.

Ans:

A page fault on an invalid address fault_addr triggers automatic stack growth if all of the following 2 conditions are met:

1. **Page Not Already Mapped in SPT**: The page containing fault_addr (i.e., pg_round_down(fault_addr)) must not already have an entry in the process's supplemental page table (SPT). This is checked by pagedir_get_info(pd, upage) == NULL in **grow_stack**.
2. **Valid Stack Access Heuristic** (is_stack_access returns true):
   To check it is within a growable stack region, the fault_addr must be at or above **MAX_STACK_SIZE**. MAX_STACK_SIZE is defined as PHYS_BASE - PGSIZE * 64 (256KB below PHYS_BASE), establishing the lower bound for the stack. This prevents the stack from growing excessively into other memory regions or kernel space.

   To check the AND proximity/relation to stack pointer (esp), one of the following must be true, where esp is the user's stack pointer value at the time of the fault

   1. fault_addr is exactly 32 bytes below esp (i.e., esp - fault_addr == 32). This heuristically identifies a PUSHA instruction (which pushes 32 bytes).
   2. fault_addr is exactly 8 bytes below esp (i.e., esp - fault_addr == 8). This is a heuristic for PUSH instructions or similar operations. While a standard 32-bit PUSH decrements esp by 4, this specific offset might be chosen to catch certain instruction patterns or calling conventions, or it might be an artifact of how esp is captured relative to the faulting access.
   3. fault_addr is at or above esp (i.e., fault_addr >= esp). This covers cases where the access is to the current top of the stack or slightly "above" it (higher address) if that part of the page is unmapped. This can also handle the initial state where esp might point to the very top of an unallocated stack page.

In summary, a page fault is treated as a stack access if:

- The faulting address is within a “growable” range (vaddr >= MAX_STACK_SIZE).
- The difference esp - vaddr is small (e.g., 8 or 32), or vaddr >= esp (user-esp check).

If these conditions hold, grow_stack() allocates a new page.

- **Related code**

```
bool is_stack_access (const void *vaddr)
{
 struct thread *cur = thread_current ();
 void *esp = cur->user_esp;

 return (vaddr >= MAX_STACK_SIZE && ((esp - vaddr) == 8 || (esp -
vaddr) == 32 || vaddr >= esp));
}
```

```
void grow_stack (uint32_t *pd, const void *vaddr)
{
 struct page_info *page_info;
 void *upage = pg_round_down (vaddr);

 if (pagedir_get_info (pd, upage) == NULL && is_stack_access
(vaddr))
   {
     page_info = pageinfo_create ();
     if (page_info != NULL)
       {
         pageinfo_set_upage (page_info, pg_round_down (vaddr));
         pageinfo_set_pagedir (page_info, pd);
         pageinfo_set_type (page_info, PAGE_TYPE_ZERO);
         pageinfo_set_writable (page_info, WRITABLE_TO_SWAP);
         pagedir_set_info (pd, upage, page_info);
       }
   }
}
```

4. Test results

We passed all of the tests under the vm folder, including the bonus part. Below is a high level explanation of how we pass all of the tests.

When a program calls mmap, it sets up page table entries for a file-backed memory region, but doesn't load any data yet (lazy loading). SPT stores the file, offset, and length for each page.

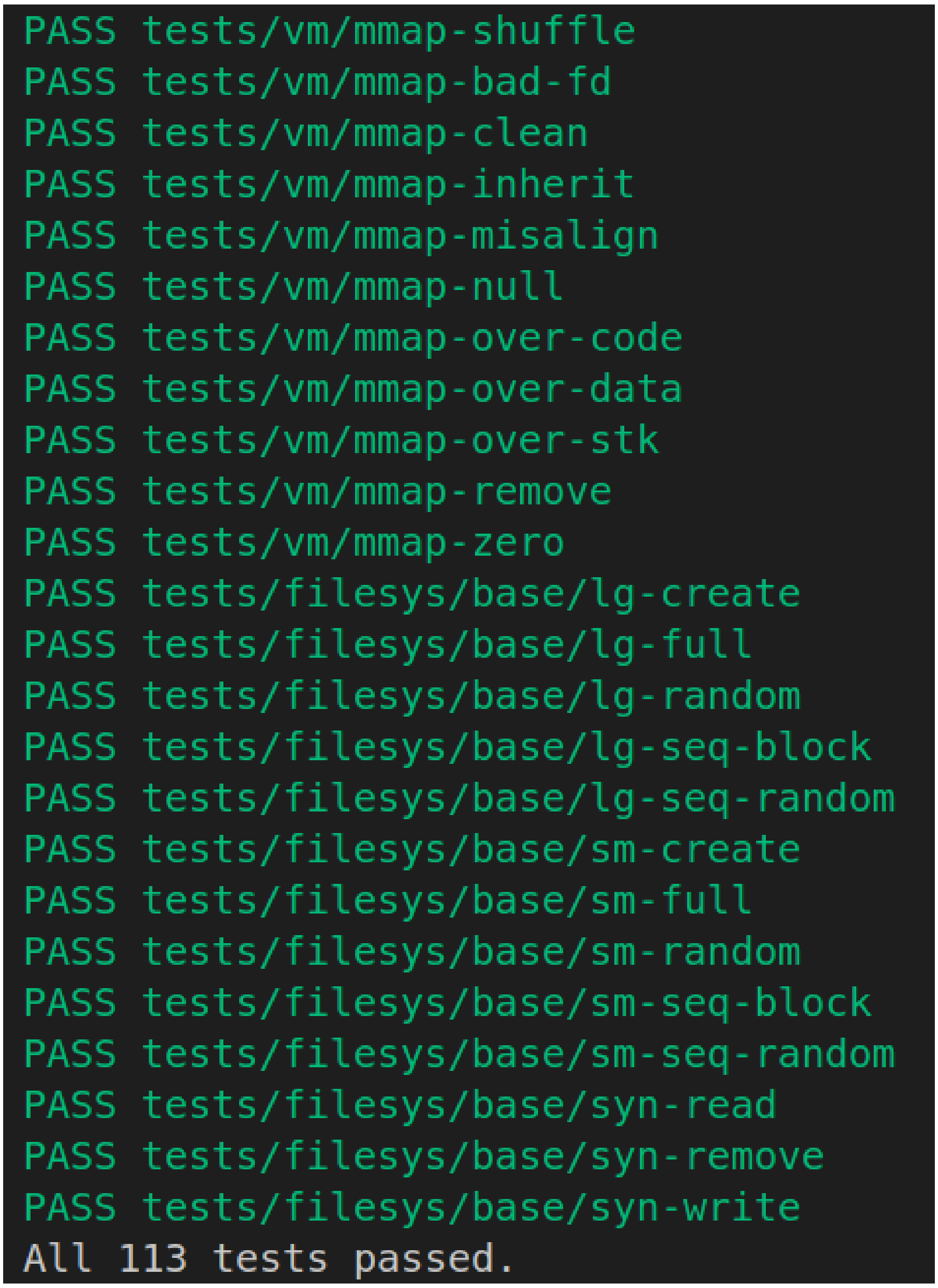
When a process accesses a page in the mmaped region for the first time, a page fault occurs, and the corresponding page is read from the file into a frame.

When munmap is called or the process exits, our code will perform the following operations.

1. Dirty pages in memory are written back to the file.
2. Page table entries are cleared.
3. Frames are freed if they're not shared.
4. The file descriptor is also managed (reopened on mmap, closed on munmap).

The mmap system checks whether the following requirements are met.

1. The file descriptor is valid.
2. The address is page-aligned and not null.
3. Length is non-zero.
4. The mapping doesn't overlap other regions.
5. STDIN and STDOUT aren't allowed.


```
PASS 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
All 113 tests passed.
```