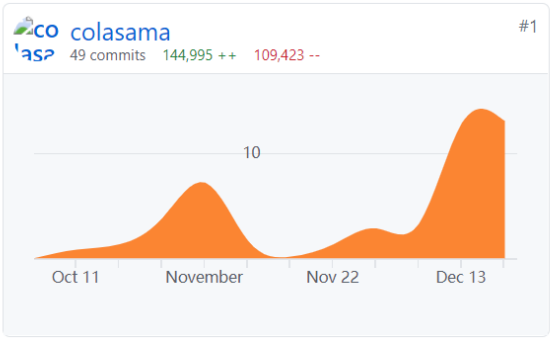
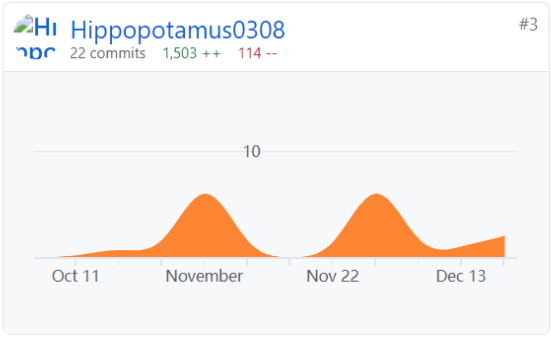
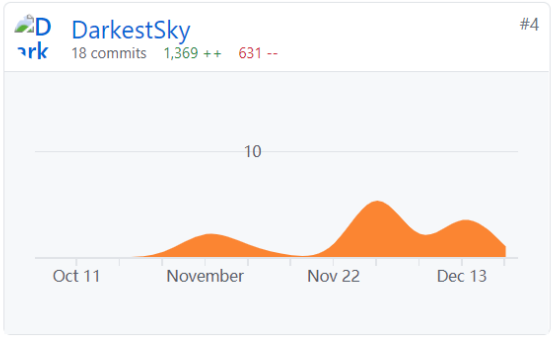
小组相关

姓名	学号	贡献百分比
张涵珂	18373734	25%
杨壮	17376193	25%
裴昱	17375244	25%
杨瑞羿	17241055	25%

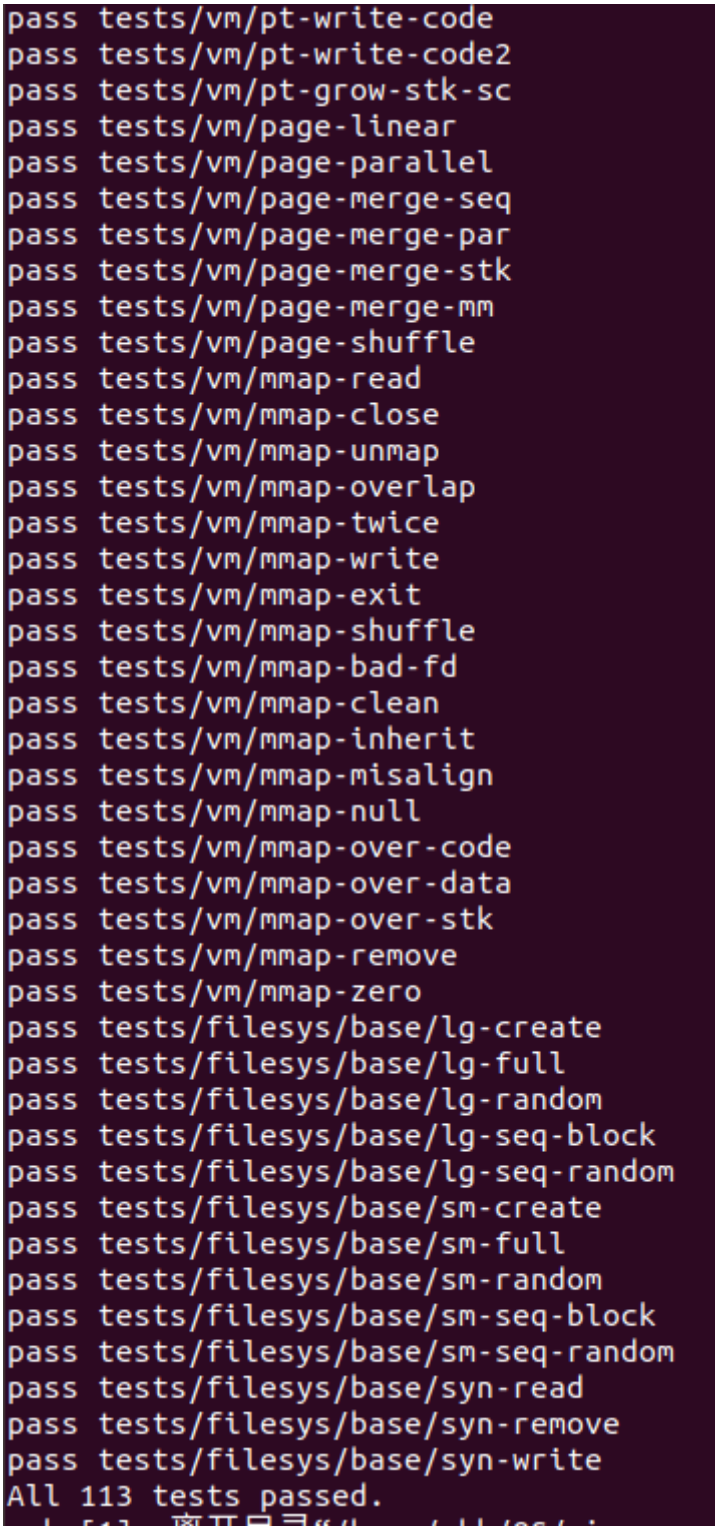
Github截图







通过用例情况



需求分析

Paging

1.概述

在 Project 2 中，可执行文件在load阶段直接加载到代码段中。而在Project 3中，需要实现可执行文件的懒加载，在真正访问代码时才加载。在 page-*测试点中，程序先通过

```
#define SIZE (2 * 1024 * 1024)

static char buf[SIZE];
```

分配了很大的静态内存，消耗了所有的帧，因此代码段会无帧可用，load失败。

2. 题目分析

可以把可执行文件视为普通的只读文件，那么可执行文件的懒加载就和 Memory Map 的实现思路一样，在load时为每页可执行文件分配一个 `struct page` ，里面储存着必要的文件相关的信息。当程序真正访问可执行文件时，会触发 `page fault` ，这是程序再找到相关信息，获得一个帧，再将文件内容写入该帧。这样就实现了程序的懒加载。

唯一的区别只不过是可执行文件是只读的，因此在pagedir中安装页时要设置 writable 为 false。这样任何对可执行文件的修改都将抛出 rights violation 异常。

同时，为可执行文件分配的帧需要设置为private，这样可以保证evict时这些帧不会被swap到交换区。

Stack Growth

1.概述

在project2中，栈是单页的，整个程序的运行受栈大小的限制。而如果栈需要进行扩大，我们则需要分配额外的页面。同时，和大多数操作系统一样，我们需要对堆栈的大小施加一定的绝对限制，将其大小限制在一定范围内。

2.题目分析

根据题目的要求，我们需要在栈需要进行扩大的时候才分配额外的页面，所以我们需要设计一种试探性的方法来将堆栈的访问和其他访问区分开。

由于程序在堆栈指针下方写入堆栈，会出现bug，考虑到80x86的PUSH指令会在调整堆栈指针前先检查访问权限，因此可能会导致堆栈指针下方4字节的页面错误。类似，PUSHA指令一次性压入32字节，可能会在堆栈指针以下32字节出错。而为了判断堆栈是否需要扩大，我们需要设计一个方法来获取用户程序堆栈指针的当前值。

对于第一个堆栈的页，无需延迟分配，可以在加载时使用命令行参数之间分配和初始化它，无需等待其出错。同时所有栈的页应该能被驱逐，被驱逐的页需要被写进交换区。同时考虑到frame锁的要求，被驱逐的页必须要有一个有锁的frame。具体的操作见下。

Memory Mapped Files

1. 概述

在 Project 2 中，通过 `read()` 和 `write()` 两个系统调用实现了对文件的读写功能。在本项中，将实现文件操作的另一个接口：可以通过 `mmap()` 系统调用将打开的文件内容映射到内存中，之后可以通过对内存的操作来实现对文件内容的读写；以及通过 `munmap()` 系统调用来移除文件映射，修改过的页将被写回文件。

2. 题目分析

根据题目的描述，关于文件内存映射的设计存在一些限制和要求，整理如下：

1. 进程因为种种原因退出时，需要能够将进行了的映射全部撤销；
2. 撤销映射时，要保证所有写过的页在撤销过程中被写回文件，而其他的页不执行写入，再页从进程的列表中移除；
3. 删除或关闭文件 不会对映射产生影响；
4. 两个或以上进程映射同一文件时，不要求他们数据的一致性。

为了正确执行 `mmap()` 和 `munmap()` ，最重要的在于记录「哪些内存空间」映射了「哪些文件」，才能在进程退出等需要时对相应的映射进行操作。具体设计见下。

Accessing User Memory

1. 概述

和project2中系统调用时检查用户栈指针一样，内核必须非常小心，因为用户可以传递空指针，指向未映射的虚拟内存的指针或指向内核虚拟地址空间的指针（位于上方 `PHYS_BASE` ）。通过终止有问题的进程并释放其资源，必须拒绝所有这些类型的无效指针，从而不会损害内核或其他正在运行的进程。

2. 题目分析

有两种合理的方法可以正确地执行此操作。

第一种方法是验证用户提供的指针的有效性，然后取消引用它。通过 `userprog/pagedir.c` 和 `threads/vaddr.h` 中的函数进行处理即可完成。第二种方法则是仅检查用户指针是否指向下方 `PHYS_BASE` ，然后取消引用。无效的用户指针将导致“页面错误”，在 `userprog/exception.c` 中的 `page_fault()` 进行相关修复即可。该技术通常更快，因为它利用了处理器的MMU。我们本次则是使用第二种方法来完成。具体可参考下方功能设计。

功能设计

Paging

在 `load` 函数中，先打开文件，读取可执行程序的 `Program header` ，再根据程序头的信息从相应的文件位置载入代码。因此，只需要修改 `load_segment` ，将原先直接分配帧的方式改为懒加载即可

原来的代码：

```
      /* 拿到一个帧 */
      uint8_t *kpage = palloc_get_page (PAL_USER);

      /* 从文件中读取代码 */
      file_read (file, kpage, page_read_bytes);

      /* 将剩余部分置为0 */
      memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* 安装帧，将物理地址和用户虚拟地址联系在一起 */
      install_page (upage, kpage, writable);
```

现在：

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* 计算出要读取的字节数，清零的字节数 */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;
      /* 将用户虚拟地址和一个 struct page 绑定在一起 */
      struct page *p = page_allocate (upage, !writable);
      if (p == NULL)
        return false;
      /* 为这个 struct page 提供必要的信息，以便page fault的时候读取文件 */
      if (page_read_bytes > 0)
        {
          p->file = file;
          p->file_offset = ofs;
          p->file_bytes = page_read_bytes;
        }
      /* 为下一页作准备 */
      read_bytes -= page_read_bytes;
```

```
      zero_bytes -= page_zero_bytes;
      ofs += page_read_bytes;
      upage += PGSIZE;
    }
  return true;
}
```

接下来就是在 `page_fault` 的时候安装页了

```
 else if (p->file != NULL)
   {
     // 读文件
     off_t rb = file_read_at (p->file, p->frame->base, p->file_bytes, p->file_offset);
     off_t zb = PGSIZE - rb;
     memset (p->frame->base + rb, 0, zb);
   }
```

在 `install_page_in` 中，如果 `p->file` 不为 `NULL` ，就说明这个page是文件懒加载产生的，就会从根据相应信息从文件读取内容。

Stack Growth

```
static struct page *
get_page_with_addr (const void *address)
{
  if (address < PHYS_BASE)
    {
      struct page p;
      struct hash_elem *e;

      /* Find existing page. */
      p.addr = (void *) pg_round_down (address);
      e = hash_find (thread_current ()->pages, &p.hash_elem);
      if (e != NULL)
        return hash_entry (e, struct page, hash_elem);
      /* No page.  Expand stack? */
  /* add code */
      if((p.addr > PHYS_BASE - STACK_MAX) && (thread_current()->user_esp - 32 < address))
        return page_allocate(p.addr, false);
    }
  return NULL;
}

```

get_page_with_addr()描述如上

在get_page_with_addr()函数中，我们会返回一个包含指定虚拟地址的页（如果能找到此页），同时调用了page_allocate()函数用来分配新的页给当前堆栈，函数流程如下：

函数传入参数address，先将其与PHYS_BASE做比较，如果小的话说明需要扩展堆栈；

新建一个页 p，将p的地址利用pg_roung_down()转到最近的边界处,再试图利用hash_find()找到p中元素

上述功能都是在寻找已经建立好的页

接着判断(p.addr > PHYS_BASE - STACK_MAX) && (thread_current()->user_esp - 32 < address)是否为真，此处需要这样判断来避免出现上述题目分析中提到的错误，如果符合条件，跳转至page_allocate，page_allocate()函数实现了栈增长

```
struct page *
page_allocate (void *vaddr, bool read_only)
{
  struct thread *t = thread_current ();
  struct page *p = malloc (sizeof *p);
  if (p != NULL)
    {
      p->addr = pg_round_down (vaddr);
```

```
      p->read_only = read_only;
      p->private = !read_only;

      p->frame = NULL;

      p->sector = (block_sector_t) -1;

      p->file = NULL;
      p->file_offset = 0;
      p->file_bytes = 0;

      p->thread = thread_current ();

      if (hash_insert (t->pages, &p->hash_elem) != NULL)
        {

          free (p);
          p = NULL;
        }
    }
  return p;
}
```

Memory Mapped Files

首先，为了每个进程能够维护自己的内存映射文件，在 `struct thread` 中添加了必要的 list 项：

```
struct thread
  {
    // ...
    struct list mappings;               /* 映射列表 */
    // ...
  }
```

list 实际的结构体为描述内存映射文件的结构体，其中维护了相对应的文件指针、以及映射到的内存起始点和占用页的数量。详细设计如下：

```
struct mapping {
    struct list_elem elem;      /* 列表元素. */
    int handle;                 /* 映射的id */
    struct file *file;
    uint8_t *base;              /* 内存映射基地址 */
    size_t page_cnt;            /* 映射的数量 */
};
```

根据这一设计，具体实现了 `mmap()` 和 `munmap()` 系统调用的处理程序具体实现分别如下：

```
/* Mmap system call. */
int
sys_mmap (int handle, void *addr)
{
  struct fd *fd = lookup_fd (fd);    // 在进程打开的文件中查找目标文件的描述符
  struct mapping *m = malloc (sizeof *m);             // 申请空间
  size_t offset;
  off_t length;

  if (m == NULL || addr == NULL || pg_ofs (addr) != 0)    // 有效性检查
    return -1;

  m->handle = thread_current ()->next_handle++;       // ↓ 设置 mapping 的内容
  lock_acquire (&fs_lock);
  m->file = file_reopen (fd->file);
  lock_release (&fs_lock);
```

```
  if (m->file == NULL)
    {
      free (m);
      return -1;
    }
  m->base = addr;
  m->page_cnt = 0;                                      // ↑ 设置 mapping 的内容
  list_push_front (&thread_current ()->mappings, &m->elem);   // 加入列表

  offset = 0;
  lock_acquire (&fs_lock);
  length = file_length (m->file);
  lock_release (&fs_lock);
  while (length > 0)        // 根据文件长度分配内存页
    {
      struct page *p = page_allocate ((uint8_t *) addr + offset, false);
      if (p == NULL)
        {
          unmap (m);
          return -1;
        }
      p->private = false;
      p->file = m->file;
      p->file_offset = offset;
      p->file_bytes = length >= PGSIZE ? PGSIZE : length;
      offset += p->file_bytes;
      length -= p->file_bytes;
      m->page_cnt++;
    }

  return m->handle;
}
```

```
/* Munmap system call. */
int
sys_munmap (int mapping)
{
/* add code here */
  unmap(lookup_mapping(mapping));       // 调用逻辑函数，见下
  return 0;
}

/* Remove mapping M from the virtual address space,
   writing back any pages that have changed. */
void
unmap (struct mapping *m)
{
  list_remove(&m->elem);        // 移除列表元素
  for(int i = 0; i < m->page_cnt; i++)      // 检查各页是否有写操作
  {
    //Pages written by the process are written back to the file
    if (pagedir_is_dirty(thread_current()->pagedir, m->base + (PGSIZE * i)))
    {
      lock_acquire(&fs_lock);
      file_write_at(m->file, m->base + (PGSIZE * i), PGSIZE, (PGSIZE * i)); // Check 3rd parameter
      lock_release(&fs_lock);
    }
  }
  for(int i = 0; i < m->page_cnt; i++)    // 释放页
  {
    page_deallocate(m->base + (PGSIZE * i));
  }
}
```

Accessing User Memory

为了验证用户指针的有效性，在每次系统调用之前我们都通过上文所述的第二种方法实现对用户指针有效性的验证。具体实现可参考下面代码。

```
```

```
static void syscall_handler (struct intr_frame *f) {
    ...
    copy_in (&call_nr, f->esp, sizeof call_nr);
    ...
}
static void copy_in (void *dst_, const void *usrc_, size_t size) {
  ...
      if (!page_lock (usrc, false))
        thread_exit ();
  ...
}

bool page_lock (const void *addr, bool will_write) {
  struct page *p = get_page_with_addr (addr);
  if (p == NULL || (p->read_only && will_write))
    return false;

  ...
}

static struct page * get_page_with_addr (const void *address) {
  if (address < PHYS_BASE){
      ...
  }
  return NULL;
}
```

`syscall_handler()->copy_in()->page_lock()->get_page_with_addr()->page_lock()->copy_in()->thread_exit()`

我们的逻辑就是通过在syscall_handler()函数中，进行用户指针的一个处理，如果用户指针没问题，则可以通过其指针访问内存获取对应数据，如果存在问题，则最终会在get_page_with_addr()函数中返回NULL，导致page_lock()函数页返回false，并且进而导致在copy_in()函数中因为用户指针的无效导致的进程退出，即thread_exit()，从而完成一个用户指针的校验。总流程可参考上面一行函数的执行过程。

相关问题的回答

PAGE TABLE MANAGEMENT

DATA STRUCTURES

A1: Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration. Identify the purpose of each in 25 words or less.

```
//vm/frame.h
struct frame
{
struct lock lock;           /* 锁，防止同时访问. */
struct page *page;          /* 已映射的进程页. */
void *base;                 /* 对应内核虚拟基址. */
bool pinned;                 /* 防止frame在获取资源时被evicted. */
};
//vm/frame.c
static struct frame *frames;    /* frames.*/
static struct lock vm_sc_lock;  /* 锁，避免race和多线程访问.*/
static size_t hand;             /* 获取某个帧的具体位置.*/
static size_t f_count;          /* 帧的大小. */

//vm/page.h
struct page
{
struct thread *thread;      /* 拥有的线程. */
void *addr;                 /* 用户虚拟地址. */
bool r_only;                /* 是否只读 */
bool dirty;
struct hash_elem elem;      /* 本结构体的hash_elem. */
block_sector_t sector;      /* swap的起始sector位置. */

struct frame *frame;
```

```
26  /* Memory-mapped file相关的字段，也可用于页面懒加载 */
27  bool private;               /* False则file,true则swap. */
28  struct file *file;          /* 文件指针. */
29  off_t file_offset;          /* 文件的偏移量. */
30  off_t file_bytes;           /* 文件大小. */
31  };
```

ALGORITHMS

A2: In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

对于一个给定的页，其包含很多子成员，其中包括了存放数据的frame。Frame的struct里包含了一个指向内核中存放数据位置的指针，同时还有一个struct page *page，来存放引用了数据的页。页最初创建时， 其struct里的页是设为NULL的，只有通过frame.c中的frame_alloc_and_lock（）函数分配，页才能获取到frame

A3: How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?

在我们的设计中，我们采取了只能使用用户虚拟地址来获取数据来避免这个问题。

SYNCHRONIZATION

A4: When two user processes both need a new frame at the same time, how are races avoided?

我们在frame.c中加入锁scan_lock，使得一次只能在一个进程里搜索frame table。在此基础上，两个进程无法同时争取同一个frame，因此竞争也被避免了。另外，每个单独的frame的struct里都包含其自己的锁（frame-> lock），表示它是否被占用。

RATIONALE

A5: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

在之前的project中，一个进程的所有页都是映射在该进程的页表里的，而现在我们需要懒加载页，所以我们将页面目录扩展为每个进程的补充页表，并使用用户虚拟地址作为该表项基于以下内容隐式指示用户虚拟地址流程的页面基础。于是所有映射在frame里，当前在内存中的页现在都可以被分配到原始页表里，而没有加载的页则可以在补充页表中找到。

PAGING TO AND FROM DISK

DATA STRUCTURES

B1: Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration. Identify the purpose of each in 25 words or less.

```
1  //vm/page.h
2  struct page {
3      /* Memory-mapped file相关的字段，也可用于页面懒加载 */
4      bool private;               /* False则file,true则swap. */
5      struct file *file;          /* 文件指针. */
6      off_t file_offset;          /* 文件的偏移量. */
7      off_t file_bytes;           /* 文件大小. */
8  };
```

用于页面懒加载以及相关文件的操作。

ALGORITHMS

B2: When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

遍历所有已分配的帧，跳过所有满足以下条件的帧：

- 被设置为pinned
- 访问位为true

B3: When a process P obtains a frame that was previously used by a process Q, how do you adjust the page table (and any other data structures) to reflect the frame Q no longer has?

1. 在 Q 的 pagedir 中清楚对应的虚拟地址。 `pagedir_clear_page(p->thread->pagedir, p->addr);`
2. 在 Q 的pages中，将虚拟地址对应的 page的frame设为NULL。 `p->frame = NULL;`

B4: Explain your heuristic for deciding whether a page fault for an invalid virtual address should cause the stack to be extended into the page that faulted.

不会，如果访问无效的虚拟地址，进程会直接退出

```
if((page.addr > PHYS_BASE - S_MAXSIZE) && (thread_current()->user_esp - 32 < address))
  return page_allocate(page.addr, false);
return NULL;
```

检测是否合法，不合法直接返回NULL，再有page_fault退出

SYNCHRONIZATION

B5: Explain the basics of your VM synchronization design. In particular, explain how it prevents deadlock. (Refer to the textbook for an explanation of the necessary conditions for deadlock.)

1. 为交换区添加了一个全局锁 `s_lock` ，为每次 `bitmap_scan_and_flip` 提供保护
2. 为帧表frames添加了一个全局锁 `vm_sc_lock` ，每次对frames操作时（即frame_alloc_and_lock和page_deallocate）都加锁保护

B6: A page fault in process P can cause another process Q's frame to be evicted. How do you ensure that Q cannot access or modify the page during the eviction process? How do you avoid a race between P evicting Q's frame and Q faulting the page back in?

牺牲帧的过程并不是一个原子操作，但我们只需要保证，在 P 获得新帧之前，在 Q 的 `pagedir` 中将其与Q的关系清除即可。这样就可以保证，Q 执行 `palloc_get_page` 时，要么获得一个合法的帧，要么 `page_fault` ，而不会得到一个非法的帧。

B7: Suppose a page fault in process P causes a page to be read from the file system or swap. How do you ensure that a second process Q cannot interfere by e.g. attempting to evict the frame while it is still being read in?

每个frame有 `pinned` 字段，初始值设为 `true` ，只有在从文件或者交换区读完数据后才会被设为 `false` ，这个帧才允许被牺牲。

B8: Explain how you handle access to paged-out pages that occur during system calls. Do you use page faults to bring in pages (as in user programs), or do you have a mechanism for "locking" frames into physical memory, or do you use some other design? How do you gracefully handle attempted accesses to invalid virtual addresses?

我们的实现是第一种，在需要的时候 `page in` 。为每一个虚拟地址生成一个 `struct page` ，若其被 `swap out` ，则调用 `pagedir_clear_page` ，以便在下次CPU使用时产生一个 `page fault` 。在生成一个 `page` 之前，先检查该用户虚拟地址是否已经有了一个 `page` ，若已经有了，说明这个虚拟地址已经被使用过了 `（mmap/data segment/code segment/stack segment）` ，是无效的。

RATIONALE

B9: A single lock for the whole VM system would make synchronization easy, but limit parallelism. On the other hand, using many locks complicates synchronization and raises the possibility for deadlock but allows for high parallelism. Explain where your design falls along this continuum and why you chose to design it this way.

我们并不行引入过多的锁，所以只为全局的帧表和交换区引入了两个锁。在仔细思考后，我们发现只要保证在··palloc_get_page `不为` NULL`时，获得的帧是合法的，就不会导致同步错误的问题。所以，尽管并没有为每个帧使用一个锁，我们的程序依然能够通过测试点。

MEMORY MAPPED FILES

DATA STRUCTURES

C1: Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef` , or enumeration. Identify the purpose of each in 25 words or less.

```
// userprog/syscall.c
struct mapping {
    struct list_elem elem;      /* 列表元素. */
    int handle;                 /* 映射的id */
    struct file *file;
    uint8_t *base;              /* 内存映射基地址 */
    size_t page_cnt;            /* 映射的数量 */
};
```

mapping结构体，用于在文件映射时进行记录相应的内存基地址以及映射的文件等等内容。

ALGORITHMS

C2: Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages.

一个线程包含其打开的所有文件，且结合上文所说的mapping结构体即可实现。内存映射文件封装在mapping结构体中。每个映射都包含对其在内存中的地址的引用，以及它映射的文件。每个线程都包含映射到该线程的所有文件的列表，这些文件可用于管理哪些文件直接存在于内存中。否则，包含内存映射文件信息的页面的管理方式与其他页面相同。

不同之处在于：对于属于内存映射文件的页，页错误和逐出过程稍微存在不同。收回时，与文件无关的页将被移动到交换分区，而不管该页是否脏。收回时，只有在修改内存映射文件页时，才能将其写回文件。否则，不需要写入。而对于内存映射文件，交换分区是完全避免的。

C3: Explain how you determine whether a new file mapping overlaps any existing segment.

我们只有在找到空闲且未映射的页时，才会分配用于新文件映射的页。page_allocated()函数可以访问所有现有的文件映射，并将拒绝分配任何已占用的空间。page_allocate()函数只会分配空闲且未映射的页。它将尝试将新映射添加到页表，如果页已存在，它将取消映射并返回NULL。

RATIONALE

C4: Mappings created with "mmap" have similar semantics to those of data demand-paged from executables, except that "mmap" mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain why your implementation either does or does not share much of the code for the two situations.

任何页面，无论其来源如何，最终都将通过page_out()函数进行分页。唯一的区别是检查页面是否应该写回磁盘。如果页面被标记为私有，那么它会被交换到交换分区，否则它应该被写到磁盘上的文件中。