LAB4：

编程题：

第一题：** 使用 sbrk，mmap,munmap,mprotect 内存相关系统调用的 linux 应用程序。

我们可以设计程序的代码如下：

```c
int main()
{
        printf("Test sbrk start.\n");
        uint64 PAGE_SIZE = 0x1000;
        uint64 origin_brk = sbrk(0);
        printf("origin break point = %p\n", origin_brk);
        uint64 brk = sbrk(PAGE_SIZE);
        if(brk != origin_brk){
                return -1;
        }
        brk = sbrk(0);
        printf("one page allocated, break point = %p\n", brk);
        printf("try write to allocated page\n");
        char *new_page = (char *)origin_brk;
        for(uint64 i = 0;i < PAGE_SIZE;i ++) {
                new_page[i] = 1;
        }
        printf("write ok\n");
        sbrk(PAGE_SIZE * 10);
        brk = sbrk(0);
```

```c
        brk = sbrk(0);
        printf("one page allocated, break point = %p\n", brk);
        printf("try write to allocated page\n");
        char *new_page = (char *)origin_brk;
        for(uint64 i = 0;i < PAGE_SIZE;i ++) {
                new_page[i] = 1;
        }
        printf("write ok\n");
        sbrk(PAGE_SIZE * 10);
        brk = sbrk(0);
        printf("10 page allocated, break point = %p\n", brk);
        sbrk(PAGE_SIZE * -11);
        brk = sbrk(0);
        printf("11 page DEALLOCATED, break point = %p\n", brk);
        printf("try DEALLOCATED more one page, should be failed.\n");
        uint64 ret = sbrk(PAGE_SIZE * -1);
        if(ret != -1){
```

```
        uint64 ret = sbrk(PAGE_SIZE * -1);
        if(ret != -1){
                printf("Test sbrk failed!\n");
                return -1;
        }
        printf("Test sbrk almost OK!\n");
        printf("now write to deallocated page, should cause page fault.\n");
        for(uint64 i = 0;i < PAGE_SIZE;i ++){
                new_page[i] = 2;
        }
        return 0;
}
```

把该程序放到 user/src 目录下，运行程序，我们能看到结果：

```
Test sbrk start.
origin break point = 16000
one page allocated,  break point = 17000
try write to allocated page
write ok
10 page allocated,  break point = 21000
11 page DEALLOCATED,  break point = 16000
try DEALLOCATED more one page, should be failed.
Test sbrk almost OK!
```

表明该程序使用了 sbrk 系统调用。

```c
int main() {
    // 打开文件
    int fd = open("ttt.txt", O_RDWR);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    // 获取文件的大小
    struct stat sb;
    if (fstat(fd, &sb) == -1) {
        perror("Error getting file size");
        return 1;
    }

    // 执行内存映射
    char *mapped = (char *)mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (mapped == MAP_FAILED) {
        perror("Error mapping file");
        return 1;
    }

    // 现在可以通过映射区域来访问文件内容了
    // 例如，修改文件的一个字符
    mapped[0] = 'J';

    // 同步映射区域到文件
    if (msync(mapped, sb.st_size, MS_SYNC) == -1) {
        perror("Error syncing file");
    }

    // 解除映射
    if (munmap(mapped, sb.st_size) == -1) {
        perror("Error un-mapping file");
    }

    // 关闭文件
    close(fd);
    return 0;
}
```

Mmap.c

```c
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    // 分配一块内存
    size_t pageSize = getpagesize();
    void *mem = mmap(NULL, pageSize, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    strcpy((char *)mem,"Hello");
    if (mem == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    // 修改这块内存的保护属性为只读
    if (mprotect(mem, pageSize, PROT_READ) == -1) {
        perror("mprotect");
        return 1;
    }

    // 尝试写入这块内存（会导致程序
    strcpy((char *)mem, "world"); // 这一行会导致段错误（segmentation fault）
    // 清理资源
    if (munmap(mem, pageSize) == -1) {
        perror("munmap");
        return 1;
    }

    return 0;
}
```

Mprotect.c

编程题二：

修改本章操作系统内核，实现任务和操作系统内核共用同一张页表的单页表机制。

要实现任务和操作系统内核通用一张页表，需要了解清楚内核地址空间和任务地址空间的布局，然后为每个任务在内核地址空间中单独分配一定的地址空间。

在描述任务的 struct proc 中添加新的成员“kpgtbl”、“trapframe_base”，前者用户保存内核页表，后者用于保存任务的 TRAPFRAME 虚地址。并增加获取内核页表的函数“get_kernel_pagetable()”。

```c
void usertrapret()
{
    set_usertrap();
    struct trapframe *trapframe = curr_proc()->trapframe;
    trapframe->kernel_satp = r_satp(); // kernel page table
    trapframe->kernel_sp =
            curr_proc()->kstack + KSTACK_SIZE; // process's kernel stack
    trapframe->kernel_trap = (uint64)usertrap;
    trapframe->kernel_hartid = r_tp(); // unuesd
    w_sepc(trapframe->epc);
    // set up the registers that trampoline.S's sret will use
    // to get to user space.
    // set S Previous Privilege mode to User.
    uint64 x = r_sstatus();
    x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
    x |= SSTATUS_SPIE; // enable interrupts in user mode
    w_sstatus(x);
    // tell trampoline.S the user page table to switch to.
    uint64 satp = MAKE_SATP(curr_proc()->pagetable);
    uint64 fn = TRAMPOLINE + (userret - trampoline);
    tracef("return to user @ %p", trapframe->epc);
    ((void (*)(uint64, uint64))fn)(curr_proc()->trapframe_base, satp); //使用任务自己的TRAPFRAME
    //((void (*)(uint64, uint64))fn)(TRAPFRAME, satp);
}
```

Trap.c

```
pagetable_t get_kernel_pagetable(){
    return kernel_pagetable;
}
```

修改 vm.c。

```
struct proc {
    enum procstate state; // Process state
    int pid; // Process ID
    pagetable_t pagetable; // User page table
    uint64 ustack;
    uint64 kstack; // Virtual address of kernel stack
    struct trapframe *trapframe; // data page for trampoline.S
    struct context context; // swtch() here to run process
    uint64 max_page;
    uint64 program_brk;
    uint64 heap_bottom;
    pagetable_t kpgtbl; // 增加kpgtbl，用于保存内核页表
    uint64 trapframe_base; // 增加trapframe，用于保存任务自己的trapframe
}
```

修改 proc.h。

```
pagetable_t bin_loader(uint64 start, uint64 end, struct proc *p, int num)
{
    //pagetable_t pg = uvmcreate(); //任务不创建自己的页表
    pagetable_t pg = get_kernel_pagetable(); //获取内核页表
    uint64 trapframe = TRAPFRAME - (num + 1)* PAGE_SIZE; // 为每个任务依次指定TRAPFRAME
    if (mappages(pg, trapframe, PGSIZE, (uint64)p->trapframe,
                 PTE_R | PTE_W) < 0) {
            panic("mappages fail");
    }
    if (!PGALIGNED(start)) {
            panic("user program not aligned, start = %p", start);
    }
    if (!PGALIGNED(end)) {
            // Fix in ch5
            warnf("Some kernel data maybe mapped to user, start = %p, end = %p",
                  start, end);
    }
    end = PGROUNDUP(end);
    uint64 length = end - start;
    uint64 base_address = BASE_ADDRESS + (num * (p->max_page + 100)) * PAGE_SIZE; //设置任务的起始地址，并为任务保留100个页
用做堆内存
    if (mappages(pg, base_address, length, start,
                 PTE_U | PTE_R | PTE_W | PTE_X) != 0) {
            panic("mappages fail");
    }
    p->pagetable = pg;
    uint64 ustack_bottom_vaddr = base_address + length + PAGE_SIZE;
    if (USTACK_SIZE != PAGE_SIZE) {
            // Fix in ch5
            panic("Unsupported");
    }
    mappages(pg, ustack_bottom_vaddr, USTACK_SIZE, (uint64)kalloc(),
             PTE_U | PTE_R | PTE_W | PTE_X);
    p->ustack = ustack_bottom_vaddr;
    p->trapframe->epc = base_address;
    p->trapframe->sp = p->ustack + USTACK_SIZE;
    p->max_page = PGROUNDUP(p->ustack + USTACK_SIZE - 1) / PAGE_SIZE;
    p->program_brk = p->ustack + USTACK_SIZE;
    p->heap_bottom = p->ustack + USTACK_SIZE;
    p->trapframe_base = trapframe; //任务保存自己的TRAPFRAME
```

修改 loader.c。

```
pub const USER_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_HEAP_SIZE: usize = 0x30_0000;
pub const MEMORY_END: usize = 0x80800000;       // 可用物理内存的末尾地址为0x80800000，(起始地址为)0x80000000，两者正好相差8MiB
pub const PAGE_SIZE: usize = 0x1000;
pub const PAGE_SIZE_BITS: usize = 0xc;

pub const TRAMPOLINE: usize = usize::MAX - PAGE_SIZE + 1;       // 0xFFFFFFFFFFFFF000
pub const TRAP_CONTEXT: usize = TRAMPOLINE - PAGE_SIZE;         // 0xFFFFFFFFFFFFE000
/// Return (bottom, top) of a kernel stack in kernel space.
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);    // 分配内核空间，注意不同的内核栈空间隔着一张页表
    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
}
```

修改 config.rs。

编程题三：

扩展内核，支持基于缺页异常机制，具有 Lazy 策略的按需分页机制。

在页面懒分配（Lazy allocation of pages）技术中，内存分配并不会立即发生，而是在需要使用内存时才分配，这样可以节省系统的资源并提高程序的性能。

实现页面懒分配的思路是：当调用 sbrk 时不分配实际的页面，而是仅仅增大堆的大小，当实际访问页面时，就会触发缺页异常，此时再申请一个页面并映射到页表中，这时再次执行触发缺页异常的代码就可以正常读写内存了。

注释掉 growproc()函数，增加堆的 size。

```
uint64 sys_sbrk(int n)
{
     uint64 addr;
     struct proc *p = curr_proc();
     addr = p->program_brk;
     int heap_size = addr + n - p->heap_bottom;
     if(heap_size < 0){
             errorf("out of heap_bottom\n");
             return -1;
     }
     else{
             p->program_brk += n; //增加堆的size，但不实际分配内存
             if(n < 0){
                     printf("uvmdealloc\n");
                     uvmdealloc(p->pagetable, addr, addr + n); //如果减少内存则调用内存释放函数
             }
     }
     //if(growproc(n) < 0) //注释掉growproc()函数，不实际分配内存
     //        return -1;
     return addr;
}
```

syscall.c

```
    switch (cause) {
    case UserEnvCall:
            trapframe->epc += 4;
            syscall();
            break;
    case StorePageFault: // 读缺页错误
    case LoadPageFault:  // 写缺页错误
            {
                    uint64 addr = r_stval(); // 获取发生缺页错误的地址
                    if(lazy_alloc(addr) < 0){ // 调用页面懒分配函数
                            errorf("lazy_aolloc() failed!\n");
                            exit(-2);
                    }
                    break;
            }
```

修改 loader.c。

```
struct proc *p = curr_proc();
// 通过两个if判断发生缺页错误的地址是否在堆的范围内，不在则返回
if (addr >= p->program_brk) {
        errorf("lazy_alloc: access invalid address");
        return -1;
}
if (addr < p->heap_bottom) {
        errorf("lazy_alloc: access address below stack");
        return -2;
}
uint64 va = PGROUNDDOWN(addr);
char* mem = kalloc(); // 调用kalloc()实际分配页面
if (mem == 0) {
        errorf("lazy_alloc: kalloc failed");
        return -3;
}
```

修改 trap.c。

问答题:

1、在使用高级语言编写用户程序的时候，手动用嵌入汇编的方法随机访问一个不在当前程序逻辑地址范围内的地址，比如向该地址读/写数据。该用户程序执行的时候可能会生什么?

触发越界访问异常(内中断) ---> trap 进内核处理异常--->把程序标记为 exit，切换为 next 进程之后 os 就不会管被标记为 exit 程序的事了，就在那让他停着。

2、用户程序在运行的过程中，看到的地址是逻辑地址还是物理地址？从用户程序访问某一个地址，到实际内存中的对应单元被读/写，会经过什么样的过程，这个过程中操作系统有什么作用？（站在学过计算机组成原理的角度）

逻辑地址

这个过程需要经过页表的转换，操作系统会负责建立页表映射。实际程序执行时的具体 VA 到 PA 的转换是在 CPU 的 MMU 之中进行的。

操作系统的作用: 完成地址虚拟地址到物理地址的映射

3、覆盖、交换和虚拟存储有何异同，虚拟存储的优势和挑战体现在什么地方？

相同点: 都是为了提高内存空间的利用率

不同点: 覆盖需要程序员自己写程序来确定覆盖的顺序与位置，并且是程序内的覆盖

交换是由 os 完成的，是程序间的交换，换出旧进程，运行新进程，不过在换出的时候依赖 I/O 速度

虚拟存储是由 os 完成的，核心思想是按需分配

优势: 允许程序员编写不受物理内存大小限制的程序，程序编程简单，隔离性与安全性强，内存利用率高

挑战: 页面置换依赖于 I/O 速度、选择恰当的页面置换算法

4、什么是局部性原理？为何很多程序具有局部性？局部性原理总是正确的吗？为何局部性原理为虚拟存储提供了性能的理论保证？

(1)、什么是局部性原理？

局部性分为时间局部性与空间局部性

1. 时间局部性

如果一个数据项被访问，那么它在不久的将来可能再次被访问。这是因为循环和频繁的数据访问模式导致的。

2. 空间局部性

如果一个数据项被访问，那么存储在其附近地址的数据项很快也可能被访问。这通常是由于数据结构的顺序存储（如数组）和编程中的顺序执行造成的。

(2)、为何很多程序具有局部性

1. 这是由编程语言决定的

编程语言里面包含了大量的循环语句

2. 常用的连续存储的数据结构

比如数组之类的

(3)、局部性原理一定是正确的吗?

在大多数情况下是正确的，但也有可能不生效，比如大量的 goto 语句跳转可能会导致局部性原理失效

(4)、为何局部性原理为虚拟存储提供了性能的理论保证?

当程序的地址被访问后，其或者其旁边的地址很有可能会被再次访问，而这些地址访问通常频繁的落在一页或几页上，这就导致 OS 不需要经常的进行页面置换，以此提高了效率

5、一条 load 指令，最多导致多少次页访问异常？尝试考虑较多情况。

考虑多级页表的情况。首先指令和数据读取都可能缺页。因此指令会有 3 次访存，之后的数据读取除了页 表页缺失的 3 次访存外，最后一次还可以出现地址不对齐的异常，因此可以有 7 次异常。若考更加极端的 情况，也就是页表的每一级都是不对齐的地址并且处在两页的交界处（straddle），此时一次访存会触 发

2 次读取页面，如果这两页都缺页的话，会有更多的异常次数。

6、如果在页访问异常中断服务例程执行时，再次出现页访问异常，这时计算机系统（软件或硬件）会如何处理？这种情况可能出现吗？

我们实验的 os 在此时不支持内核的异常中断，因此此时会直接 panic 掉，并且这种情况在我们的 os 中这种情况不可能出现。像 linux 系统，也不会出现嵌套的 page fault。

7、全局和局部置换算法有何不同？分别有哪些算法？

全局置换算法会在所有任务中进行页面置换，而局部置换算法只会在该任务内部进行页面置换

全局页面置换算法：工作集置换算法，缺页率置换算法。

局部页面置换算法：最优置换算法、FIFO 置换算法、LRU 置换算法、Clock 置换算法。

8、简单描述 OPT、FIFO、LRU、Clock、LFU 的工作过程和特点（不用写太多字，简明扼要即可）

OPT：选择一个应用程序在随后最长时间内不会被访问的虚拟页进行换出。性能最佳但无法实现。

FIFO：由操作系统维护一个所有当前在内存中的虚拟页的链表，从交换区最新换入的虚拟页放在表尾，最久换入的虚拟页放在表头。当发生缺页中断时，淘汰/换出表头的虚拟页并把从交换区新换入的虚拟页加到表尾。实现简单，对页访问的局部性感知不够。

LRU：替换的是最近最少使用的虚拟页。实现相对复杂，但考虑了访存的局部性，效果接近最优置换算法。

Clock：将所有有效页放在一个环形循环列表中，指针根据页表项的使用位（0 或 1）寻找被替换的页面。考虑历史访问，性能略差于但接近 LRU。

LFU：当发生缺页中断时，替换访问次数最少的页面。只考虑访问频率，不考虑程序动态运行。

10.Clock 算法仅仅能够记录近期是否访问过这一信息，对于访问的频

度几乎没有记录，如何改进这一点？

如果想要改进这一点，可以将 Clock 算法和计数器结合使用。具体做法是为每个页面设置一个计数器，记录页面在一段时间内的访问次数，然后在置换页面时，既考虑页面最近的访问时间，也考虑其访问频度。当待缓存对象在缓存中时，把其计数器的值加 1。同时，指针指向该对象的下一个对象。若不在缓存中时，检查指针指向对象的计数器。如果是 0，则用待缓存对象替换该对象；否则，把计数器的值减 1，指针指向下一个对象。如此直到淘汰一个对象为止。由于计数器的值允许大于 1，所以指针可能循环多遍才淘汰一个对象。

11.哪些算法有 belady 现象？思考 belady 现象的成因，尝试给出说明

OPT 和 LRU 等为何没有 belady 现象。

FIFO 算法、Clock 算法。

页面调度算法可分为堆栈式和非堆栈式，LRU、LFU、OPT 均为堆栈类算法，FIFO。

Clock 为非堆栈类算法，只有非堆栈类才会出现 Belady 现象。

14.请问一个任务处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存(给出数量级即可)?

大致占用 10G/512=20M 内存。

实验题：

实践作业：

重写 sys_get_time：

```
pub fn sys_get_time() -> isize {
    syscall(SYSCALL_GET_TIME, [0, 0, 0])
}
```

mmap 和 munmap 匿名映射：

```
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;
const SYSCALL_GET_TIME: usize = 169;
const SYSCALL_SBRK: usize = 214;
const SYSCALL_MMAP: usize = 222;//重写mmap
const SYSCALL_MUNMAP: usize = 215;//重写mumap
```

```
pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        SYSCALL_WRITE => sys_write(args[0], args[1] as *const u8, args[2]),
        SYSCALL_EXIT => sys_exit(args[0] as i32),
        SYSCALL_YIELD => sys_yield(),
        SYSCALL_GET_TIME => sys_get_time(),
        SYSCALL_SBRK => sys_sbrk(args[0] as i32),
        SYSCALL_MMAP => sys_mmap(args[0] as usize,args[1] as usize,args[2] as usize), // fn sys_mmap(start: usize, len:
usize, prot: usize) -> isize
        SYSCALL_MUNMAP => sys_munmap(args[0] as usize,args[1] as usize),            // fn sys_munmap(start: usize, len:
usize) -> isize
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}
```

修改 mod.rs，增加 mmap 和 munmap。

```
// MREAD | MWRITE | MEXE
pub fn sys_mmap(start: usize,len:usize,port:usize) -> isize{  // // fn sys_mmap(start: usize, len: usize, prot: usize) ->
isize
    kernel_mmap(start,len,port) as isize
}

pub fn sys_munmap(start: usize,len:usize) ->isize{                  // //// fn sys_munmap(start: usize, len: usize) ->
isize
    kernel_munmap(start,len) as isize
}
```

修改 process。

```
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;
const SYSCALL_GET_TIME: usize = 169;
const SYSCALL_SBRK: usize = 214;
const SYSCALL_MMAP: usize = 222;//mmap
const SYSCALL_MUNMAP: usize = 215;//mumap
```

```rust
pub fn sys_sbrk(size: i32) -> isize {
    syscall(SYSCALL_SBRK, [size as usize, 0, 0])
}

pub fn sys_mmap(start : usize,len : usize,port : usize) -> isize{
    syscall(SYSCALL_MMAP, [start, len, port])
}

pub fn sys_munmap(start : usize,len:usize) -> isize{
    syscall(SYSCALL_MUNMAP, [start, len, 0])
}
```

修改 syscall.rs。

```rust
/// 要与kernel 的MapPermission一致
pub const MREAD:usize = 1;
pub const MWRITE:usize = 2;
pub const MEXEC :usize = 4;

pub fn mmap(start : usize,len : usize,port : usize)  -> isize{
    sys_mmap(start, len, port)
}

pub fn munmap(start : usize,len : usize)  -> isize{
    sys_munmap(start, len)
}
```

修改 lib.rs。

```rust
// test1 测试能不能正常的mmap与munmap
    // let start = 0x40000000;     // 基地址是0x40000000
    // let len = 0x10000;          // 长度是10页
    // let port = MREAD | MWRITE |MEXEC;

    // let ret = mmap(start, len, port);     //分配一个可读可写可执行的页面
    // if ret == 0{
    //     println!("no , mmap error");
    // }


    // let ret = munmap(start, len);
    // if ret == 0{
    //     println!("no , munmap error");
    // }
```

修改测试文件。