**Lab 2 system calls**

1. **Eliminate allocation from sbrk() (easy)** 
   1. **实验目的**

从sys\_sbrk()系统调用中删除页面分配

* 1. **实验步骤**
     1. 修改函数

kernel/sysproc.c

uint64

sys\_sbrk(void)

{

int addr;

int n;

if(argint(0, &n) < 0)

return -1;

addr = myproc()->sz;

// 如果要扩展的堆大小超出了可寻址的最大虚拟地址范围，或者小于等于 0，直接返回当前堆大小

if (addr + n >= MAXVA || addr + n <= 0)

return addr;

myproc()->sz += n; // 虚假地增加size而不物理分配,真正使用该地址时触发缺页中断进行实际页分配

// 如果n大于0,则不做操作

// 如果n小于0,则需要释放堆内存

if(n < 0)

uvmdealloc(p->pagetable, addr, p->sz);

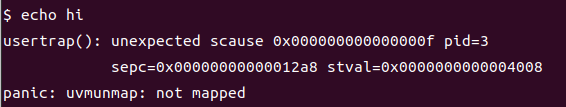
//if(growproc(n) < 0) //不实际物理分配

// return -1;

return addr;

}

* + 1. make测试，启动xv6并键入echo hi，出现缺页中断。



* 1. **实验中遇到的问题和解决方法**

问题：实验较为简单，未出现错误。

* 1. **实验心得**

该实验修改sbrk函数，使其不直接调用growproc分配物理页，而是只让p->sz增加，等到出现缺页中断时再分配真正的物理页。

1. **Lazy allocation (moderate)**
   1. **实验目的**

修改中断处理相关代码，响应来自用户控件的页面错误，通过在故障地址处映射新分配的物理内存，然后返回到用户空间，让进程继续执行。

* 1. **实验步骤**
     1. 修改usertrap函数内的处理逻辑

kernel/trap.c

关于va需要用PGROUNDDOWN()向下取整：终止页last是通过va计算出来的, 若va未向下取整, 则last==a+1而非last==a, 这会导致多映射一页, 导致页面释放时，多映射的一页未被取消映射。

void

usertrap(void)

{

......

} else if (r\_scause() == 13 || r\_scause() == 15) {//13/15对应发生读/写缺页中断

//printf("page fault trap: signal %d at address %p\n", r\_scause(), r\_stval()); // 打印中断类型以及引发中断的虚拟地址

uint64 va = r\_stval(); // 错误的虚拟地址被保存在此处

uint64 pa = (uint64) kalloc(); // 物理地址

if (pa == 0) {// 分配物理地址失败

p->killed = 1;

} else if (va >= p->sz || va <= PGROUNDDOWN(p->trapframe->sp)) {

kfree((void\*)pa);

p->killed = 1;

} else {

va = PGROUNDDOWN(va);

memset((void \*)pa, 0, PGSIZE); // 将分配的物理页面清零

// 将物理页面映射到虚拟地址，并设置页面标志为可写、可执行、可读、用户模式

if (mappages(p->pagetable, va, PGSIZE, pa, PTE\_W|PTE\_R|PTE\_U) != 0) {//若分配失败

kfree((void \*)pa); // 解除分配的物理页面

p->killed = 1; // 终止进程

}

}

} else if((which\_dev = devintr()) != 0){

......

}

* + 1. 修改部分原本会引起panic，但在懒分配模式下需要直接跳过的代码

kernel/vm.c/ void

uvmunmap(pagetable\_t pagetable, uint64 va, uint64 npages, int do\_free)

虚拟地址尚未映射时，原本应引起panic，现在可以直接跳过（即未分配就无需释放）

......

if((pte = walk(pagetable, a, 0)) == 0) // 使用walk获取页表项,若获取失败则说明该虚拟地址尚未映射,跳过

continue;

//panic("uvmunmap: walk");

if((\*pte & PTE\_V) == 0) // PTE\_V判断虚拟地址对应的也表象是否已被映射,若没有被映射自然也不需要解除映射

continue;

//panic("uvmunmap: not mapped");

uint64 sysinfo(struct sysinfo\*);

kernel/vm.c/int

uvmcopy(pagetable\_t old, pagetable\_t new, uint64 sz)

进行类似uvmunmap中的跳过操作。

if((pte = walk(old, i, 0)) == 0) //与uvmunmap类似

continue;

//panic("uvmcopy: pte should exist");

if((\*pte & PTE\_V) == 0)

continue;

//panic("uvmcopy: page not present");

kernel/vm.c/ uint64

walkaddr(pagetable\_t pagetable, uint64 va)

walkaddr根据给定的虚拟地址和页表，找到对应的物理地址。若页表项不存在或虚拟页没有被映射，则实行懒分配操作。

uint64

walkaddr(pagetable\_t pagetable, uint64 va)

{

pte\_t \*pte;

uint64 pa;

if(va >= MAXVA)

return 0;

pte = walk(pagetable, va, 0);

//if(pte == 0)

// return 0;

//if((\*pte & PTE\_V) == 0)

// return 0;

// 如果页表项不存在或者虚拟页面没有被映射，执行下面的懒分配操作

if (pte == 0 || (\*pte & PTE\_V) == 0) {

struct proc \*p = myproc();

// 检查虚拟地址是否在进程的有效范围内

if(va >= p->sz || va < PGROUNDUP(p->trapframe->sp))

return 0;

// 分配物理内存

pa = (uint64)kalloc();

if (pa == 0)

return 0;

// 在页表中建立映射

if (mappages(p->pagetable, va, PGSIZE, pa, PTE\_W|PTE\_R|PTE\_U|PTE\_X) != 0) {

kfree((void\*)pa);

return 0;

}

return pa; // 返回物理地址

}

if((\*pte & PTE\_U) == 0)

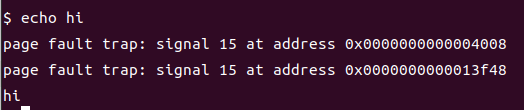
return 0;

pa = PTE2PA(\*pte);

return pa;

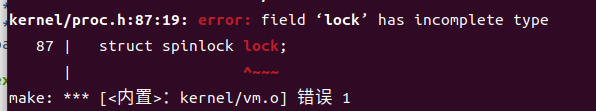
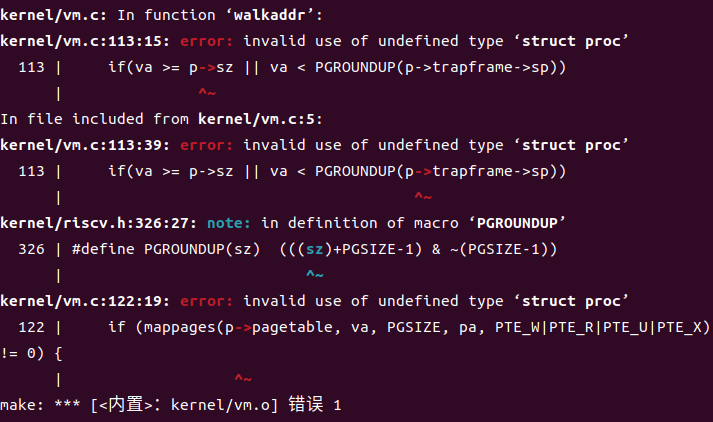
}

* + 1. make测试



* 1. **实验中遇到的问题和解决方法**

问题：



解决方法：头文件包括错误导致的，仅需在vm.c中#inlucde "proc.h"与"spinlock.h"即可

* 1. **实验心得**

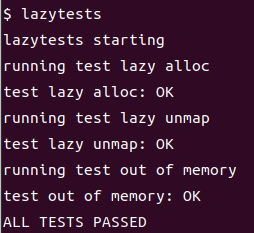
实现了懒分配，因为概念较为清晰感觉该实验较为简单。

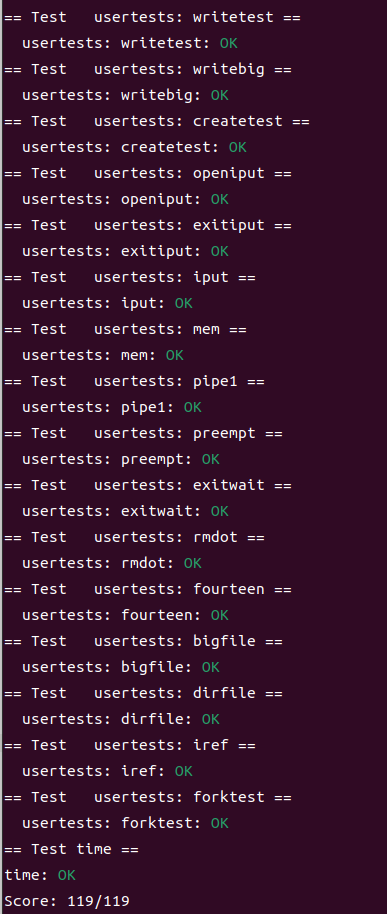
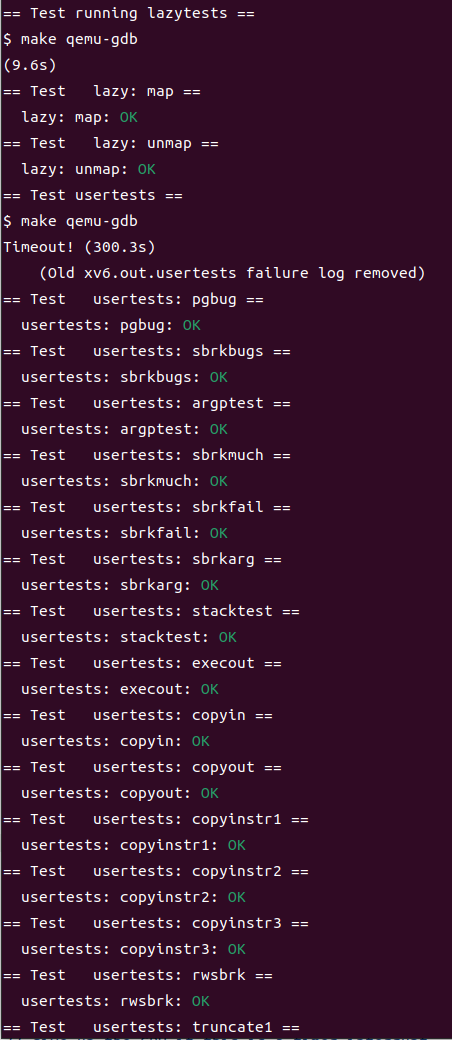
在后续的mmap实验中有再次用到懒分配相关内容，值得复习。

1. **Lazy allocation (moderate)**
   1. **实验目的**

需要通过懒分配测试用例。

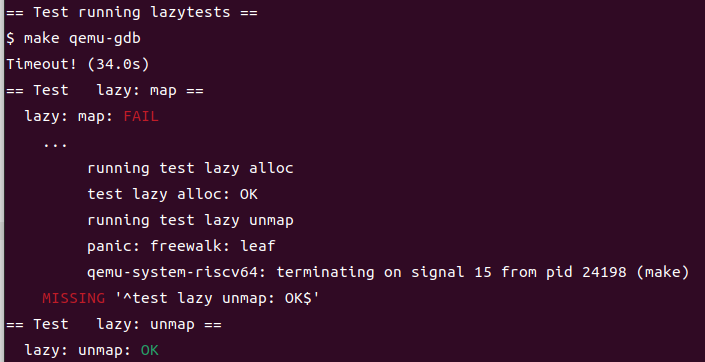
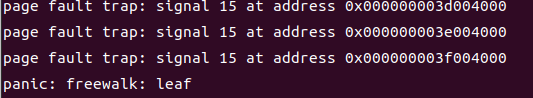
* 1. **实验步骤**
     1. make测试





* 1. **实验中遇到的问题和解决方法**

问题：执行echo hi正常，而



一开始我直接使用了 PGROUNDDOWN 宏来将va虚拟地址进行页面对齐操作，导致在else if (va >= p->sz || va <= PGROUNDDOWN(p->trapframe->sp)) 这一步中，会出现va在对比之前就已经对齐过了，可能会影响判断结果。

解决方法：先将虚拟地址保存到 va 变量，然后根据该虚拟地址进行页面对齐操作。这样，在分配和映射之前，你保持了原始的虚拟地址，从而正确地将物理页面映射到了对应的虚拟地址上。

* 1. **实验心得**

通过懒分配，更加熟悉了内存交互用的各种函数，加深了对虚拟内存和物理内存及其之间交互的理解。