**Lab 10 mmap(hard)**

* 1. **实验目的**

理解内存映射的概念和原理：针对一个文件, 将其映射到用户态的一个地址中去, 使得用户可以直接通过对于这个地址的读/写来实现文件的操作，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。

掌握在操作系统中实现内存映射的基本方法。

学习如何修改操作系统内核以添加新的系统调用。

* 1. **实验步骤**
     1. 根据实验手册提示，添加系统调用

Makefile(UPROGS): 编译声明

......

$U/\_zombie\

$U/\_mmaptest\

......

user/user.h:系统调用函数声明

......

void\* mmap(void \*addr, int length, int prot, int flags, int fd, uint offset);

int munmap(void \*addr, int length);

user/usys.pl:entry项

......

entry("mmap");

entry("munmap");

kernel/syscall.h:系统调用码

#define SYS\_mmap 22

#define SYS\_munmap 23

kernel/syscall.c:系统调用函数引用

...

extern uint64 sys\_mmap(void);

extern uint64 sys\_munmap(void);

...

[SYS\_mmap] sys\_mmap,

[SYS\_munmap] sys\_munmap,

}

* + 1. 根据实验手册提示，添加vma(Virtual Memory Area)结构体定义并修改proc内容.(采用实验手册建议，proc结构体中的vma数组长度16)

在每个进程结构体中开启vma虚拟内存区。对应此处结构体定义了dirty这一标志以实现实验手册中指出的本实验不必需的dirty读写相关操作。

kernel/proc.h

#define VMASIZE 16 //根据实验手册定义VMA结构体,数组长度为16

......

//vma(Virtual Memory Area)结构体

struct vma {

int used;

uint64 addr; // 映射地址

int length; // 映射长度

int prot; // 权限

int flags; // 标志

int fd; // 文件描述符

int offset; // 偏移量

int dirty; // 是否修改过标志

struct file \*file;

};

struct proc {

......

struct vma vma[VMASIZE];

//在proc中加入文件系统的vma结构体数组

};

* + 1. 实现mmap和munmap函数

kernel/sysfile.c/uint64 sys\_mmap(void)

变量设置

uint64 addr;

int length, prot, flags, fd, offset;// 映射参数,目标虚拟地址,映射长度,权限标志,标志,文件描述副,起始偏移量

struct file \*file;

struct proc \*p = myproc();

获取从用户空间传入的参数，包括目标虚拟地址 addr、映射长度 length、权限标志 prot、标志 flags、文件描述符 fd、起始偏移量 offset。检查传入的参数是否合法，包括从用户态获取传入的参数，并检查文件的权限是否与映射的权限匹配。

// 检查从用户态获取传入的参数

if (argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||

argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset))

return -1; // 参数传递错误，返回 -1

// 检查文件的权限是否与映射的权限匹配

if (!file->writable && (prot & PROT\_WRITE) && flags == MAP\_SHARED)

return -1; // 权限不匹配，返回 -1

对映射的长度 length 进行向上对齐，以满足内存分页的要求。

// 向上对齐长度

length = PGROUNDUP(length);

检查进程的地址空间是否足够，即当前进程的大小 p->sz 是否超过了最大虚拟地址 MAXVA 减去映射长度。若地址空间足够，则遍历当前进程的 VMA 数组，寻找一个未使用的 VMA 插槽。如果遍历完 VMA 数组没有找到可用的插槽，则返回 -1 表示映射失败；如果找到，则将相关信息填入该 VMA 结构：设置虚拟地址、映射长度、权限标志、映射标志、文件描述符、文件指针、起始偏移等。

// 检查进程的地址空间是否足够

if (p->sz > MAXVA - length)

return -1; // 地址空间不足，返回 -1

// 遍历 VMA 数组，找到一个未使用的 VMA

for (int i = 0; i < VMASIZE; ++i) {

if (p->vma[i].used == 0) {

p->vma[i].used = 1; // 标记 VMA 插槽为已使用

p->vma[i].addr = p->sz; // 设置映射的虚拟地址

p->vma[i].length = length; // 设置映射的长度

p->vma[i].prot = prot; // 设置权限标志

p->vma[i].flags = flags; // 设置映射标志

p->vma[i].fd = fd; // 设置文件描述符

p->vma[i].file = file; // 设置映射的文件

p->vma[i].offset = offset; // 设置映射的起始偏移

filedup(file); // 增加文件的引用计数，以保证文件在映射期间不会被关闭

p->sz += length; // 虚拟地增加进程大小，但没有实际分配物理页

return p->vma[i].addr; // 返回分配的虚拟地址

}

}

根据实验手册提示，通过 filedup 增加文件的引用计数，以保证文件在映射期间不会被关闭。

虚拟地增加进程大小 p->sz，表示在进程的虚拟地址空间中分配了一段地址。

返回分配的虚拟地址。

kernel/sysfile.c/uint64 sys\_munmap(void)

变量设置：

uint64 addr;

int length;

struct proc \*p = myproc();

struct vma \*vma = 0;

解析并检查传入的参数，包括要取消映射的起始虚拟地址 addr 和取消映射的长度 length。

// 解析参数

if(argaddr(0, &addr) || argint(1, &length))

return -1;

类似mmap，对虚拟地址和长度进行向下和向上对齐，以满足内存分页的要求。

// 将虚拟地址和长度进行对齐

addr = PGROUNDDOWN(addr);

length = PGROUNDUP(length);

遍历当前进程的 VMA 数组，查找包含取消映射地址的 VMA并操作。

// 查找对应的 VMA

for(int i = 0; i < VMASIZE; i++) {

if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length) {

vma = &p->vma[i];

break;

}

}

// 如果找不到对应的 VMA，则返回

if(vma == 0)

return 0;

// 如果要取消映射的地址是 VMA 的起始地址

if(vma->addr == addr) {

// 更新 VMA 的地址和长度

vma->addr += length;

vma->length -= length;

// 如果是共享映射，将修改的内容写回文件

if(vma->flags & MAP\_SHARED){

if (vma->dirty) {

// 页面被修改过，将区域的数据写回文件

if (filewrite(vma->file, addr, length) < 0)

printf("munmap: filewrite < 0\n");

vma->dirty = 0; // 写入文件后，重置 dirty 标志位为 0

}

}

// 取消映射并释放对应的物理内存

uvmunmap(p->pagetable, addr, length/PGSIZE, 1);

// 如果 VMA 长度变为 0，则关闭文件并标记为未使用

if(vma->length == 0) {

fileclose(vma->file);

vma->used = 0;

}

}

如果找不到对应的 VMA，表示该虚拟地址没有映射，直接返回 0。

如果找到对应的 VMA，检查是否要取消映射的地址是 VMA 的起始地址。

如果地址是 VMA 的起始地址，则更新 VMA 的地址和长度，即将取消映射的区域从 VMA 中移除。

如果 VMA 是共享映射且内容被修改（标志 dirty = 1），则将修改的内容写回文件中。

通过 uvmunmap 函数取消映射，并释放对应的物理内存。

// 取消映射并释放对应的物理内存

uvmunmap(p->pagetable, addr, length/PGSIZE, 1);

如果 VMA 长度变为 0，则关闭文件并将 VMA 标记为未使用，释放资源。

// 如果 VMA 长度变为 0，则关闭文件并标记为未使用

if(vma->length == 0) {

fileclose(vma->file);

vma->used = 0;

}

首先解析传入的参数，即要取消映射的虚拟地址和长度，对虚拟地址和长度进行对齐处理。接着，它遍历进程的 VMA 数组，寻找与要取消映射的虚拟地址相关联的 VMA。如果找到了对应的 VMA，检查是否是从 VMA 的起始地址开始取消映射。如果是，则更新 VMA 的地址和长度，检查dirty标志位以确定是否需要将修改的内容写回文件（如果是共享映射），并在需要的情况下释放物理内存。

如果 VMA 的长度变为 0，说明整个 VMA 都被取消映射了，关闭文件并将 VMA 标记为未使用。最后，函数返回 0 表示操作成功。总之，这个函数的逻辑是取消进程的虚拟内存映射，并进行必要的操作，如文件写回和资源释放。

uint64

sys\_munmap(void)

{

uint64 addr;

int length;

struct proc \*p = myproc();

struct vma \*vma = 0;

// 解析参数

if(argaddr(0, &addr) || argint(1, &length))

return -1;

// 将虚拟地址和长度进行对齐

addr = PGROUNDDOWN(addr);

length = PGROUNDUP(length);

// 查找对应的 VMA

for(int i = 0; i < VMASIZE; i++) {

if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length) {

vma = &p->vma[i];

break;

}

}

// 如果找不到对应的 VMA，则返回

if(vma == 0)

return 0;

// 如果要取消映射的地址是 VMA 的起始地址

if(vma->addr == addr) {

// 更新 VMA 的地址和长度

vma->addr += length;

vma->length -= length;

// 如果是共享映射，将修改的内容写回文件

if(vma->flags & MAP\_SHARED){

if (vma->dirty) {

// 页面被修改过，将区域的数据写回文件

if (filewrite(vma->file, addr, length) < 0)

printf("munmap: filewrite < 0\n");

vma->dirty = 0; // 写入文件后，重置 dirty 标志位为 0

}

}

// 取消映射并释放对应的物理内存

uvmunmap(p->pagetable, addr, length/PGSIZE, 1);

// 如果 VMA 长度变为 0，则关闭文件并标记为未使用

if(vma->length == 0) {

fileclose(vma->file);

vma->used = 0;

}

}

return 0;

}

* + 1. 中断处理

实验手册指定以懒加载方式分配映射的内存资源，因此在读取或写入相应的虚拟地址时，会存在地址未映射的中断情况。此时需要将物理地址上的数据读到对应的虚拟地址中，然后重新进行读取或写入操作。需要检查地址是否合法。

kernel/trap.c

...

//新增头文件(file结构体定义与权限相关宏定义)

#include "fcntl.h"

#include "fs.h"

#include "sleeplock.h"

#include "file.h"

...

kernel/trap.c/void usertrap(void)

在原有的usertrap处理中断逻辑上，增加else if检查是否是读写造成的缺页中断。如果是缺页中断，说明发生了虚拟地址到物理地址的映射缺失，需要分配物理内存并加载数据到虚拟地址。

......

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

// ok

} else if (r\_scause() == 13 || r\_scause() == 15) {

当发生缺页中断时，通过 r\_stval() 函数获取造成缺页中断的虚拟地址。

uint64 va = r\_stval(); // 获取造成缺页中断的虚拟地址

检查虚拟地址的合法性，包括是否在进程的地址空间范围内，以及是否合理与栈的地址重叠，如果不合法，标记进程为被终止状态。

// 检查虚拟地址的合法性

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

p->killed = 1;

else {

struct vma \*vma = 0;

如果虚拟地址合法，代码遍历进程的 VMA 数组，查找与虚拟地址匹配的 VMA，以确定映射的文件和偏移量。

// 在进程的 VMA 数组中查找匹配的 VMA

for (int i = 0; i < VMASIZE; i++) {

if (p->vma[i].used == 1 && va >= p->vma[i].addr && va < p->vma[i].addr + p->vma[i].length) {

vma = &p->vma[i];

break;

}

}

如果找到匹配的 VMA，代码对虚拟地址进行页对齐，计算出在文件中的偏移量，并分配一个物理页。

if (vma) {

va = PGROUNDDOWN(va); // 对虚拟地址进行对齐

uint64 offset = va - vma->addr; // 计算在文件中的偏移量

uint64 mem = (uint64)kalloc(); // 分配一个物理页

分配物理页过程中，如果物理页分配失败，则标记进程终止。如果物理页分配成功，则清空物理页的内容，并在进程的 VMA 对应文件的锁下读取文件内容到物理页。

if (mem == 0) {

// 分配物理页失败，将进程标记为被终止

p->killed = 1;

} else {

memset((void\*)mem, 0, PGSIZE); // 清空物理页内容

ilock(vma->file->ip); // 获取文件的锁

readi(vma->file->ip, 0, mem, offset, PGSIZE); // 从文件中读取内容到物理页

iunlock(vma->file->ip); // 释放文件的锁

为页表项设置标志，根据 VMA 的权限属性，以及是否可读、可写和可执行调用 mappages 函数，将物理页映射到虚拟地址，并设置页表项的标志。

int PTEflag = PTE\_U; // 设置页表项的标志

if (vma->prot & PROT\_READ) PTEflag |= PTE\_R;

if (vma->prot & PROT\_WRITE) PTEflag |= PTE\_W;

if (vma->prot & PROT\_EXEC) PTEflag |= PTE\_X;

// 将物理页映射到虚拟地址，并设置页表项的标志

if (mappages(p->pagetable, va, PGSIZE, mem, PTEflag) != 0) {

kfree((void\*)mem); // 映射失败，释放物理页

p->killed = 1; // 标记进程为被终止状态

}

vma->dirty = 1; // 页面被修改，设置 dirty 标志位为 1

}

}

如果映射失败，释放之前分配的物理页，标记进程为被终止状态。

如果映射成功，将 VMA 的 dirty 标志位设置为 1，表示该页面已被修改。

kernel/vm.c

部分地址并未映射(有效位为0),则直接。

void

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

{

...

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

continue;

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

//在执行虚拟内存映射的解除操作时，如果当前页表项的 PTE\_V（有效位）为 0，表示该虚拟地址没有进行有效映射，直接跳过处理即可。这可以避免处理不必要的操作。

...

}

int

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

{

...

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

continue;

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

//在执行虚拟内存映射的解除操作时，如果当前页表项的 PTE\_V（有效位）为 0，表示该虚拟地址没有进行有效映射，直接跳过处理即可。这可以避免处理不必要的操作。

...

}

* + 1. fork和exit函数处理

在进程创建和退出时，需要复制和清空相应的文件映射：

kernel/proc.c

int

fork(void){

...

//

// 复制父进程的文件映射

for(int i = 0; i < VMASIZE; i++) {

if(p->vma[i].used) {

// 复制 VMA 数据到新进程

memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));

// 增加文件引用计数，以防止文件被关闭

filedup(p->vma[i].file);

}

}

//

...

}

void

exit(int status){

...

// 清理当前进程的文件映射

for(int i = 0; i < VMASIZE; i++) {

if(p->vma[i].used) {

// 如果映射标志为 MAP\_SHARED，则将内容写回文件

if(p->vma[i].flags & MAP\_SHARED)

filewrite(p->vma[i].file, p->vma[i].addr, p->vma[i].length);

// 关闭文件

fileclose(p->vma[i].file);

// 解除虚拟内存映射并释放物理页

uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length/PGSIZE, 1);

// 将 VMA 标记为未使用

p->vma[i].used = 0;

}

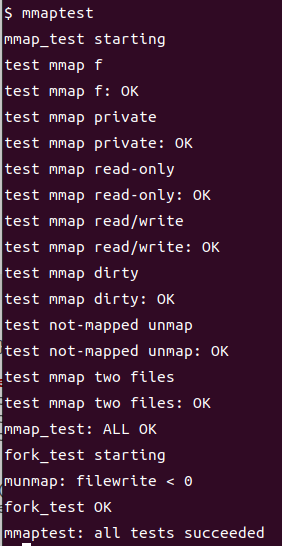
} begin\_op();

...

}

* + 1. 测试，分数生成及提交

同其他实验，在实验目录下使用make qemu进行测试、用git commit与make grade指令生成分数、提交至远程仓库。

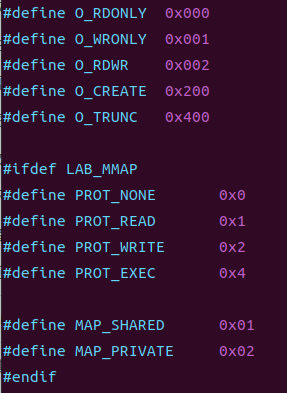
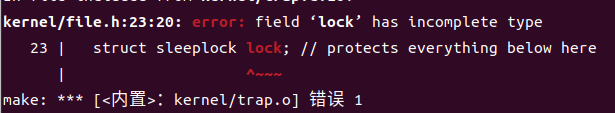
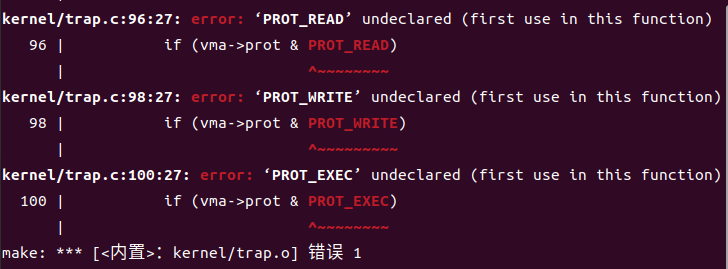


mmaptest与grade运行结果。

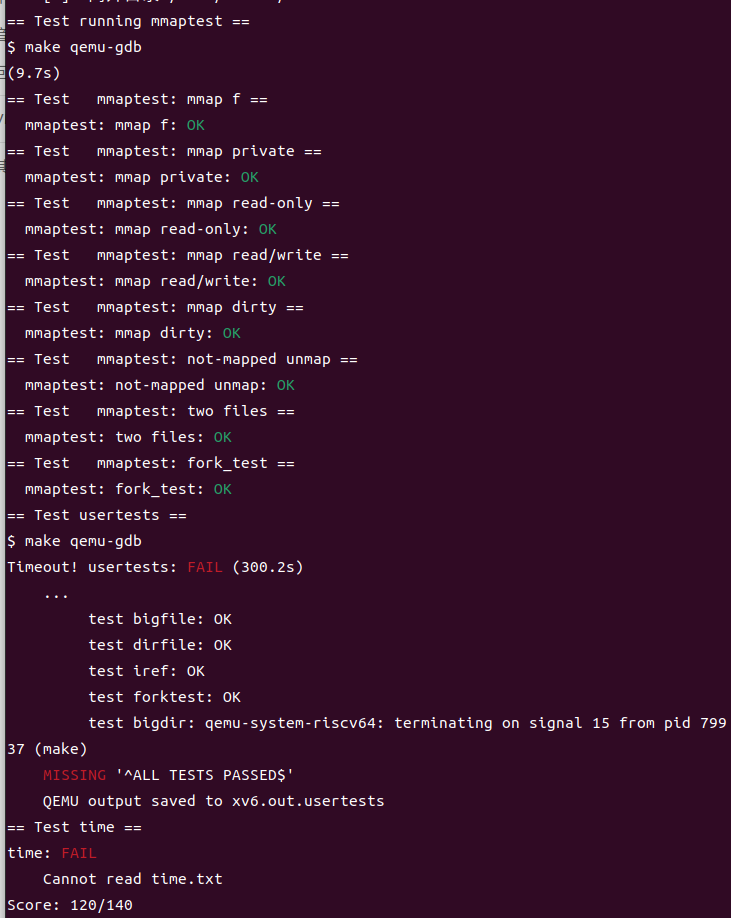
* 1. **实验中遇到的问题和解决方法**

问题：头文件相关错误：trap.c文件中，mmap函数定义时权限相关宏定义声明不明确；trap.c文件中若包含file.h则提示file.h中sleeplock定义不明确，一开始怀疑是file.h和defs.h中重复包含sleeplock.h的问题，后来include一个sleeplock.h解决问题。

解决方法：查看实验手册及系统文件，发现权限定义在kernel/fcntl.h头文件中，在trap.h中添加对应头文件包含；



问题：部分测试内容timeout超时。



解决方式：同lab9-filesystem实验，增大grade-lab-mmap文件中对应限时数即可。300->3000

grade-lab-mmap

@test(19, "usertests")

def test\_usertests():

r.run\_qemu(shell\_script([

'usertests'

]), timeout=3000)

r.match('^ALL TESTS PASSED$')

* 1. **实验心得**

mmap实验是综合性很强的一个实验，结合了前面包括懒分配在内的许多内容，我在进行实验时也出现了不少错误。该实验也是一个很好的机会，通过自主实现mmap 和 munmap让我更深入地了解操作系统内存管理和文件系统的交互，理解在内核中如何管理进程的虚拟地址空间、如何将文件映射到内存、如何处理页面错误等。