Lab 10 Memory Map

目录

[mmap (hard) 1](#_Toc174491710)

[1.1 实验目的 1](#_Toc174491711)

[1.2 实验步骤 1](#_Toc174491712)

[1.3 实验结果 11](#_Toc174491713)

[1.4 实验中遇到的问题和解决方法 11](#_Toc174491714)

[1.5 实验心得 12](#_Toc174491715)

[测试结果 13](#_Toc174491716)

mmap (hard)

* 1. 实验目的
* 学习 mmap 和 munmap 的接口和参数
* 实现 lazy allocation 优化内存映射文件的性能
* 维护每个进程的内存映射信息
* 实现 mmap 和 munmap 系统调用
* 处理 mmap 相关的 page fault
* 学习文件和内存的映射机制
* 处理进程退出和 fork 的内存映射维护
  1. 实验步骤
* 将 \_mmaptest 添加到 Makefile 的 UPROGS 中：
* 在 user/user.h 中添加两个系统调用函数的声明：

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

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

* 在 user/usys.pl 中添加调用入口：

entry("mmap");

entry("munmap");

* 在  kernel/syscall.h  中添加系统调用号：

#define SYS\_mmap 22

#define SYS\_munmap 23

* 在  kernel/syscall.c 中添加系统调用

extern uint64 sys\_mmap(void);

extern uint64 sys\_munmap(void);

static uint64(\*syscalls[])(void) = {

...

[SYS\_mmap] sys\_mmap,

[SYS\_munmap] sys\_munmap,

};

* 在 kernel/proc.h 中定义一个对应于虚拟内存区域（VMA）的结构，记录地址、长度、权限、文件等信息。在进程结构中添加一个固定大小的 VMA 数组（如 16 个），用于记录映射区域。

#define VMASIZE 16

struct VMA {

int active;

uint64 addr;

int length;

int prot;

int flags;

int fd;

int offset;

struct file\* fp;

};

// Per-process state

struct proc {

struct spinlock lock;

// p->lock must be held when using these:

enum procstate state; // Process state

void\* chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

int xstate; // Exit status to be returned to parent's wait

int pid; // Process ID

// wait\_lock must be held when using this:

struct proc\* parent; // Parent process

// these are private to the process, so p->lock need not be held.

uint64 kstack; // Virtual address of kernel stack

uint64 sz; // Size of process memory (bytes)

pagetable\_t pagetable; // User page table

struct trapframe\* trapframe; // data page for trampoline.S

struct context context; // swtch() here to run process

struct file\* ofile[NOFILE]; // Open files

struct inode\* cwd; // Current directory

char name[16]; // Process name (debugging)

struct VMA vma[VMASIZE];

};

* 在 kernel/sysfile.c 中实现函数 sys\_mmap() ，将文件映射到进程地址空间

uint64 sys\_mmap(void)

{

int length, prot, flags, fd, offset;

uint64 addr;

struct file\* fp;

struct proc\* p = myproc();

// 处理传入参数

argaddr(0, &addr); // 获取 addr 参数

argint(1, &length); // 获取 length 参数

argint(2, &prot); // 获取 prot 参数

argint(3, &flags); // 获取 flags 参数

argfd(4, &fd, &fp); // 获取文件描述符 fd 和文件指针 fp

argint(5, &offset); // 获取 offset 参数

// 参数检查

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

// 如果文件不可写，并且内存保护需要写权限且标志为共享映射，则返回错误

return -1;

}

// 将 length 向上取整到页面大小

length = PGROUNDUP(length);

if (p->sz + length > MAXVA) {

// 如果映射后进程地址空间超出最大虚拟地址，则返回错误

return -1;

}

// 遍历进程的虚拟内存区域（VMA）数组，找到一个未使用的区域

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

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

// 标记该区域为已使用

p->vma[i].active = 1;

// 直接映射到进程当前大小（sz）对应的虚拟地址

p->vma[i].addr = p->sz;

p->vma[i].length = length;

p->vma[i].prot = prot;

p->vma[i].flags = flags;

p->vma[i].fd = fd;

p->vma[i].fp = fp;

p->vma[i].offset = offset;

// 增加文件引用计数，以防止文件被关闭

filedup(fp);

// 更新进程的大小

p->sz += length;

// 返回映射的虚拟地址

return p->vma[i].addr;

}

}

// 如果没有找到可用的 VMA 区域，返回错误

return -1;

}

* 在 kernel/trap.c 中修改 usertrap( ) 函数以实现了缺页异常处理中的物理页懒分配，确保在需要时为虚拟地址分配物理页面并读取文件内容。文件上面包含头文件 #include "fcntl.h"

// handle an interrupt, exception, or system call from user space.

// called from trampoline.S

//

void

usertrap(void)

{

int which\_dev = 0;

if((r\_sstatus() & SSTATUS\_SPP) != 0)

panic("usertrap: not from user mode");

// send interrupts and exceptions to kerneltrap(),

// since we're now in the kernel.

w\_stvec((uint64)kernelvec);

struct proc \*p = myproc();

// save user program counter.

p->trapframe->epc = r\_sepc();

if(r\_scause() == 8){

// system call

if(killed(p))

exit(-1);

// sepc points to the ecall instruction,

// but we want to return to the next instruction.

p->trapframe->epc += 4;

// an interrupt will change sepc, scause, and sstatus,

// so enable only now that we're done with those registers.

intr\_on();

syscall();

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

// ok

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

// supervisor interrupt exception code

uint64 scause = r\_scause();

// the faulting virtual address

uint64 addr = r\_stval();

if (addr >= MAXVA) {

setkilled(p);

goto out;

}

struct VMA\* vma = 0;

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

if (p->vma[i].active == 0) continue;

uint64 begin = (uint64)p->vma[i].addr;

uint64 end = begin + p->vma[i].length;

if (addr >= begin && addr < end) {

vma = &p->vma[i];

break;

}

}

if (vma == 0) {

printf("usertrap: the faulting virtual address %p is not in the VMA\n", addr);

setkilled(p);

goto out;

}

if (scause == 13 && vma->fp->readable == 0) {

printf("usertrap: the file is unreadable\n");

setkilled(p);

goto out;

}

if (scause == 15 && vma->fp->writable == 0) {

printf("usertrap: the file is unwritable\n");

setkilled(p);

goto out;

}

void\* pa = kalloc();

if (pa == 0) {

printf("usertrap: unable to allocate memory\n");

setkilled(p);

goto out;

}

memset(pa, 0, PGSIZE);

addr = PGROUNDDOWN(addr);

int perm = 0;

// mappages will set PTE\_V

perm |= PTE\_U;

if (vma->prot & PROT\_READ)

perm |= PTE\_R;

if (vma->prot & PROT\_WRITE)

perm |= PTE\_W;

if (vma->prot & PROT\_EXEC)

perm |= PTE\_X;

if (mappages(p->pagetable, addr, PGSIZE, (uint64)pa, perm) != 0) {

kfree(pa);

setkilled(p);

goto out;

}

ilock(vma->fp->ip);

if (readi(vma->fp->ip, 1, addr, addr - (uint64)vma->addr + vma->offset, PGSIZE) == 0) {

iunlock(vma->fp->ip);

uvmunmap(p->pagetable, addr, 1, 1);

setkilled(p);

goto out;

}

iunlock(vma->fp->ip);

}else {

printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p->pid);

printf(" sepc=%p stval=%p\n", r\_sepc(), r\_stval());

setkilled(p);

}

if(killed(p))

exit(-1);

// give up the CPU if this is a timer interrupt.

if(which\_dev == 2)

yield();

out:

if(killed(p))

exit(-1);

// give up the CPU if this is a timer interrupt.

if(which\_dev == 2)

yield();

usertrapret();

}

* 同时修改 kernel/vm.c 中的 uvmunmap( ) 和 uvmcopy( ) 函数的逻辑,正确处理可能未分配物理页的虚拟地址，从而避免错误。

// Remove npages of mappings starting from va. va must be

// page-aligned. The mappings must exist.

// Optionally free the physical memory.

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

{

uint64 a;

pte\_t \*pte;

if ((va % PGSIZE) != 0)

panic("uvmunmap: not aligned");

for (a = va; a < va + npages \* PGSIZE; a += PGSIZE) {

if ((pte = walk(pagetable, a, 0)) == 0)

panic("uvmunmap: walk");

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

continue;

if (PTE\_FLAGS(\*pte) == PTE\_V)

panic("uvmunmap: not a leaf");

if (do\_free) {

uint64 pa = PTE2PA(\*pte);

kfree((void \*)pa);

}

\*pte = 0;

}

}

// Given a parent process's page table, copy

// its memory into a child's page table.

// Copies both the page table and the

// physical memory.

// returns 0 on success, -1 on failure.

// frees any allocated pages on failure.

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

{

pte\_t \*pte;

uint64 pa, i;

uint flags;

char \*mem;

for (i = 0; i < sz; i += PGSIZE) {

if ((pte = walk(old, i, 0)) == 0)

panic("uvmcopy: pte should exist");

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

continue;

pa = PTE2PA(\*pte);

flags = PTE\_FLAGS(\*pte);

if ((mem = kalloc()) == 0)

goto err;

memmove(mem, (char \*)pa, PGSIZE);

if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0) {

kfree(mem);

goto err;

}

}

return 0;

err:

uvmunmap(new, 0, i / PGSIZE, 1);

return -1;

}

* 在 kernel/sysfile.c 中添加函数 sys\_munmap() ,实现了取消虚拟地址的映射关系，并在必要时将修改的内容写回到文件中。

uint64 sys\_munmap(void)

{

int length;

uint64 addr;

// 获取参数

argaddr(0, &addr); // 获取虚拟地址

argint(1, &length); // 获取解除映射的长度

struct proc \*p = myproc();

struct VMA \*vma = 0;

// 查找对应的 VMA

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

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

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

// 因为 addr 和 length 是页对齐的，所以只要 addr 相等，就一定是同一个 VMA

vma = &p->vma[i];

break;

}

}

}

// 如果没有找到对应的 VMA，返回 0

if (vma == 0) {

return 0;

} else {

// 更新 VMA 的地址和长度

vma->addr += length;

vma->length -= length;

// 如果是共享映射，需要将内容写回文件

if (vma->flags & MAP\_SHARED)

filewrite(vma->fp, addr, length);

// 解除映射

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

// 如果 VMA 的长度为 0，说明已经全部解除映射，需要释放资源

if (vma->length == 0) {

fileclose(vma->fp); // 关闭文件

vma->active = 0; // 标记 VMA 为未使用

}

return 0;

}

}

* 修改 kernel/proc.c 中的 fork() 函数,确保在复制页表时正确处理虚拟内存区域 (VMA) 的复制。

// fork时要复制文件内存映射信息

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

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

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

filedup(p->vma[i].fp); // refcount++

}

}

* 要注意退出进程时要将映射也一并清空，因此需要修改 kernel/exit() 函数,此外，主要在proc.c 文件上面包含头文件 #include "fcntl.h"

// exit时清空进程的文件内存映射信息

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

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

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

// 写回磁盘文件

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

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

// 取消虚拟内存映射

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

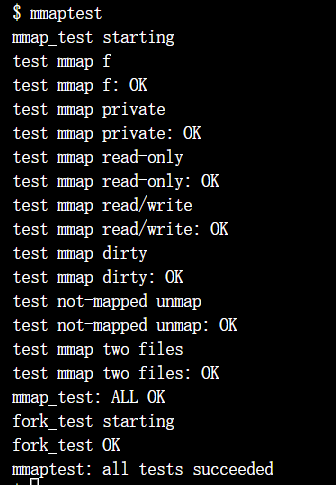
// 复位

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

}

}

* 1. 实验结果
* 运行 make qemu 后执行mmaptest ：



* 1. 实验中遇到的问题和解决方法
* 问题：

运行测试时，出现如下错误信息：panic: mappages: remap

* 解决方法：

定位问题，发现是在映射时发生的错误，对于 remap 错误，可能是由于占用了已映射的地址，经过检查发现是在设置 PTE 位时传入 flags 导致的错误，改为传入 prot 即可。

* 1. 实验心得

在这次实验中，我对操作系统的内存管理机制有了更深入的理解，尤其是在实现 mmap 和 munmap 系统调用的过程中，体会到了虚拟内存管理的复杂性与精妙之处。通过研究文件与内存的映射机制，我认识到操作系统如何高效地将文件内容映射到进程的虚拟地址空间，从而实现内存与文件的无缝交互。

在实现 mmap 相关的页面异常处理时，我深入理解了懒分配技术的应用。懒分配不仅优化了内存的使用，还显著提高了程序的执行效率。通过处理页面异常，操作系统可以在需要时才为内存页分配物理内存，这种按需分配的策略能够有效降低内存的浪费。

此外，我在实验中学习了如何维护每个进程的内存映射信息，并掌握了在进程退出和 fork 操作时正确处理内存映射的技巧。

测试结果

