Lab 10: mmap

1950787 杨鑫

实验目的：

- 了解 mmap 系统调用和 munmap 系统调用的功能和实现方法
- 修改一些数据结构，添加部分代码以实现一个简化版的 mmap 系统调用和 munmap 系统调用
- 了解文件映射等相关操作
- j进一步熟悉懒加载和文件操作

实验步骤：

首先切换分支至 mmap branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout mmap
make clean
```

a. mmap

实验目的：

mmap 和 munmap 系统调用允许 UNIX 程序对其地址空间进行详细控制。它们可用于在进程之间共享内存，将文件映射到进程地址空间，以及作为用户级页面错误方案的一部分，例如讲座中讨论的垃圾收集算法。在本实验中，您将向 xv6 添加 mmap 和 munmap，重点关注内存映射文件。

mmap 可以通过多种方式调用，但本实验只需要其与内存映射文件相关的部分功能。你可以假设 addr 总是为零，这意味着内核应该决定映射文件的虚拟地址。 mmap 返回该地址，如果失败则返回 0xffffffffffffffff。 length 是要映射的字节数；它可能与文件的长度不同。 prot 指示内存是否应该被映射为可读、可写和/或可执行；您可以假设 prot 是 PROT_READ 或 PROT_WRITE 或两者兼而有之。 flags 要么是 MAP_SHARED，这意味着对映射内存的修改应该写回文件，要么是 MAP_PRIVATE，这意味着它们不应该。您不必在标志中实现任何其他位。 fd 是要映射的文件的打开文件描述符。您可以假设偏移量为零（它是文件中映射的起点）。

如果映射同一个 MAP_SHARED 文件的进程不共享物理页面，那也没关系。

munmap(addr, length) 应该删除指定地址范围内的 mmap 映射。如果进程修改了内存并映射了 MAP_SHARED，则应首先将修改写入文件。一个 munmap 调用可能只覆盖 mmap-ed 区域的一部分，但您可以假设它会在开始、结束或整个区域取消映射（但不会在区域中间打孔）.

您应该实现足够的 mmap 和 munmap 功能以使 mmaptest 测试程序正常工作。

实验分析：

本次实验要求实现将文件映射到用户的虚拟内存区域的 mmap 和 munmap 系统调用。首先类似于第二节系统调用实验的内容，添加 mmap 和 munmap 的声明以及系统调用编号等，以便用户能够使用该系统调用；然后添加虚拟内存区域的数据结构，用于记录其地址，长度等信息，并且添加相关的初始化，分配虚拟内存空间，释放虚拟内存空间的函数；然后完成 mmap 和 munmap 系统调用的函数功能实现：mmap 取出一块空闲的虚拟内存空间，然后修改其长度，权限等信息，并且将文件指针赋给它；munmap 则找到要释放的虚拟内存，如果是被共享的要先将内容写回物理空间，然后再看是否将所有的页面均删除了，如果是的话还要将文件引用 -1，并将该虚拟空间的指针置空，若未删完，则修改其起始地址和长度。

此外，还需要修改 usertrap，因为虚拟内存区域也是采用懒分配的方式所以当产生页错误时，要及时将相应的虚拟内存放入用户的页表中。最后修改 exec 和 fork 系统调用，退出进程的时候将相应虚拟内存释放掉，产生子进程的时候将虚拟内存复制给子进程。

实验核心代码：

vma.h

```
struct vma {
    char* addr;
    uint64 len;
    struct file *file;
    char prot;
    char flag;
};
```

vma.c

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
#include "vma.h"

struct {
  struct spinlock lock;
  struct vma areas[NOFILE];
} vma_table;

void
vma_init(void)
{
  initlock(&vma_table.lock, "vma_table");
}

struct vma*
vma_alloc(void)
{
  struct vma *vmap;
```

```
  acquire(&vma_table.lock);
  for(vmap = vma_table.areas; vmap < vma_table.areas + NOFILE; vmap++){
    if(vmap->file == 0){
      release(&vma_table.lock);
      return vmap;
    }
  }
  release(&vma_table.lock);
  return 0;
}

void
vma_free(struct vma *vmap)
{
  vmap->file = 0;
}
```

sysfile.c

```
uint64 sys_mmap(void){
  int length, prot, flags, fd;
  if (argint(1, &length) < 0 || argint(2, &prot) < 0 || argint(3, &flags) < 0 ||
argint(4, &fd) < 0){
    return 0xffffffffffffffff;
  }
  struct proc *pr = myproc();
  if(!pr->ofile[fd]->readable){
    if (prot & PROT_READ)
      return 0xffffffffffffffff;
  }
  if(!pr->ofile[fd]->writable){
    if (prot & PROT_WRITE && flags==MAP_SHARED)
      return 0xffffffffffffffff;
  }

  struct vma *vmap;
  if ((vmap = vma_alloc()) == 0){
    return 0xffffffffffffffff;
  }

  acquire(&pr->lock);
  int i;
  for (i = 0; i < NOFILE; i++){
    if(pr->vmaps[i] == 0){
      pr->vmaps[i] = vmap;
      release(&pr->lock);
      break;
    }
  }
  if (i == NOFILE){
    return 0xffffffffffffffff;
  }

  uint64 sz = pr->sz;
  pr->sz += length;
  vmap->addr = (char*)sz;
```

```
  vmap->len = length;
  vmap->prot = (prot & PROT_READ) | (prot & PROT_WRITE);
  vmap->flag = flags;
  vmap->file = pr->ofile[fd];
  filedup(pr->ofile[fd]);
  return sz;
}

uint64 sys_munmap(void){
  struct proc *pr = myproc();
  int startAddr, length;

  if (argint(0, &startAddr) < 0 || argint(1, &length) < 0){
    return -1;
  }

  for(int i = 0; i < NOFILE; i++){
    if (pr->vmaps[i] == 0) {
      continue;
    }
    if ((uint64)pr->vmaps[i]->addr == startAddr){
      if (length >= pr->vmaps[i]->len) {
        length = pr->vmaps[i]->len;
      }
      if (pr->vmaps[i]->prot & PROT_WRITE && pr->vmaps[i]->flag == MAP_SHARED) {
        begin_op();
        ilock(pr->vmaps[i]->file->ip);
        writei(pr->vmaps[i]->file->ip, 1, (uint64)startAddr, 0, length);
        iunlock(pr->vmaps[i]->file->ip);
        end_op();
      }
      uvmunmap(pr->pagetable, (uint64)startAddr, length/PGSIZE, 1);
      if (length == pr->vmaps[i]->len){
        fileclose(pr->vmaps[i]->file);
        vma_free(pr->vmaps[i]);
        pr->vmaps[i] = 0;
        return 0;
      } else {
        pr->vmaps[i]->addr += length;
        pr->vmaps[i]->len -= length;
        return 0;
      }
    }
  }
  return -1;
}
```

trap.c

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;
```

```c
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(p->killed)
    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 sstatus &c registers,
  // so don't enable until done with those registers.
  intr_on();

  syscall();
} else if((which_dev = devintr()) != 0){
  // ok
}  else if(r_scause() == 13 || r_scause() == 15) {
  uint64 stval = r_stval();
  if (stval >= p->sz) {
    p->killed = 1;
  } else {
    uint64 protectTop = PGROUNDDOWN(p->trapframe->sp);
    uint64 stvalTop = PGROUNDUP(stval);
    if (protectTop != stvalTop) {
      struct vma *vmap;
      int i;
      uint64 addr;
      for (i = 0; i < NOFILE; i++) {
        if (p->vmaps[i] == 0) {
          continue;
        }
        addr = (uint64) (p->vmaps[i]->addr);
        if (addr <= stval && stval < addr + p->vmaps[i]->len) {
          vmap = p->vmaps[i];
          break;
        }
      }
      if (i != NOFILE){
        char *mem = kalloc();
        int prot = PTE_U;
        if (mem == 0) {
          p->killed = 1;
        } else {
          memset(mem, 0, PGSIZE);
```

```
            ilock(vmap->file->ip);
            readi(vmap->file->ip, 0, (uint64) mem, PGROUNDDOWN(stval - addr),
PGSIZE);
            iunlock(vmap->file->ip);
            if (vmap->prot & PROT_READ){
              prot |= PTE_R;
            }
            if (vmap->prot & PROT_WRITE){
              prot |= PTE_W;
            }
            if (mappages(p->pagetable, PGROUNDDOWN(stval), PGSIZE, (uint64)mem,
prot) != 0) {
              kfree(mem);
              p->killed = 1;
            }
          }
        }else {
          p->killed = 1;
        }
      } else {
        p->killed = 1;
      }
    }
  }else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

exit.c

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
```

```
    freeproc(np);
    release(&np->lock);
    return -1;
  }

    for(i=0; i < NOFILE; i++){
    if (p->vmaps[i]){
      np->vmaps[i] = vma_alloc();
      np->vmaps[i]->addr = p->vmaps[i]->addr;
      np->vmaps[i]->len = p->vmaps[i]->len;
      np->vmaps[i]->prot = p->vmaps[i]->prot;
      np->vmaps[i]->flag = p->vmaps[i]->flag;
      np->vmaps[i]->file = p->vmaps[i]->file;
      filedup(p->vmaps[i]->file);
    }
  }

  np->sz = p->sz;

  np->parent = p;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  np->state = RUNNABLE;

  release(&np->lock);

  return pid;
}


// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait().
void
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    panic("init exiting");

  for(int i = 0; i < NOFILE; i++){
    if(p->vmaps[i]){
```

```
      struct vma *vmap = p->vmaps[i];
      if (vmap->prot & PROT_WRITE && vmap->flag == MAP_SHARED){
        begin_op();
        ilock(vmap->file->ip);
        writei(vmap->file->ip, 1, (uint64)vmap->addr, 0, vmap->len);
        iunlock(vmap->file->ip);
        end_op();
      }
      fileclose(vmap->file);
      vma_free(vmap);
      p->vmaps[i] = 0;
    }
  }

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  // we might re-parent a child to init. we can't be precise about
  // waking up init, since we can't acquire its lock once we've
  // acquired any other proc lock. so wake up init whether that's
  // necessary or not. init may miss this wakeup, but that seems
  // harmless.
  acquire(&initproc->lock);
  wakeup1(initproc);
  release(&initproc->lock);

  // grab a copy of p->parent, to ensure that we unlock the same
  // parent we locked. in case our parent gives us away to init while
  // we're waiting for the parent lock. we may then race with an
  // exiting parent, but the result will be a harmless spurious wakeup
  // to a dead or wrong process; proc structs are never re-allocated
  // as anything else.
  acquire(&p->lock);
  struct proc *original_parent = p->parent;
  release(&p->lock);

  // we need the parent's lock in order to wake it up from wait().
  // the parent-then-child rule says we have to lock it first.
  acquire(&original_parent->lock);

  acquire(&p->lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup1(original_parent);
```

```
  p->xstate = status;
  p->state = ZOMBIE;

  release(&original_parent->lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

测试：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
```

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
```

```
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

可以通过 mmaptest 和 usertests 测试

实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：

```
== Test running mmaptest ==
$ make qemu-gdb
(5.0s)
== Test   mmaptest: mmap f ==
  mmaptest: mmap f: OK
== Test   mmaptest: mmap private ==
  mmaptest: mmap private: OK
== Test   mmaptest: mmap read-only ==
  mmaptest: mmap read-only: OK
== Test   mmaptest: mmap read/write ==
  mmaptest: mmap read/write: OK
== Test   mmaptest: mmap dirty ==
  mmaptest: mmap dirty: OK
== Test   mmaptest: not-mapped unmap ==
  mmaptest: not-mapped unmap: OK
== Test   mmaptest: two files ==
  mmaptest: two files: OK
== Test   mmaptest: fork_test ==
  mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (236.9s)
== Test time ==
time: OK
Score: 140/140
```

通过了所有测试

问题以及解决办法：

a. 如何确定文件映射到用户虚拟地址空间的位置

本次实验是想实现文件到用户空间的映射，其实也就是为文件指针设置一个合适的数据结构，并将其放置在用户空间的一块区域上。但是文件应当被映射到具体哪个位置呢？这里我采用的是将其映射到当前内存区域的最后面，这样可以确保不会和已经存在的其他信息产生位置冲突或者覆盖。

实验心得：

1. 了解了如何将文件映射到用户虚拟内存中，并将修改后内容写回磁盘
2. 进一步熟悉了懒加载和文件系统