xv6 lab 报告：lab mmap

任务：mmap

mmap和munmap系统调用允许UNIX程序对其地址空间进行详细的控制。它们可用于在进程间共享内存，将文件映射到进程地址空间，并作为用户级页面故障方案的一部分，如讲座中讨论的垃圾回收算法。在本实验室中，将在xv6中添加mmap和munmap，重点是内存映射文件。

任务目标是实现mmap和munmap系统调用。

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, int length)
```

在本lab中对mmap系统调用作出了一些简化：我们可以假设addr和offset永远为0（其实这个简化有和没有没啥区别……）

第一步：创建系统调用入口

与lab syscall相同

user/user.h

```
void *mmap(void*, int, int, int, int, int);
int munmap(void*, int);
```

user/usys.pl

```
entry("mmap");
entry("munmap");
```

kernel/syscall.h

```
#define SYS_mmap   22
#define SYS_munmap 23
```

kernel/syscall.c

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);

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

```
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
};
```

第二步：实现mmap系统调用处理函数

kernel/proc.h

新增VMA结构定义，用于存放mmap创建的映射信息

```
#define NVMAS 16

struct VMA {
  uint64 address;
  int length;
  struct file *file;
  int prot;
  int flags;
  int offset;
  char used;
};

struct proc {
  ...
  struct VMA vmas[NVMAS];         // VMAs
};
```

kernel/sysfile.c

mmap系统调用实现：在进程的VMA表中新增一项存放对应的mmap信息

```
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  struct file *f;
  int i;
  if (argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) ||
      argint(3, &flags) < 0 || argfd(4, &fd, &f) < 0 || argint(5, &offset) < 0) // 读取参数
    return -1;
  if (addr != 0 || offset != 0) return -1;    // addr和offset必须为0
  if ((prot & PROT_READ) && !f->readable) return -1;  // 文件描述符无法读
  if ((flags & MAP_SHARED) && ((prot & PROT_WRITE) && !f->writable)) return -1; // 标记为
MAP_SHARED但是无法写文件

  struct proc *p = myproc();
  addr = p->sz;
  p->sz += length;
  p->sz = PGROUNDUP(p->sz); // 扩展进程p的空间大小以获取mmap所需的虚拟地址范围
  for (i = 0; i < NVMAS; ++i) {
    if (!p->vmas[i].used) { // 寻找一个VMA表中的空位，存放mmap映射信息
      p->vmas[i].used = 1;
      p->vmas[i].address = addr;
      p->vmas[i].length = length;
      p->vmas[i].file = f;
      p->vmas[i].prot = prot;
      p->vmas[i].flags = flags;
      p->vmas[i].offset = 0;
      filedup(f); // 文件f的引用数+1
      return addr;
    }
  }
  return -1;
}
```

第三步：处理当进程访问mmap对应的地址时的操作

kernel/trap.c

创建newpage函数用来给进程的某一虚拟地址创建物理页面。与lazy-allocation lab基本相同

在usertrap函数中，创建page fault相关的处理代码，处理落在某一个mmap范围内的page fault

```
uint64 newpage(struct proc *p, uint64 addr, int flags)
{
  if (addr >= p->sz) {
```

```
    printf("index out of range: %p/%p\n", addr, p->sz);
    return 0;
  }
  addr = PGROUNDDOWN(addr);
  char *mem = kalloc();
  if (mem == 0) {
    printf("memory allocation failed!\n");
    return 0;
  }
  memset(mem, 0, PGSIZE);
  if (mappages(p->pagetable, addr, PGSIZE, (uint64)mem, flags | PTE_U) != 0) {
    kfree(mem);
    return 0;
  }
  return (uint64)mem;
}

void
usertrap(void)
{
  ...
  } else if ((which_dev = devintr()) != 0) {
    // ok
  } else if (r_scause() == 13 || r_scause() == 15) {  // Page fault, 判断是否属于mmap
    char valid = 0;
    addr = r_stval();
    addr = PGROUNDDOWN(addr);
    for (i = 0; i < NVMAS; i++) {
      struct VMA *vma = &(p->vmas[i]);
      if (vma->address <= addr
          && addr < vma->address + vma->length) { // a unmapped mmap page
        int prot = 0;
        if (vma->prot & PROT_READ) prot |= PTE_R; // 看一下对应的内存页需要哪些flags
        if (vma->prot & PROT_WRITE) prot |= PTE_W;
        if (vma->prot & PROT_EXEC) prot |= PTE_X;
        newpage(p, addr, prot); // 创建新的内存页

        acquiresleep(&vma->file->ip->lock);
        readi(vma->file->ip, 1, addr, addr - vma->address, PGSIZE); // 从文件读取该页
        releasesleep(&vma->file->ip->lock);
        valid = 1;
        break;
      }
    }
    if (!valid) {
      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;
    }
  } 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;
  }
  ...
}
```

第四步：处理munmap

kernel/fs.c

创建一个函数用于向某一文件写入一段内存地址，对于未映射的页面则直接跳过（这边可以更进一步优化：只有标记为dirty的页面才写入）

```
int
writeback(struct file *f, uint64 addr, uint off, uint npages)
{
  for (int i = 0; i < npages; i++) {
    begin_op();
    ilock(f->ip);
    writei(f->ip, 1, addr + i * PGSIZE, off + i * PGSIZE, PGSIZE);
    iunlock(f->ip);
    end_op();
  }
  return 0;
}
```

kernel/sysfile.c

处理munmap，分三种情况讨论：删除某一映射的开头；删除某一映射的结尾；删除某一映射的全部

三种情况处理的流程都包括将原有的虚拟内存地址对应的页表项删除，以及当标记为MAP_SHARED的时候的回写操作。

假如删除了整个映射，则移除VMA表中的映射项并将文件引用数-1；否则修改对应的address，length，offset项

```
uint64
sys_munmap(void)
{
  uint64 addr;
  int length;
  if (argaddr(0, &addr) < 0 || argint(1, &length) < 0) return -1;
  struct proc *p = myproc();
  for (int i = 0; i < NVMAS; ++i) {
    struct VMA *vma = &p->vmas[i];
    if (vma->used && vma->address <= addr && addr < vma->address + vma->length) {
      if (vma->address != addr && vma->address + vma->length != addr + length)
        return -1;
      if (length > vma->length) return -1;
      if (vma->flags & MAP_SHARED)  // 标记为MAP_SHARED，则将改动写回到文件
        writeback(vma->file, PGROUNDDOWN(addr), vma->offset, PRANGE(addr,
```

```
                addr + length) / PGSIZE);
      if (length == vma->length) {  // 删除了整个映射
        uvmunmap(p->pagetable, PGROUNDDOWN(addr), PRANGE(addr + length, addr) / PGSIZE,
                 1);
        fileclose(vma->file);       // 文件引用数-1
        vma->used = 0;              // 删除映射
        return 0;
      }
      if (vma->address == addr) {   // 删除这个映射的开头
        if (PGROUNDDOWN(addr) != PGROUNDDOWN(vma->address))
          uvmunmap(p->pagetable, PGROUNDDOWN(vma->address),
                   (PGROUNDDOWN(addr) - PGROUNDDOWN(vma->address)) / PGSIZE, 1);
        vma->address += length;     // address, length, offset对应修改
        vma->length -= length;
        vma->offset += length;
        return 0;
      }
      if (vma->address + vma->length == addr + length) {  // 删除这个映射的结尾
        int endold = PGROUNDUP(vma->address + vma->length);
        int endnew = PGROUNDUP(addr + length);
        if (endold != endnew)
          uvmunmap(p->pagetable, endnew, (endold - endnew) / PGSIZE, 1);
        vma->length -= length;      // 只需要修改length
      }
    }
  }
  return -1;
}
```

kernel/vm.c

修改uvmunmap使得当该页表项本来就不存在的时候直接忽视掉而非报错

```
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_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if((*pte & PTE_V) != 0 && do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
```

```
    *pte = 0;
  }
}
```

第五步：完善exit()和fork()

kernel/proc.c

修改exit()使得它移除该进程的mmap占用的虚拟地址

修改fork使得子进程继承父进程的mmap映射。此处采用直接复制而非写时复制

```
void
exit(int status)
{
  ...
  begin_op();
  for (int i = 0; i < NVMAS; i++) {
    struct VMA *vma = &p->vmas[i];
    if (vma->used) {
      if (vma->flags & MAP_SHARED)
        writeback(vma->file, PGROUNDDOWN(vma->address), vma->offset,
                  PRANGE(vma->address, vma->address + vma->length) / PGSIZE); // 写回文件
      fileclose(vma->file);     // 文件引用数-1
      vma->used = 0;
    }
  }
  end_op();
  ...
}

int
fork(void)
{
  ...
  memmove(np->vmas, p->vmas, sizeof(p->vmas));  // 复制整个VMA表
  for (int i = 0; i < NVMAS; i++) {
    if (np->vmas[i].used) filedup(np->vmas[i].file);  // 每个文件的引用数+1
  }
  ...
}
```

测试结果

```
== Test running mmaptest ==
$ make qemu-gdb
(6.6s)
== 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 (130.0s)
== Test time ==
time: OK
Score: 140/140
```