Lab10 mmap

1. mmap(hard)

1）实验目的

> You should implement enough `mmap` and `munmap` functionality to make the `mmaptest` test program work. If `mmaptest` doesn't use a `mmap` feature, you don't need to implement that feature
> 您应该实现足够的 `mmap` 和 `munmap` 功能，以使 `mmaptest` 测试程序正常工作。 如果 `mmaptest` 不使用 `mmap` 功能，则无需实现该功能。

2）实验步骤

编写代码

添加 `mmap` 和 `munmap` 两个系统调用
在 `user/user.h` 文件中声明需要添加的系统调用 `mmap` 和 `munmap`

```
int uptime(void);
// lab10
void* mmap(void *, int, int, int, int, uint);
int munmap(void *, int);
```

在 `user/usys.pl` 文件中添加上述系统调用的入口

```
# lab10
...
entry("uptime");
entry("mmap");
entry("munmap");
```

在 kernel/syscall.h 添加定义

```
#define SYS_close 21
// lab10
#define SYS_mmap 22
#define SYS_munmap 23
```

在 kernel/syscall.c 添加函数声明

```
// lab10
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
...
static uint64 (*syscalls[])(void) = {
...
// lab10
[SYS_close]  sys_close,
[SYS_mmap] sys_mmap,
[SYS_munmap] sys_munmap,
};
```

在 Makefile 文件中添加对 mmaptest 的编译

```
    ...
        $U/_zombie\
        $U/_mmaptest\
```

在 kernel/proc.h 文件中结构体 struct proc 之前定义虚拟内存区域结构体 vma， 一个进程具有有多少 vma， 并且在 struct proc 结构体中添加数据成员 vma 数组

```
// 定义虚拟内存区域结构体lab10
#define VMASIZE 16

struct vma {
  struct file *file;  // 文件结构体指针
  uint64 addr;        // mmap映射起始地址
  int len;            // mmap映射内存大小
  int prot;           // 用户权限
  int flags;          // mmap标志位
  int offset;         // 文件偏移量
  int fd;             // 文件描述符
  int used;           // 是否使用过
};

// Per-process state
struct proc {
  ...
  char name[16];               // Process name (debugging)
  // 一个进程对应一个vma数组 lab10
  struct vma vma[VMASIZE];
};
```

在 kernel/sysfile.c 文件夹中实现上述两个系统调用

```
uint64 sys_mmap(void) {
  uint64 addr;
  int len, prot, flags, fd, offset;
  struct proc *p = myproc();
  struct file *file;
  if(argaddr(0, &addr) || argint(1, &len) || argint(2, &prot) ||
    argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset))
    return -1;

  if(!file->writable && (prot & PROT_WRITE) && flags == MAP_SHARED)
    return -1;

  len = PGROUNDUP(len);
  if(p->sz > MAXVA - len)
    return -1;

  for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].used == 0) {
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].len = len;
      p->vma[i].prot = prot;
      p->vma[i].flags = flags;
      p->vma[i].fd = fd;
      p->vma[i].file = file;
      p->vma[i].offset = offset;
      filedup(file);
      p->sz += len;
      return p->vma[i].addr;
    }
  }
  return -1;
}

uint64 sys_munmap(void) {
  uint64 addr;
  int len;
  struct proc *p = myproc();
  struct vma *vma = 0;
  if(argaddr(0, &addr) || argint(1, &len))
    return -1;
  addr = PGROUNDDOWN(addr);
  len = PGROUNDUP(len);
  for(int i = 0; i < VMASIZE; i++) {
    if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].len) {
      vma = &p->vma[i];
      break;
    }
  }
  if(vma == 0)
```

```
    return 0;
  if(vma->addr == addr) {
    vma->addr += len;
    vma->len -= len;
    if(vma->flags & MAP_SHARED)
      filewrite(vma->file, addr, len);
    uvmunmap(p->pagetable, addr, len/PGSIZE, 1);
    if(vma->len == 0) {
      fileclose(vma->file);
      vma->used = 0;
    }
  }
  return 0;
}
```

修改 `kernel/trap.c` 文件中的 `usertrap()` 函数，增加对page fault 的判断

```
void
usertrap(void)
{
  ...
  if(r_scause() == 8){
    ...
  }
  // 处理page fault lab10
  else if (r_scause() == 13 || r_scause() == 15) {
    uint64 va = r_stval();
    if(va >= p->sz || va > MAXVA || PGROUNDUP(va) == PGROUNDDOWN(p->trapframe->sp))
      p->killed = 1;
          else {
      struct vma *vma = 0;
      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].len) {
          vma = &p->vma[i];
          break;
        }
      }

      if(vma) {
        va = PGROUNDDOWN(va);
        uint64 offset = va - vma->addr;
        uint64 mem = (uint64)kalloc();
        if(mem == 0) {
          p->killed = 1;
        }
        else {
          memset((void*)mem, 0, PGSIZE);
                      ilock(vma->file->ip);
          readi(vma->file->ip, 0, mem, offset, PGSIZE);
          iunlock(vma->file->ip);
          int flag = PTE_U;
          if(vma->prot & PROT_READ) flag |= PTE_R;
          if(vma->prot & PROT_WRITE) flag |= PTE_W;
          if(vma->prot & PROT_EXEC) flag |= PTE_X;
          if(mappages(p->pagetable, va, PGSIZE, mem, flag) != 0) {
            kfree((void*)mem);
            p->killed = 1;
          }
        }
      }
    }
  }
  else if((which_dev = devintr()) != 0){
    // ok
  }
```

```
  ...
}
```

在 kernel/vm.c 文件中修改 uvmunmap() 函数和 uvmcopy() 函数，取消不有效就panic的操作

```
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)
      // panic("uvmunmap: not mapped");
      continue;
    if(PTE_FLAGS(*pte) == PTE_V)
      // panic("uvmunmap: not a leaf");
      continue;
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      // panic("uvmcopy: page not present");
      continue;
    pa = PTE2PA(*pte);
    ...
  }
  ...
}
```

修改 kernel/proc.c 文件中的 exit() 函数和 fork() 函数

```
int
fork(void)
{
  ...

  np->state = RUNNABLE;

  // lab10
  for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].used){
      memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
      filedup(p->vma[i].file);
    }
  }

  release(&np->lock);

  return pid;
}

void
exit(int status)
{
  ...
  // 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;
    }
  }

  for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].used) {
      if(p->vma[i].flags & MAP_SHARED)
        filewrite(p->vma[i].file, p->vma[i].addr, p->vma[i].len);
      fileclose(p->vma[i].file);
      uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].len / PGSIZE, 1);
      p->vma[i].used = 0;
    }
  }

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

  ...
}
```

测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `mmaptest` 进行测试

```
$ 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
```

测试通过
键入指令 `usertests` 进行测试

```
$ 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 reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3242
            sepc=0x00000000000056a4 stval=0x00000000000056a4
usertrap(): unexpected scause 0x000000000000000c pid=3243
            sepc=0x00000000000056a4 stval=0x00000000000056a4
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test dirtest: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validatetest: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
```

```
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... 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
```

测试通过

键入 `Ctrl+a` ， 松开，然后键入 `x` ， 退出xv6系统
退出xv6进行单元测试

```
root@LAPTOP-UER420HO:~/lab10_mmap# ./grade-lab-mmap make: 'kernel/kernel' is up to date.
== Test running mmaptest == (1.9s)
== 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 == usertests: OK (87.9s)
== Test time ==
time: OK
Score: 140/140
```

3）实验中遇到的问题和解决方法

增加文件描述符fd

4）实验心得

这个实验的要求有点抽象，在看懂两个系统系调用究竟是要干什么上花费了一点时间，虽然提示上说的很详细，但是还在网上找了很多资料来理解

其中的主要思想是给每个进程一个虚拟内存区域数组来形成自己的地址空间，不受其他进程访问的打扰

对于 `mmap()` 函数，它是一种用于在用户空间创建内存映射区域的系统调用。它允许进程将一个文件或其他设备映射到其自己的地址空间中，以便可以像访问内存一样访问这些文件或设备
使用 `mmap()` 系统调用时，进程需要指定以下参数：

- **起始地址（`addr`）**：指定映射区域的起始地址。如果指定为0，操作系统将选择合适的地址
- **长度（`len`）**：指定映射区域的长度。通常以页面大小（通常为4KB）的倍数进行对齐
- **访问权限（`prot`）**：指定映射区域的访问权限，包括可读（`PROT_READ`）、可写（`PROT_WRITE`）、可执行（`PROT_EXEC`）等
- **映射标志（`flags`）**：指定映射的一些属性和行为。例如，可以指定共享映射（`MAP_SHARED`）或私有映射（`MAP_PRIVATE`）、映射后立即预读（`MAP_POPULATE`）等
- **文件描述符（`fd`）**：指定要映射的文件的文件描述符
- **文件偏移量（`offset`）**：指定文件中的偏移量，从该偏移量开始映射数据

调用 `mmap()` 后，操作系统将在进程的地址空间中创建一个新的VMA，将文件内容或设备映射到该VMA中。进程可以通过访问VMA来读取或修改映射区域的数据，就好像在访问内存一样。对映射区域的修改也可以写回到文件中（如果是共享映射）

相反，对于 `munmap()` 函数，是一种用于解除内存映射区域的系统调用。它允许进程释放先前使用 `mmap()` 系统调用创建的内存映射区域

使用 `munmap()` 系统调用时，进程需要指定以下参数：

- **起始地址（`addr`）**：指定要解除映射的区域的起始地址
- **长度（`len`）**：指定要解除映射的区域的长度

调用 `munmap()` 后，操作系统将解除进程地址空间中指定区域的映射。它会释放相关的资源，包括VMA），并更新页表以反映解除映射的改变。解除映射后，进程将无法再访问该区域的数据

需要注意的是，解除映射的起始地址和长度必须与之前调用 `mmap` 时指定的地址和长度匹配。否则，解除映射可能会失败或导致不可预测的行为

这种机制有以下几个优点

- 读取文件跨过了页缓存，减少了拷贝次数
- 用内存读写代替I/O读写，提高了文件读写的效率

- 用户空间和内核空间可以在映射区域中高效交互
- 进程间可以通过映射区域进行通信和共享
- 可用于实现高效的大规模数据传输，利用 `mmap` 实现磁盘空间代替内存