Lab 10 mmap

mmap (hard)

实验目的

本次实验旨在实现操作系统 xv6 中的 mmap 和 munmap 两个系统调用，主要实现其 memory-mapped files 功能，即将一个文件映射到内存中。具体目的包括：

- 熟悉系统调用的实现过程和原理。
- 掌握在操作系统中如何管理虚拟内存和内存映射。
- 实现 mmap 系统调用，将文件映射到指定的虚拟地址。
- 实现 munmap 系统调用，取消内存映射。

实验步骤

1. 首先，需要定义结构体 VMA 来记录进程的内存映射信息，并在进程初始化时对其进行初始化。

```
#define VMA_MAX 16

struct VMA {
  int valid;        // 有效位，0 表示无效
  uint64 addr;      // 起始地址
  int len;          // 长度
  int prot;         // 权限（read/write）
  int flags;        // 区域类型（shared/private）
  int off;          // 偏移量
  struct file *f;   // 映射的文件
  uint64 mapcnt;    // 已映射的页数量
};

struct proc {
  // ...
  struct VMA vma[VMA_MAX];  // 虚拟内存地址区域数组
  uint64 maxaddr;           // 可用虚拟内存最大地址
  // ...
};
```

2. mmap 系统调用的实现

```
// sysfile.c

uint64 sys_mmap(void) {
  // 获取参数和进程
  uint64 addr;
  int len, prot, flags, fd, off;
  if (argaddr(0, &addr) < 0 || argint(1, &len) < 0 || argint(2, &prot) < 0
```

```
||
      argint(3, &flags) < 0 || argint(4, &fd) < 0 || argint(5, &off) < 0)
    return -1;
  struct proc *p = myproc();
  struct file *f = p->ofile[fd];

  // 确保权限
  if ((flags == MAP_SHARED && f->writable == 0 && (prot & PROT_WRITE)))
    return -1;

  // 找到一个空的 VMA 并初始化
  int idx;
  for (idx = 0; idx < VMA_MAX; idx++)
    if (p->vma[idx].valid == 0)
      break;

  if (idx == VMA_MAX)
    panic("no empty vma field");

  struct VMA *vp = &p->vma[idx];
  vp->valid = 1;
  vp->len = len;
  vp->flags = flags;
  vp->off = off;
  vp->prot = prot;
  vp->f = f;
  filedup(f);
  vp->addr = (p->maxaddr -= len);

  return vp->addr;
}
```

3. 处理缺页中断，根据地址找到对应的 VMA 并进行映射。

```
// trap.c

// ...
} else if (r_scause() == 0xd) {  // 缺页中断
  uint64 addr = r_stval();
  struct VMA *vp = 0;
  for (int i = 0; i < VMA_MAX; i++) {
    if (p->vma[i].addr <= addr && addr < p->vma[i].addr + p->vma[i].len &&
        p->vma[i].valid == 1) {
      vp = &p->vma[i];
      break;
    }
  }
  if (vp != 0) {
    uint64 mem = (uint64)kalloc();
    memset((void *)mem, 0, PGSIZE);

    if (-1 == mappages(p->pagetable, addr, PGSIZE,
```

```
                        mem, PTE_U | PTE_V | (vp->prot << 1)))
      panic("pagefault map error");

    vp->mapcnt += PGSIZE;
    ilock(vp->f->ip);
    readi(vp->f->ip, 0, mem, addr - vp->addr, PGSIZE);
    iunlock(vp->f->ip);
  } 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;
  }
} else {
// ...
}
```

4. munmap 系统调用的实现

```
// sysfile.c

uint64 sys_munmap(void) {
  // 获取参数和进程
  uint64 addr;
  int len;
  if (argaddr(0, &addr) < 0 || argint(1, &len) < 0)
    return -1;
  struct proc *p = myproc();

  // 找到目标 VMA
  struct VMA *vp = 0;
  for (int i = 0; i < VMA_MAX; i++) {
    if (p->vma[i].addr <= addr && addr < p->vma[i].addr + p->vma[i].len &&
        p->vma[i].valid == 1) {
      vp = &p->vma[i];
      break;
    }
  }
  if (vp == 0)
    panic("munmap no such vma");

  // 如果页面已经映射，写回修改（如果需要）并解映射
  if (walkaddr(p->pagetable, addr) != 0) {
    if (vp->flags == MAP_SHARED)
      filewriteoff(vp->f, addr, len, addr - vp->addr);
    uvmunmap(p->pagetable, addr, len / PGSIZE, 1);
    return 0;
  }

  // 维护文件引用和 VMA 有效性
  if (0 == (vp->mapcnt -= len)) {
    fileclose(vp->f);
```

```
    vp->valid = 0;
  }
  return 0;
}
```

5. 在 fork 和 exit 中复制、删除当前进程的 VMA 字段。

```
// proc.c

// ...
int fork(void) {
  // ...

  np->maxaddr = p->maxaddr;
  for (int i = 0; i < VMA_MAX; i++) {
    if (p->vma[i].valid) {
      filedup(p->vma[i].f);
      memmove(&np->vma[i], &p->vma[i], sizeof(struct VMA));
    }
  }
  return pid;
}
// ...

void exit(int status) {
  struct proc *p = myproc();

  for (int i = 0; i < VMA_MAX; i++) {
    if (p->vma[i].valid == 1) {
      struct VMA *vp = &p->vma[i];
      for (uint64 addr = vp->addr; addr < vp->addr + vp->len;
           addr += PGSIZE) {
        if (walkaddr(p->pagetable, addr) != 0) {
          if (vp->flags == MAP_SHARED)
            filewriteoff(vp->f, addr, PGSIZE, addr - vp->addr);
          uvmunmap(p->pagetable, addr, 1, 1);
        }
      }
      fileclose(p->vma[i].f);
      p->vma[i].valid = 0;
    }
  }
  // ...
}
// ...
```

实验中遇到的问题和解决办法

1. 问题：在编译过程中遇到了隐式声明 symlink 函数的错误。

- 解决办法： 在 user/symlinktest.c 文件中添加 #include <unistd.h> 头文件，以解决隐式声明问题。

2. 在映射地址的确定过程中，需要考虑如何正确处理延迟申请的情况。

- 解决办法： 选择从 trapframe 的底部向下生长，同时修改对 uvmunmap 的实现，使得只取消已经映射的页，以解决这个问题。

3. 在 munmap 函数中，假定要 unmap 的页全是 mapped 或全是 unmapped 的，存在瑕疵。

- 解决办法： 修改 munmap 实现，判断页是否已经映射，如果是，则写回修改（如果需要），然后解映射。同时，维护文件引用和 VMA 有效性。

实验心得

通过本次实验，我深入了解了操作系统中内存映射的原理与实现。学习如何处理系统调用，如何管理虚拟内存，以及如何在内核中维护进程的内存映射信息。在实现 mmap 和 munmap 的过程中，我遇到了一些挑战，尤其是在处理缺页中断和确定映射地址时。通过仔细阅读 xv6 源码，查阅相关文档，并结合调试，我逐步克服了这些问题，最终成功完成了实验任务。

在整个实验过程中，我加深了对操作系统内存管理的理解，掌握了操作系统内核中的数据结构和函数调用，提升了我的编程和调试能力。通过不断的实践和探索，我对操作系统的各个组成部分有了更深入的认识，为我今后在系统编程和操作系统领域的学习和研究打下了坚实的基础。

评分

```
== Test running mmaptest ==
$ make qemu-gdb
(9.1s)
== 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 (289.1s)
== Test time ==
time: OK
Score: 140/140
```