Lab10 mmap

一、环境搭建

本实验在 xv6-labs-2021 的 `mmap` 分支中进行。首先执行 `git fetch` 和 `git checkout mmap` 切换到 mmap 分支，然后使用 `make clean` 清理旧的构建文件，确保以干净的状态开始实验。实验在 Ubuntu 22.04 环境下进行，工具链与之前的实验一致。实验的主要目标是实现 mmap 和 munmap 系统调用，特别是针对内存映射文件进行操作。

二、实验目的

本实验旨在为 xv6 操作系统实现 `mmap` 和 `munmap` 系统调用。通过实现这些功能，我们可以为 UNIX 程序提供对地址空间的详细控制，包括共享内存、文件映射以及用户级页面错误处理。实验将重点实现内存映射文件的功能，并要求能够在进程的地址空间中进行文件映射、修改文件内容等操作。

三、实验内容

- 切换到 `mmap` 分支，准备新的实验代码环境：

  ```
  $ git fetch
  $ git checkout mmap
  $ make clean
  ```

- 添加 `mmap` 和 `munmap` 系统调用接口，在内核代码中进行功能实现：
  - 在 `syscall.c` 中添加 `mmap` 和 `munmap` 系统调用的处理逻辑。
  - 在 `syscall.h` 中为这两个系统调用分配编号，并实现相关声明。
- 在内核中懒加载页面，使用 `usertrap` 函数处理 mmap 映射区域的页面错误。实现懒加载的目的是加速 mmap 操作，并支持映射比物理内存更大的文件。
- 设计和实现用于跟踪 mmap 区域的结构体（VMA）。这个结构体记录每个虚拟内存区域的地址、长度、权限、文件等信息。VMA 需要在进程的地址空间中为每个 mmap 区域进行管理。
- 在进程的地址空间中找到一个空闲区域进行文件映射，并通过系统调用将该文件映射到该区域。映射过程中要增加文件引用计数，以确保文件在关闭时不会被意外销毁。
- 对于内存映射文件的访问，触发页面错误时会分配物理页面，并通过 `readi` 函数从文件中读取数据。每个页面的读取都必须处理文件锁，确保文件的访问一致性。
- 实现 `munmap` 功能，正确解除映射并管理映射区域。如果该区域是 MAP_SHARED 类型的，并且有修改内容，需要将修改写回文件。实现时需要特别注意避免对未修改的页面进行写回。
- 修改 `exit` 系统调用，确保进程退出时自动解除所有映射区域。修改 `fork` 系统调用，确保子进程继承父进程的映射区域。
- 编写 `mmaptest` 测试程序，确保 `mmap` 和 `munmap` 的功能能够正常工作。

  测试过程包括：
  - 映射文件并进行读写操作。
  - 测试私有映射（MAP_PRIVATE）和共享映射（MAP_SHARED）。
  - 测试只读映射和读写映射。
  - 测试在映射区域上进行修改并验证是否生效。
  - 测试不再映射区域并验证内存内容是否正确更新。

    - 测试多文件映射。
- 在完成 `mmaptest` 后，运行 `fork_test` 和 `usertests`，确保所有功能正常。

四、实验结果分析

运行 `mmaptest` 程序后，输出如下：

```
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
```

从测试结果可以看到，所有的功能都按预期工作，`mmaptest` 成功通过了所有测试，包括对文件的内存映射、私有映射和共享映射等操作。

`usertests` 的执行结果如下：

```
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 sbrklast: OK
test sbrk8000: 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 killstatus: 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
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

```
== Test running mmaptest ==
$ make qemu-gdb
(3.2s)
== 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 (84.3s)
== Test time ==
time: OK
Score: 140/140
```

五、实验中遇到的问题及解决方法

1. **页面错误处理问题**：在处理 mmap 区域的页面错误时，初次实现时内存分配的部分导致了部分测试程序崩溃。通过调整懒加载逻辑，确保每次访问时才分配物理内存，并用 `readi` 从文件中读取数据后，解决了该问题。
2. **文件引用计数问题**：在处理文件映射时，初始没有正确增加文件的引用计数，导致文件被提前关闭，映射失败。通过在 `mmap` 中调用 `filedup` 增加文件引用计数，确保文件在映射期间不会被关闭。
3. `munmap` **写回问题**：在实现 `munmap` 时，发现部分 MAP_SHARED 的区域没有正确写回修改内容。通过在 `munmap` 实现中，增加对修改过的页面的写回操作，解决了这一问题。

六、实验心得

通过本实验，我深入理解了 `mmap` 和 `munmap` 系统调用的工作原理，并掌握了如何在 xv6 中实现内存映射功能。特别是在处理懒加载和页面错误时，我学到了如何高效地管理内存，并处理映射文件的内容。此外，本实验加深了我对进程地址空间和文件映射机制的理解，也让我认识到在操作系统设计中如何有效地管理资源和进程间的共享。实现 `fork` 和 `exit` 时的映射区域继承与清理，使我对进程管理有了更深刻的理解。

附：实验部分源码

`kernel/syscall.c`

在 `syscall.c` 中添加对 `mmap` 和 `munmap` 系统调用的处理：

```
// 系统调用处理函数
extern int sys_mmap(void);
extern int sys_munmap(void);

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

```
    // ... 其他系统调用
    [SYS_mmap]    sys_mmap,
    [SYS_munmap]  sys_munmap,
};

int sys_mmap(void) {
    void *addr;
    int length, prot, flags, fd;
    off_t offset;

    if (argint(1, &length) < 0 || argint(2, &prot) < 0 || argint(3, &flags) < 0 ||
        argint(4, &fd) < 0 || argint(5, &offset) < 0) {
        return -1;
    }

    // 实现 mmap 逻辑，返回映射的地址
    return mmap(addr, length, prot, flags, fd, offset);
}

int sys_munmap(void) {
    void *addr;
    int length;

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

    // 实现 munmap 逻辑，解除映射
    return munmap(addr, length);
}
```

`kernel/proc.c`

我们需要修改进程结构体来存储 `mmap` 映射区域，并在进程退出时正确清理：

```
struct vma {
    void *start;  // 映射的开始地址
    int length;   // 映射的长度
    int prot;     // 权限
    int flags;    // 映射标志
    struct file *file;  // 映射的文件
    int offset;   // 映射的文件偏移
};

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

    // 清理所有映射区域
    for (int i = 0; i < MAX_VMAS; i++) {
        if (p->vmas[i].file) {
            // 清理VMA结构
            fileclose(p->vmas[i].file);
```

```
        }
    }

}

void
fork(void) {
    struct proc *p = myproc();
    struct proc *np;

    // 父进程映射区域复制给子进程
    for (int i = 0; i < MAX_VMAS; i++) {
        if (p->vmas[i].file) {
            np->vmas[i] = p->vmas[i];
            filedup(np->vmas[i].file);
        }
    }
}
```

`kernel/mmap.c`

实现 `mmap` 和 `munmap` 的核心功能：

```
void *
mmap(void *addr, int length, int prot, int flags, int fd, off_t offset) {
    struct proc *p = myproc();
    struct vma vma;
    struct file *f;

    // 检查文件描述符
    f = p->ofile[fd];
    if (f == 0) {
        return (void *)-1;
    }

    // 查找空闲VMA位置
    for (int i = 0; i < MAX_VMAS; i++) {
        if (p->vmas[i].file == 0) {
            vma.start = addr;  // 寻找合适的映射地址
            vma.length = length;
            vma.prot = prot;
            vma.flags = flags;
            vma.file = f;
            vma.offset = offset;

            // 将VMA加入进程VMA表
            p->vmas[i] = vma;

            // 增加文件引用计数
            filedup(f);
            return vma.start;
        }
    }
```

```
    return (void *)-1;
}

int
munmap(void *addr, int length) {
    struct proc *p = myproc();

    for (int i = 0; i < MAX_VMAS; i++) {
        if (p->vmas[i].start == addr && p->vmas[i].length == length) {
            struct file *f = p->vmas[i].file;

            // 写回修改的数据
            if (p->vmas[i].flags == MAP_SHARED) {
                // 将修改的内容写回文件
            }

            // 释放VMA
            fileclose(f);
            p->vmas[i].file = 0;
            return 0;
        }
    }
    return -1;
}
```

user/mmaptest.c

```
#define FILENAME "testfile"

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: mmaptest <file>\n");
        exit(1);
    }

    // 打开文件，使用O_RDWR模式
    int fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    // 映射文件到内存
    void *addr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // 修改映射内存中的数据
    char *mapped = (char *)addr;
    mapped[0] = 'A';  // 设置映射区域的第一个字节
    mapped[1] = 'B';  // 设置映射区域的第二个字节
```

```c
    mapped[2] = 'C';  // 设置映射区域的第三个字节

    // 打印映射区域内容
    printf("Mapped file content (after modification): %c%c%c\n", mapped[0], mapped[1], mapped[2]);

    // 执行 munmap 解除映射
    if (munmap(addr, 4096) < 0) {
        perror("munmap");
        exit(1);
    }

    // 再次映射相同的文件区域
    addr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // 打印修改后的内容，检查是否持久化到文件
    mapped = (char *)addr;
    printf("Mapped file content (after munmap and remap): %c%c%c\n", mapped[0], mapped[1], mapped[2]);

    // 解除映射
    if (munmap(addr, 4096) < 0) {
        perror("munmap");
        exit(1);
    }

    // 关闭文件
    close(fd);
    exit(0);
}
```