Lab10: mmap (hard)

mmap (hard)

1. 实验目的

在本实验中，你将向 xv6 文件系统中添加 `mmap` 和 `munmap` 系统调用。这两个系统调用允许程序对其地址空间进行详细控制，包括共享内存、将文件映射到进程地址空间中，以及实现用户级页错误处理方案等功能。实验重点是实现内存映射文件功能。

2. 实验步骤

1. 添加系统调用

1. **在 `UPROGS` 中添加 `mmaptest`**：
   - 确保在 `UPROGS` 中添加 `mmaptest` 以便编译和测试。

```
# Makefile
UPROGS=... _mmaptest
```

2. **添加 `mmap` 和 `munmap` 系统调用**：
   - 在 `kernel/syscall.h` 中为 `mmap` 和 `munmap` 创建系统调用编号，并在 `user/user.h` 中声明它们。

```
// kernel/syscall.h
#define SYS_mmap 23
#define SYS_munmap 24
```

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

3. **在 `kernel/sysfile.c` 中实现空的 `sys_mmap` 和 `sys_munmap` 函数**：

```
// kernel/sysfile.c
void *sys_mmap(void) {
    // 实现 mmap 系统调用
    return (void *)-1;
}
```

```
int sys_munmap(void) {
    // 实现 munmap 系统调用
    return -1;
}
```

注意：确保在 `user/usys.pl` 中添加系统调用条目。

2. 管理虚拟内存区域（VMA）

1. **定义 VMA 结构**：
   - 定义一个结构来表示虚拟内存区域，记录地址、长度、权限、文件等信息。

```
// kernel/proc.h
#define MAX_VMAS 16

struct vma {
    void *addr;
    size_t length;
    int prot;
    int flags;
    struct file *file;
    int file_ref_count;
};

struct proc {
    // ...
    struct vma vmas[MAX_VMAS];
};
```

2. **实现 `mmap`**：
   - 寻找进程地址空间中的一个未使用区域来映射文件，并将 VMA 添加到进程的映射区域表中。增加文件的引用计数，以防文件在映射期间被关闭。

```
// kernel/sysfile.c
void *sys_mmap(void) {
    void *addr;
    size_t length;
    int prot, flags, fd;
    off_t offset;
    struct file *f;
    struct vma *vma;
    struct proc *p = myproc();

    if (argint(2, &prot) < 0 || argint(3, &flags) < 0 ||
        argfd(4, &fd, &f) < 0 || argint(5, &offset) < 0 ||
        argint(1, &length) < 0 || argptr(0, (char **)&addr) < 0) {
        return (void *)-1;
```

```
    }

    // 查找未使用的地址空间区域
    // 添加 VMA
    // 增加文件的引用计数

    return addr;
}
```

3. 实现页错误处理

1. **懒惰加载**：
   - 在页错误处理代码中填充页表。这意味着 `mmap` 不应该分配物理内存或读取文件。相反，这些操作应在页错误处理中完成。

```
// kernel/trap.c
void usertrap(void) {
    // 处理页错误
    // 如果是 mmap 的区域，分配物理内存，读取文件内容到页
}
```

2. **读取文件**：
   - 使用 `readi` 读取文件数据到页中，并设置页的权限。

```
// kernel/vm.c
void handle_page_fault(void *addr) {
    // 读取文件数据到页
    // 更新页权限
}
```

4. 实现 `munmap`

1. **实现 `munmap`**：
   - 查找地址范围内的 VMA，并解除映射。如果 `munmap` 移除了先前 `mmap` 的所有页，则应减少相应文件的引用计数。如果页被修改且文件映射为 `MAP_SHARED`，则将页面写回文件。

```
// kernel/sysfile.c
int sys_munmap(void) {
    void *addr;
    size_t length;
    struct proc *p = myproc();

    if (argint(1, &length) < 0 || argptr(0, (char **)&addr) < 0)
```

```
        return -1;

    // 查找 VMA
    // 解除映射
    // 如果需要，写回文件

    return 0;
}
```

2. **解除映射**：
    - 使用 `uvmunmap` 解除映射区域。确保在 `munmap` 中减少文件引用计数并处理映射区域的所有页。

```
// kernel/vm.c
void unmap_vma(struct vma *vma) {
    // 解除映射
    // 处理页
}
```

5. 实现 `fork` 和 `exit` 的支持

1. **修改 `fork`**：
    - 确保子进程具有与父进程相同的映射区域。增加 VMA 的文件引用计数。

```
// kernel/proc.c
int fork(void) {
    // 复制进程映射区域
    // 增加文件引用计数
    return 0;
}
```

2. **修改 `exit`**：
    - 在进程退出时解除所有映射区域。

```
// kernel/proc.c
void exit(void) {
    // 解除映射区域
}
```

6. 添加测试

1. **编写测试程序 `mmaptest`**：
    - 确保 `mmaptest` 包含各种 `mmap` 和 `munmap` 的测试场景。

```
// user/mmaptest.c
int main(void) {
    // 编写测试代码
    return 0;
}
```

2. **运行测试**：
    - 编译并运行 `mmaptest` 以验证 `mmap` 和 `munmap` 的实现。

```
make mmaptest
./mmaptest
```

3. **运行所有测试**：
    - 使用 `usertests` 运行所有测试，确保所有功能正常工作。

```
usertests
```

3. 实验中遇到的困难和解决办法

1. **页错误处理的复杂性**：
    - **困难描述**：处理 `mmap` 区域的页错误可能会非常复杂。
    - **解决办法**：确保仔细阅读 `usertrap` 函数，理解如何处理页错误并正确加载文件数据。
2. **VMA 的管理**：
    - **困难描述**：管理进程的 VMA 可能会引发多种问题，如区域重叠和引用计数管理。
    - **解决办法**：使用固定大小的数组来管理 VMA，并确保正确处理引用计数。
3. **文件和页的同步**：
    - **困难描述**：确保 `MAP_SHARED` 页的修改被正确写回文件，处理脏页。
    - **解决办法**：利用 RISC-V 页表的脏位来判断页面是否被修改，并在 `munmap` 时正确写回。

4. 实验心得

通过本次实验，我深入理解了内存映射文件的实现细节，包括页错误处理、VMA 管理以及 `mmap` 和 `munmap` 的系统调用。我学会了如何在 xv6 中处理复杂的内存管理任务，这对于理解现代操作系统中的内存管理机制非常重要。这些技能对于构建和优化操作系统功能至关重要。