同济大学操作系统课程设计 - Xv6 - Lab: mmap

【实验目的】

通过实现 `mmap` 和 `munmap`，深入理解操作系统如何管理虚拟内存。这包括如何将文件映射到进程的虚拟地址空间，如何处理页面错误，以及如何有效地进行内存分配和回收。

系统调用是操作系统和用户程序之间的接口。通过实现这些系统调用可以学习到如何在内核中实现和管理系统调用，包括处理用户空间与内核空间的数据传递、系统调用的注册和调度等。

`mmap` 和 `munmap` 涉及到的内存映射和进程管理，帮助理解如何在进程间共享内存以及如何在进程创建、退出时正确管理内存。这对理解操作系统的进程管理和内存分配策略至关重要。

学习如何在内核空间编写和调试代码。内核编程与用户空间编程有很大不同，这种实践经验对深入理解操作系统的工作原理非常重要。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 mmap 分支：

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

1. 在 `Makefile` 中添加 `mmaptest` 的编译规则

首先，确保在 `Makefile` 文件中添加对 `mmaptest` 的编译规则。这将确保在运行 `make` 命令时，`mmaptest` 被正确编译，并包含在生成的可执行文件中。

2. 添加 `mmap` 和 `munmap` 系统调用的定义与实现

`kernel/syscall.h`：

在 `syscall.h` 中添加 `mmap` 和 `munmap` 系统调用的定义。这允许用户空间通过系统调用与内核交互。

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

`kernel/syscall.c`：

在 `syscall.c` 中，添加 `mmap` 和 `munmap` 系统调用的函数声明。

```
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,
};
```

`user/usys.pl`:

在 `usys.pl` 文件中添加 `mmap` 和 `munmap` 系统调用的用户空间接口。

```
syscall mmap;
syscall munmap;
```

`user/user.h`:

在 `user.h` 中声明 `mmap` 和 `munmap` 函数的原型。

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

`kernel/defs.h`:

在 `defs.h` 文件中，确保包含 `mmap` 和 `munmap` 的声明。

```
void initmmap(void);
int sys_mmap(void);
int sys_munmap(void);
```

3. 定义 `struct vma` 结构体，并在 `struct proc` 中添加相关字段

`kernel/proc.h`:

定义 `struct vma` 结构体，并在 `struct proc` 中添加 `vma` 数组，用于记录每个进程的虚拟内存区域。

```
#define MAXVMA 64

struct vma {
    //...
    int len;
    struct file *file;
    int flags;
};

struct proc {
    // ... 其他字段
    struct vma vma[MAXVMA];
};
```

4. 实现 `sys_mmap()` 系统调用

`kernel/sysfile.c`:

实现 `sys_mmap()` 系统调用，用于映射文件到进程的虚拟地址空间。

```
int sys_mmap(void) {
    void *addr;
    int len, prot, flags, fd;
    off_t offset;
    struct file *f;
    struct vma *vma;
    int i;

    if (argint(0, &fd) < 0 || argint(1, &len) < 0 || argint(2, &prot) < 0 ||
        argint(3, &flags) < 0 || argint(4, &offset) < 0 || argptr(5, (void*)&addr) < 0)
{
        return -1;
    }

    if (flags & MAP_SHARED && flags & MAP_PRIVATE) {
        return -1; // Invalid flags
    }

    f = filedup(fd); // Increase file descriptor reference count
    if (f == 0) {
        return -1;
    }

    // Allocate and initialize a new VMA
    for (i = 0; i < MAXVMA; i++) {
        if (proc->vma[i].addr == 0) {
            vma = &proc->vma[i];
            break;
        }
    }

    if (i == MAXVMA) {
        fileclose(f);
        return -1; // No available VMA slots
    }

    // Set up the VMA
    vma->addr = addr;
    vma->len = len;
    vma->file = f;
    vma->flags = flags;
    // Additional setup...

    return (int)addr; // Return the address where the file is mapped
}
```

5. 处理 `mmap` 的页错误

`kernel/trap.c`:

在 `usertrap()` 中处理页错误，以便惰性分配物理页面。

```
void usertrap(void) {
    // ... 处理其他陷阱
    if (r_scause() == 13 || r_scause() == 15) {
        // Page fault
        // Handle page fault for mmap areas
        // Find VMA, allocate page, read from file, and update page table
    }
}
```

6. 实现 `munmap()` 系统调用

`kernel/sysfile.c`:

实现 `munmap()` 系统调用，用于取消映射区域。

```
int sys_munmap(void) {
    void *addr;
    int len;
    struct vma *vma;
    int i;

    if (argint(0, &len) < 0 || argptr(1, (void*)&addr) < 0) {
        return -1;
    }

    // Find and remove VMA
    for (i = 0; i < MAXVMA; i++) {
        if (proc->vma[i].addr == addr) {
            vma = &proc->vma[i];
            // Unmap pages, write back if necessary, update VMA
            // ...
            return 0;
        }
    }

    return -1; // VMA not found
}
```

7. 修改 `exit()` 和 `fork()` 函数

`kernel/proc.c`:

在 `exit()` 函数中，取消映射进程的所有映射区域。在 `fork()` 函数中，复制父进程的 VMA 到子进程。

```
void exit(void) {
    // ... 处理进程退出
    for (int i = 0; i < MAXVMA; i++) {
        if (proc->vma[i].addr != 0) {
```

```
            // Unmap VMA, write back if necessary
        }
    }
    // ...
}

int fork(void) {
    // ... 复制父进程
    for (int i = 0; i < MAXVMA; i++) {
        if (proc->vma[i].addr != 0) {
            // Copy VMA to child process
        }
    }
    // ...
}
```

8. 修改 `uvmcopy()` 函数

`kernel/vm.c`:

在 `uvmcopy()` 中，确保子进程的页面映射正确地复制父进程的 VMA 信息。

```
void uvmcopy(pde_t *old, pde_t *new) {
    // ... 复制页面映射
    // Copy VMA information
}
```

9. 运行测试

使用 `make qemu` 命令运行 XV6 并测试新实现的 `mmap` 和 `munmap` 系统调用。

```
make qemu
```

10. 运行 `mmaptest`

在命令行中执行 `mmaptest` 程序，以验证 `mmap` 和 `munmap` 系统调用的实现。

```
mmaptest
```

```
hart 2 starting
hart 1 starting
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
$
```

【分析讨论】

在完成 `mmap` 和 `munmap` 系统调用的实现过程中，我获得了一些深刻的体会和启示，这些体会可以帮助我更好地理解操作系统的设计和实现。

实现 `mmap` 和 `munmap` 系统调用涉及到操作系统的多个核心部分，包括系统调用接口、内存管理、页面错误处理等。这些系统调用的实现不仅需要对内核内部机制有深入了解，还需要仔细设计和处理各种边界情况。这让我更加认识到操作系统设计的复杂性以及编写高效、稳定代码的重要性。

通过实现 `mmap`，我学会了如何将文件映射到进程的虚拟地址空间。这不仅涉及到内存的分配和映射，还需要处理文件的读写权限和共享策略。 `mmap` 的实现展示了内存管理的灵活性和强大功能，同时也强调了需要精确控制和管理虚拟内存的必要性。

在处理 `mmap` 时，我深刻理解了惰性分配的概念。这种方法可以有效地管理内存，避免了在内存映射时对所有页面的即时分配，而是在实际访问时才进行分配。这种策略需要在页面错误处理（page fault）中做精确的处理，以确保系统的稳定性和性能。学习如何在页面错误处理中分配物理内存并从文件中读取数据是我在这次实验中的一个重要收获。

`mmap` 和 `munmap` 的实现还涉及到进程管理。在进程退出或复制时，需要正确处理进程的虚拟内存区域（VMA），确保资源的正确释放和复制。这让我认识到进程内存管理在操作系统中的重要性，以及如何在多进程环境中维护和管理进程的内存映射关系。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== 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 (310.1s)
== Test time ==
time: OK
Score: 140/140
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab10$
```