同济大学操作系统课程设计 - Xv6 - Lab: file system

【实验目的】

- 通过实现符号链接（`symlink`）系统调用，能够学习并理解文件系统的核心组件，如 `inode` 结构、数据块的管理，以及文件系统的扩展和维护。
- 实践如何在操作系统中添加新的系统调用，并实现相应的内核函数。这有助于理解用户空间与内核空间的交互过程，以及系统调用如何作为接口实现。
- 在实现过程中涉及到对文件的锁定和同步操作，认识到并发控制的重要性，学习如何使用锁机制确保数据一致性和系统稳定性。

【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 fs分支：

```
$ git fetch
$ git checkout fs
$ make clean
```

1 Large files

在原始 xv6 操作系统中，文件系统的设计较为简洁，但文件的最大尺寸受限于 inode 结构的设计。为了支持更大的文件，我们可以将现有的单层间接块扩展为双层间接块，从而大幅提高文件系统的存储能力。以下是具体的步骤和实现细节。

1. 查找并修改常量定义

首先，打开 `kernel/fs.h` 文件，查找并确认 `NDIRECT` 和 `NINDIRECT` 的定义。这两个常量分别表示直接块和单间接块的数量。

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
```

原始的 xv6 inode 包含 12 个直接数据块号和 1 个单间接数据块号。单间接数据块最多可以包含 256 个数据块号，因此一个 xv6 文件最多只能包含 268 个数据块。为了支持更大的文件，我们需要将其中的一个直接块号替换为双层间接块号。双层间接块可以容纳 256 个间接块，每个间接块又可以容纳 256 个数据块号。因此，一个文件最多可以包含 65,803 个数据块。

我们通过以下方式定义双层间接块的容量：

```
#define NDBL_INDIRECT (NINDIRECT * NINDIRECT)
```

2. 更新 inode 和 dinode 结构体

接下来，我们需要在 `kernel/file.h` 和 `kernel/fs.h` 文件中更新 `struct inode` 和 `struct dinode` 结构体。具体而言，需要修改 `addrs[]` 数组的大小，以支持双层间接块。

```
struct dinode {
  //...
  uint addrs[NDIRECT+2];  // 增加一个用于存储双间接块的地址
};
```

此修改使得 `addrs[]` 数组能够存储直接块、单层间接块和双层间接块的地址。

3. 修改 `bmap` 函数以支持双层间接映射

`bmap` 函数负责将文件的逻辑块号映射到磁盘块号。为了支持双层间接块，我们需要在 `kernel/fs.c` 文件中添加相应的逻辑。

```
if (bn < NDBL_INDIRECT) {

  if ((addr = ip->addrs[NDIRECT+1]) == 0)
    ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
  inbp = bread(ip->dev, addr);
  a = (uint*)inbp->data;
  if ((addr = a[bn/NINDIRECT]) == 0) {
    a[bn/NINDIRECT] = addr = balloc(ip->dev);
    log_write(inbp);
  }
  brelse(inbp);
  ininbp = bread(ip->dev, addr);
  b = (uint*)ininbp->data;
  if ((addr = b[bn % NINDIRECT]) == 0) {
    b[bn % NINDIRECT] = addr = balloc(ip->dev);
    log_write(ininbp);
  }
  brelse(ininbp);
  return addr;
}
panic("bmap: out of range");
```

上述代码中增加了对双层间接块的处理逻辑，当逻辑块号大于 `NDIRECT` 加上 `NINDIRECT` 时，系统会将其映射到双层间接块中的数据块。

4. 修改 `itrunc` 函数以支持双层间接块的清除

在文件被删除或截断时，`itrunc` 函数负责释放文件占用的磁盘块。为了确保双层间接块中的数据也能够被正确释放，我们在 `kernel/fs.c` 文件中更新 `itrunc` 函数。

```
void itrunc(struct inode *ip)
{
  int i, j, k;
  struct buf *bp, *inbp;
  uint *a, *b;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }
  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }
  if(ip->addrs[NDIRECT+1]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a = (uint*)bp->data;
    for (j = 0; j < NINDIRECT; j++) {
      if (a[j]) {
        inbp = bread(ip->dev, a[j]);
        b = (uint*)inbp->data;
        for (k = 0; k < NINDIRECT; k++) {
          if (b[k])
            bfree(ip->dev, b[k]);
        }
        brelse(inbp);
        bfree(ip->dev, a[j]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
  }
  ip->size = 0;
  iupdate(ip);
}
```

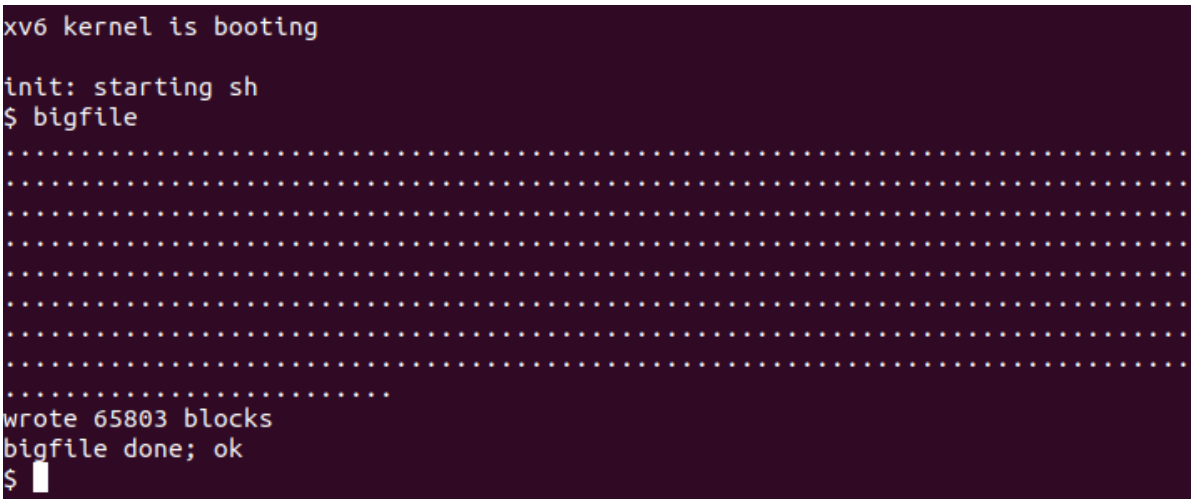
此代码中新增的逻辑确保双层间接块中的所有数据块在文件被截断时都能被释放。

5. 编译并测试

完成以上修改后，通过 `make qemu` 命令编译并运行 xv6 系统。在 xv6 终端中输入 `bigfile` 命令，测试文件系统对大文件的支持。

```
make qemu
bigfile
```

测试结果应显示文件系统能够成功创建和管理大文件，并且双层间接块工作正常：



2 Symbolic links

1. 添加 `symlink` 系统调用的声明

为了在内核中添加一个新的系统调用，我们首先需要在多个文件中声明 `symlink` 系统调用。

1. **在 `kernel/syscall.h` 中声明系统调用编号：**

   ```
   // syscall.h 文件
   #define SYS_symlink 22  // 为 symlink 系统调用分配一个系统调用编号
   ```

   该文件负责定义所有系统调用的编号。在这里，我们为 `symlink` 分配了编号 22。

2. **在 `kernel/syscall.c` 中添加系统调用处理函数：**

   ```
   extern uint64 sys_symlink(void);  // 声明 sys_symlink 函数

   static int (*syscalls[])(void) = {
       // 其他系统调用函数...
       [SYS_symlink]   sys_symlink,  // 将 sys_symlink 函数添加到系统调用表
   };
   ```

   这里，我们将 `sys_symlink` 函数添加到 `syscalls` 数组中，使得 xv6 内核在接收到 `symlink` 系统调用请求时能够正确处理。

3. **在 `user/usys.pl` 中生成用户态调用接口：**

   ```
   entry("symlink");
   ```

`usys.pl` 脚本用于生成用户态的系统调用接口。通过添加 `entry("symlink")`，我们可以在用户态直接调用 `symlink` 函数。

4. **在 `user/user.h` 中声明用户态函数：**

```
int symlink( char *, char *);
```

该声明使得用户态程序可以调用 `symlink` 函数，并将目标路径与符号链接路径作为参数传递。

2. 添加新文件类型 `T_SYMLINK`

为了区分普通文件和符号链接文件，我们需要在 `kernel/stat.h` 中添加一个新的文件类型：

```
#define T_SYMLINK  4  // 新增符号链接文件类型
```

`T_SYMLINK` 表示该文件是一个符号链接，而不是普通文件或目录。

3. 添加 `O_NOFOLLOW` 标志

符号链接文件的处理方式可以通过 `O_NOFOLLOW` 标志来定制。我们在 `kernel/fcntl.h` 中添加这个标志：

```
#define O_NOFOLLOW 0x800  // 新增 O_NOFOLLOW 标志
```

如果在调用 `open` 函数时使用 `O_NOFOLLOW` 标志，系统将打开符号链接文件本身，而不是跟随符号链接指向的目标文件。

4. 实现 `sys_symlink` 系统调用

`sys_symlink` 系统调用的主要工作是创建一个符号链接文件，并将目标路径写入符号链接文件的数据块中。在 `kernel/sysfile.c` 文件中实现该函数：

```
uint64
sys_symlink(char *target, char *path) {
  char kpath[MAXPATH], ktarget[MAXPATH];
  memset(kpath, 0, MAXPATH);
  memset(ktarget, 0, MAXPATH);
  struct inode *ip;
  int n, r;

  if((n = argstr(0, ktarget, MAXPATH)) < 0)
    return -1;

  if ((n = argstr(1, kpath, MAXPATH)) < 0)
    return -1;
  int ret = 0;
  begin_op();

  if((ip = namei(kpath)) != 0){
    ret = -1;
    goto final;
  }
  ip = create(kpath, T_SYMLINK, 0, 0);
  if(ip == 0){
```

```
    ret = -1;
    goto final;
  }
  if ((r = writei(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0)
    ret = -1;
  iunlockput(ip);

final:
  end_op();
  return ret;
}
```

5. 修改 `open` 系统调用

为了支持符号链接文件的打开操作，我们需要在 `kernel/sysfile.c` 中修改 `open` 系统调用的实现，使其能够处理符号链接：

```
int depth = 0;
while (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  char ktarget[MAXPATH];
  memset(ktarget, 0, MAXPATH);
  if ((r = readi(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlockput(ip);
  if((ip = namei(ktarget)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  depth++;
  if (depth > 10) {
    iunlockput(ip);
    end_op();
    return -1;
  }
}
```

这里的关键点在于，如果打开的文件是符号链接且没有设置 `O_NOFOLLOW` 标志，系统将递归解析符号链接指向的目标路径并打开目标文件。

6. 添加测试程序并编译

为了测试 `symlink` 系统调用的正确性，我们在 `Makefile` 中添加对测试程序 `symlinktest.c` 的编译：

```
UPROGS=\
        $U/_symlinktest\
```

这个测试程序会调用 `symlink` 系统调用，并验证符号链接的创建和打开功能。

7. 运行测试

最后，通过执行 `make qemu` 来编译并运行 xv6。在 qemu 终端中运行测试程序 `symlinktest`：

```
make qemu
symlinktest
```

测试结果应显示符号链接创建和打开的功能是否正常工作：

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab9$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
 1 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$
```

【分析讨论】

通过本实验，我成功地扩展了 xv6 文件系统的文件大小限制，掌握了文件系统核心模块的设计与实现。在真实操作系统开发中，这样的能力能够帮助你优化系统性能，提升系统的可扩展性和稳定性。我还成功地为 xv6 添加了 `symlink` 系统调用，支持符号链接文件的创建和处理。这一扩展增强了 xv6 的文件系统功能，提升了系统的灵活性与可操作性。在实际操作系统开发中，理解并实现类似的功能扩展能够帮助开发者更深入地掌握文件系统的设计原理和操作机制。

在实现 `symlink` 的过程中，我不仅学习了如何扩展文件系统的功能，还深入理解了 inode 结构、数据块管理、文件类型的区分等底层细节。这些知识对掌握操作系统的核心原理至关重要。处理符号链接时，需要递归地解析路径并进行相应的文件操作。这种递归思想在操作系统开发中非常常见，也是一种处理复杂逻辑的有效方式。

系统调用是操作系统与应用程序之间的接口，通过这次实践，我学会了如何在内核中添加新的系统调用，并理解了从用户态到内核态的参数传递和错误处理的细节。这种设计思路不仅适用于操作系统，也能扩展到其他复杂系统的设计中。在操作 inode 时，我们需要确保线程安全，避免出现数据竞态条件。这让我认识到锁机制在操作系统中的重要性，良好的并发控制对系统稳定性至关重要。

在实现过程中，我不仅关注单个函数的实现，还必须考虑整个文件系统的整体架构和相互依赖。这种宏观与微观相结合的思考方式，对解决复杂问题非常有帮助。

【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：

```
make[1]: Leaving directory '/home/yzz/Desktop/xv6/xv6-2021/lab9'
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (143.0s)
== Test running symlinktest ==
$ make qemu-gdb
(0.7s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (224.9s)
== Test time ==
time: OK
Score: 100/100
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab9$
```