Lab9 File system

Sublab1 Large Files

一、环境搭建

本实验在 xv6-labs-2021 的 `fs` 分支中进行。首先执行 `git fetch` 和 `git checkout fs` 切换分支，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。实验中会涉及到对 xv6 内核源码的修改，特别是在文件系统部分。需要注意的是，文件系统的大小由 `FSSIZE` 在 `kernel/param.h` 中定义，实验中该值为 200,000 块。在修改完文件系统代码后，可以通过 `make qemu` 来启动模拟器，运行修改后的 xv6 系统。

二、实验目的

本实验旨在修改 xv6 文件系统以支持更大的文件。在原始的 xv6 文件系统中，文件大小的上限为 268 块，这一限制来自于文件的 inode 结构。在本实验中，通过在 inode 中加入“双重间接”块，我们将文件的最大大小扩展至 65803 块。此外，还将通过实现大文件的创建命令 `bigfile`，验证文件系统扩展的有效性。

三、实验内容

- 切换到 `fs` 分支，准备新的实验代码环境：

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

- 修改 `bmap()` 函数，增加对双重间接块的支持：
  - 在 `fs.h` 中修改 `struct dinode`，将原有的 12 个直接块、1 个单重间接块扩展为 11 个直接块、1 个单重间接块和 1 个双重间接块。
  - 修改 `bmap()` 函数，增加对双重间接块的处理逻辑。需要根据文件的逻辑块号来判断是否需要使用双重间接块，将逻辑块号映射到正确的磁盘块号。
  - 在 `addrs[]` 数组中，第 12 个元素存储单重间接块，第 13 个元素存储双重间接块。
- 修改文件系统的结构，使其支持更大的文件：
  - 将文件系统的最大文件块数增加至 65803 块（256*256 + 256 + 11），通过调整 `bmap()` 中的块映射逻辑，实现对双重间接块的支持。
  - 使用 `bigfile` 命令验证修改后的文件系统是否支持创建更大的文件。
- 重新生成文件系统映像：
  - 运行 `make fs.img` 重新生成文件系统映像。
  - 使用 `bigfile` 命令创建一个 65803 块大小的文件，确保文件系统扩展成功。
  - 运行 `usertests` 确保所有测试通过：

四、实验结果分析

执行 `bigfile` 命令后，输出显示成功写入 65803 块，证明文件系统的最大文件大小已成功扩展。在完成文件系统扩展后，`usertests` 运行成功，所有测试通过，表明修改后的文件系统工作正常。通过本实验，验证了对文件系统进行的扩展有效地支持了大文件的创建和管理。

```
init: starting sh
$ bigfile
..........................................................................
..........................................................................
..........................................................................
..........................................................................
wrote 65803 blocks
bigfile done; ok
$
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

五、实验中遇到的问题及解决方法

在实现双重间接块时，遇到了块号计算和内存分配的问题。在初期实现时，误将文件的逻辑块号映射到错误的物理块，导致文件大小未能达到预期。通过仔细分析 `bmap()` 函数的实现逻辑并逐步调试，成功解决了该问题。最终实现了对双重间接块的正确处理，并成功扩展了文件系统的最大文件大小

六、实验心得

本实验使我深入理解了文件系统的内部结构以及如何扩展文件系统的容量。通过修改 `bmap()` 函数来支持双重间接块，学习了如何在保持 inode 结构不变的情况下扩展文件系统的容量。在实现过程中，我还体会到了文件系统在管理文件块时需要的精确计算和内存管理技巧，特别是在涉及到间接块和双重间接块时。此外，本实验通过创建和测试大文件，增强了我对操作系统文件系统设计的理解，为后续更复杂的文件系统设计奠定了基础。

附：实验源码

```
// 修改后的 bmap 函数片段，支持双重间接块
uint64
bmap(struct inode *ip, uint bn)
{
  uint addr, *a, *b;

  // 判断是否使用直接块
  if(bn < NDIRECT){
    addr = ip->addrs[bn];
  }
  // 判断是否使用单重间接块
  else if(bn < NDIRECT + NINDIRECT){
    if(ip->addrs[NDIRECT] == 0)
      ip->addrs[NDIRECT] = balloc(ip->dev);
    a = (uint*)bread(ip->dev, ip->addrs[NDIRECT]);
    addr = a[bn - NDIRECT];
    brelse(a);
  }
  // 判断是否使用双重间接块
  else{
    if(ip->addrs[NDIRECT + 1] == 0)
      ip->addrs[NDIRECT + 1] = balloc(ip->dev);
    a = (uint*)bread(ip->dev, ip->addrs[NDIRECT + 1]);
    if(a[(bn - NDIRECT - NINDIRECT) / NINDIRECT] == 0)
      a[(bn - NDIRECT - NINDIRECT) / NINDIRECT] = balloc(ip->dev);
    b = (uint*)bread(ip->dev, a[(bn - NDIRECT - NINDIRECT) / NINDIRECT]);
```

```
27      addr = b[(bn - NDIRECT - NINDIRECT) % NINDIRECT];
28      brelse(b);
29      brelse(a);
30    }
31    return addr;
32  }
```

Sublab2 Symbolic Links

一、环境搭建

本实验在 xv6-labs-2021 的 `fs` 分支中进行。首先执行 `git fetch` 和 `git checkout fs` 切换分支，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。实验中将涉及到对 xv6 内核源码的修改，特别是在文件系统部分。实验的目标是添加对符号链接（symbolic links）的支持。

二、实验目的

本实验的目的是实现符号链接（symlink）的功能。符号链接是指通过路径名引用另一个文件，打开符号链接时，内核会跟随该链接指向目标文件。与硬链接不同，符号链接可以跨越设备进行链接。在 xv6 中，符号链接的实现将加深对路径查找、文件系统操作及系统调用的理解。实验的最终目标是能够创建符号链接并正确处理符号链接的相关操作。

三、实验内容

- 添加 `symlink` 系统调用：
  - 为 `symlink` 创建一个新的系统调用编号，并将其添加到 `user/usys.pl` 和 `user/user.h` 中。
  - 在 `kernel/sysfile.c` 中实现 `sys_symlink()`，该函数将创建一个符号链接。
- 修改文件类型定义：
  - 在 `kernel/stat.h` 中添加一个新的文件类型 `T_SYMLINK`，用于表示符号链接。
- 添加 `O_NOFOLLOW` 标志：
  - 在 `kernel/fcntl.h` 中添加一个新的标志 `O_NOFOLLOW`，该标志与 `open` 系统调用一起使用，以便在打开符号链接时不自动跟随。
- 实现符号链接的创建：
  - `symlink` 系统调用将接受两个参数：`target` 和 `path`。`target` 表示符号链接指向的目标文件，`path` 是符号链接的路径。
  - 将符号链接的目标路径存储在 inode 的数据块中，以便后续查找。
- 修改 `open` 系统调用，支持符号链接：
  - 在 `open` 系统调用中，检查路径是否是符号链接。如果路径是符号链接，且未指定 `O_NOFOLLOW` 标志，则递归跟随符号链接，直到找到非符号链接的目标文件。
  - 如果遇到符号链接形成的循环（例如符号链接指向自身或其他符号链接形成闭环），应返回错误。
- 测试：
  - 在 `Makefile` 中添加 `symlinktest`，运行该测试程序验证符号链接功能。

四、实验结果分析

在执行 `symlinktest` 后，输出显示符号链接的创建和解析功能正常工作。测试包括了符号链接的基本创建、跟随和递归解析等情况，均按预期完成。特别是在符号链接循环的检测和错误处理方面，实验实现了对符号链接深度的限制，防止无限循环的发生。最终，通过 `usertests`，所有测试都顺利通过，证明了符号链接的功能实现没有问题。

```
xv6 kernel is booting

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

```
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x0000000000004562 stval=0x0000000000014000
OK
test sbrkarg: OK
test sbrklast: OK
test sbrk8000: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x00000000000023b8 stval=0x0000000000011b50
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 bigfile ==
$ make qemu-gdb
running bigfile: OK (130.1s)
== 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 (200.7s)
== Test time ==
time: OK
Score: 100/100
```

五、实验中遇到的问题及解决方法

在实现符号链接的递归跟随时，遇到了一些路径查找和符号链接深度的问题。特别是在处理符号链接指向其他符号链接的情况下，容易出现无限递归的问题。通过限制符号链接的最大深度（如 10 层），成功避免了这种情况。此外，在实现 `O_NOFOLLOW` 标志时，确保了在打开符号链接时不会自动跟随，避免了不必要的链接解析。

六、实验心得

通过本实验，我深入理解了符号链接的工作原理及其在文件系统中的实现。特别是在实现符号链接的递归解析和循环检测方面，我对文件系统中路径查找和文件操作有了更深的认识。符号链接不仅仅是一个简单的路径重定向，它涉及到对文件系统结构的修改，以及如何处理符号链接的创建、删除、打开等操作。实验的实现让我更加熟悉了文件系统的内部机制，也为未来深入研究操作系统的文件管理提供了有价值的经验。

附：实验源码

```
int
sys_symlink(void)
{
  char *target, *path;
  struct inode *ip;

  // 获取系统调用参数
  if(argstr(0, &target) < 0 || argstr(1, &path) < 0)
    return -1;

  // 创建符号链接
  begin_op();
  if((ip = create(path, T_SYMLINK, 0, 0)) == 0){
    end_op();
    return -1;
  }

  // 将目标路径存储到 inode 数据块中
  strncpy(ip->addrs[0], target, sizeof(ip->addrs[0]));

  // 释放 inode 并结束操作
```

```
  iunlockput(ip);
  end_op();

  return 0;
}

// 在 open 函数中处理符号链接
int
open(char *path, int omode)
{
  struct inode *ip;
  int fd;

  if((ip = namei(path)) == 0)
    return -1;

  // 如果是符号链接并且没有设置 O_NOFOLLOW，跟随链接
  if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
    char target[DIRSIZ];
    readi(ip, target, 0, sizeof(target));
    iunlockput(ip);
    return open(target, omode);  // 递归打开目标文件
  }
}
```