Lab: file system

Git链接：[Jeery1/xv6-6.S081 at fs](https://github.com/Jeery1/xv6-6.S081/tree/fs)

**实验目的：**

1. 理解文件系统的基本结构

磁盘布局：学习 xv6 文件系统的磁盘组织方式，包括：

超级块（Superblock）：存储文件系统元数据（如块数、inode 数）。

inode 表：管理文件元信息（类型、大小、数据块指针）。

数据块区域：存储实际文件内容。

文件抽象：理解文件描述符（struct file）、inode（struct inode）和目录项（struct dirent）之间的关系。

2. 实现文件系统操作

文件读写：分析 readi() 和 writei() 如何通过 bmap() 将文件偏移映射到磁盘块。

目录管理：实现目录的创建、查找和遍历（如 dirlookup()、dirlink()）。

符号链接：扩展文件系统以支持符号链接（T\_SYMLINK），并修改 open() 和 exec() 解析链接。

3. 优化文件系统性能

分析 bread()/bwrite() 如何减少磁盘 I/O。

优化锁策略（如分桶锁）以提高多核并发性能。理解事务（Transaction）如何保证崩溃一致性。

分析 begin\_op()、log\_write() 和 end\_op() 的协作机制。

4. 扩展文件系统功能

大文件支持：修改 bmap() 实现二级间接块，突破 xv6 默认的 268KB 文件大小限制。

硬链接：实现 link() 系统调用，允许多个目录项指向同一 inode。

文件权限：扩展 inode 结构，支持简单的读写权限控制。

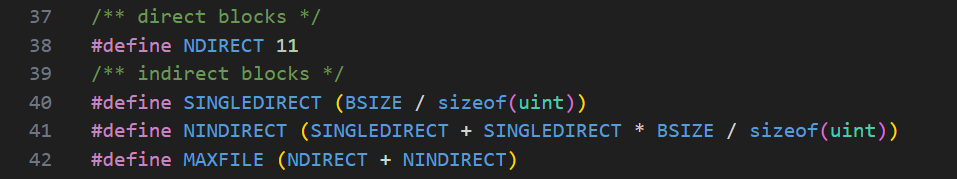
**实验步骤：**

**Large files**

xv6中的inode有12个直接索引，1个一级索引。另外xv6的数据块大小为1024B，索引指针大小为4B，因此一个数据块可以装256个索引指针，因此，目前一个inode最大支持12 + 256 = 268个数据块。需要注意的是，在xv6中有两种类型的inode，其中一个是存在磁盘中的inode，另一种是缓存在内存中的inode，在对inode结构进行修改的时候需要修改这两处。

本次任务即为xv6中的inode添加一个Doubel Indirect索引块，这样每一个inode就能够支持11 + 256 + 256 \* 256 = 65803个数据块的文件了。

我们首先修改宏定义如下：



其中，NDIRECT代表直接索引指向块的数量；SINGLEDIRECT表示一级索引能够指向的块的数量；NINDIRECT索引代表一级和二级索引一共可指向的块的数量。

其次，根据指导书的Hint：

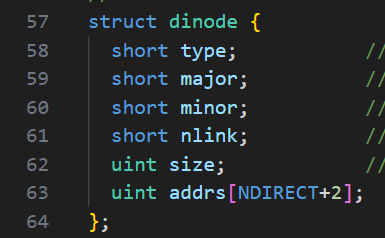
mkfs initializes the file system to have fewer than 2000 free data blocks, too few to show off the changes you’ll make. Modify kernel/param.h to change FSSIZE from 2000 to 200,000:

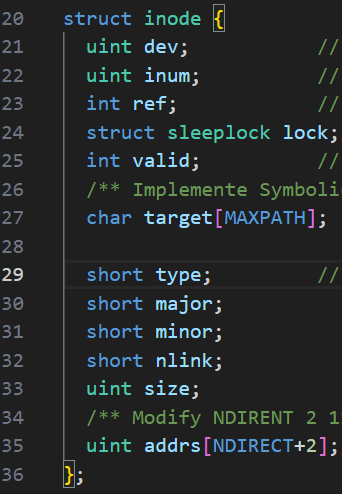
Rebuild mkfs so that is produces a bigger disk: $ rm mkfs/mkfs fs.img; make mkfs/mkfs

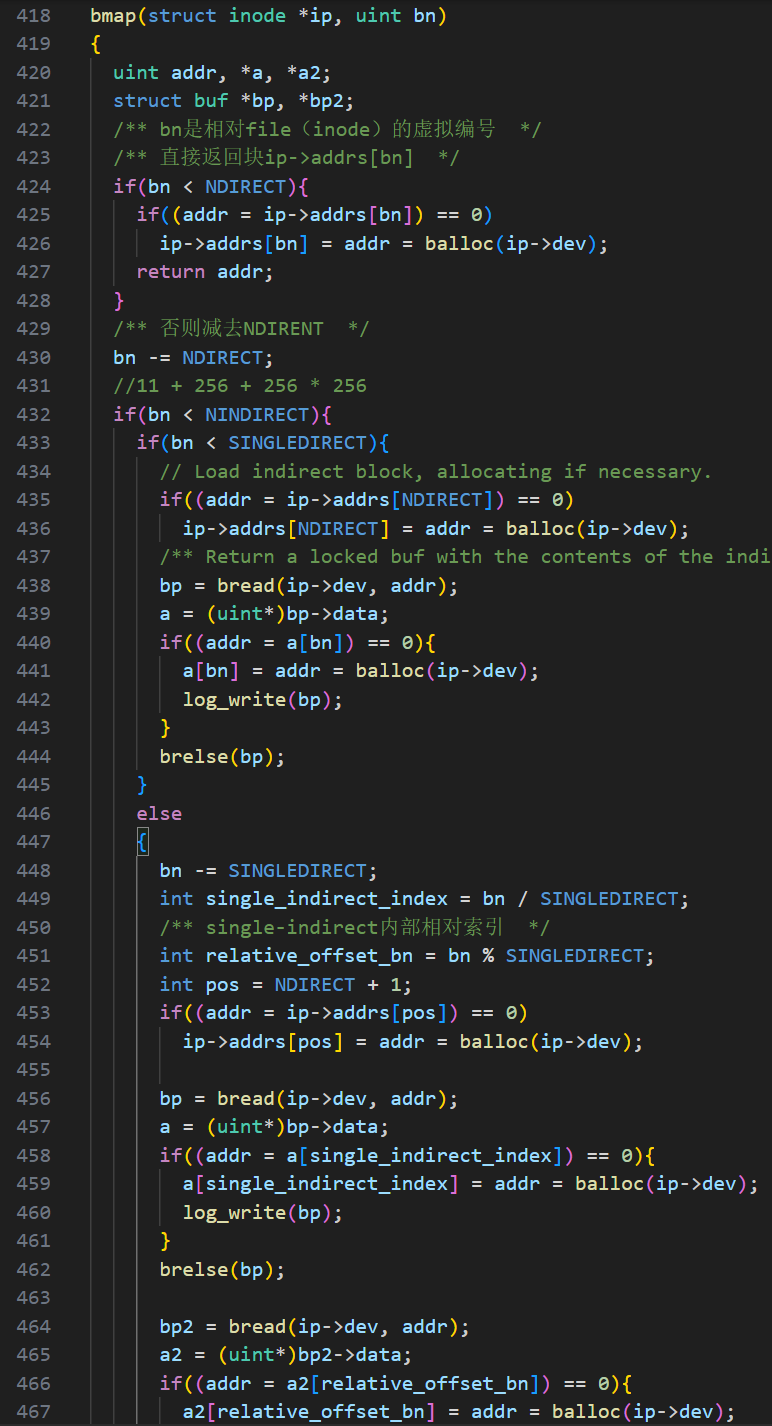
我们修改kernel/param.h中的FSSIZE宏定义为200000



然后我们修改inode的定义，注意要修改两处，分别对应缓存在内存中的，和存在磁盘中的。







修改bmap函数

输入：ip：文件的 inode 指针。bn：文件的逻辑块号（从 0 开始）。

输出：返回磁盘上的物理块号（uint）。如果目标块未分配，则调用 balloc() 分配新块并更新 inode。

直接块：  
如果 bn < NDIRECT（通常 NDIRECT=12），直接从 ip->addrs[bn] 获取物理块号。

如果未分配（addr == 0），调用 balloc() 分配新块并更新 inode。

一级间接块：  
从 ip->addrs[NDIRECT] 获取一级间接块的物理地址。

读取该间接块（bread），其内容是一个数组，存储实际数据块的地址。

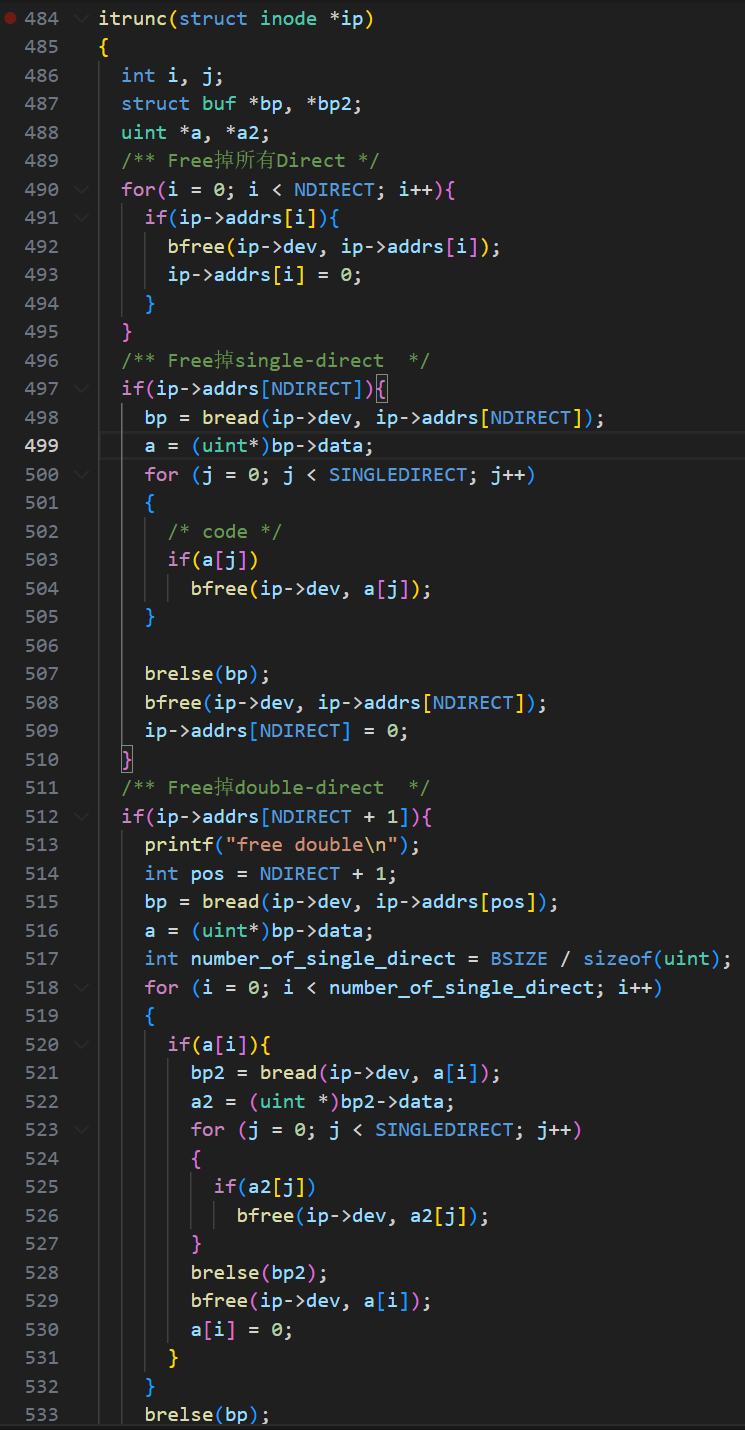
如果目标块未分配（a[bn] == 0），分配新块并写回磁盘（log\_write）。

二级间接块：  
从 ip->addrs[NDIRECT+1] 获取二级间接块的物理地址。

读取二级间接块，找到一级间接块的索引（single\_indirect\_index）。

读取一级间接块，找到数据块的索引（relative\_offset\_bn）。

如果任何块未分配，动态分配并写回磁盘。



输入：struct inode \*ip（要截断的文件 inode）。

作用：释放文件占用的所有磁盘块（通过 bfree）。

重置 inode 的块指针和文件大小（ip->size = 0）。

更新磁盘上的 inode（iupdate）。

释放直接块

遍历 ip->addrs[0..NDIRECT-1]，释放每个已分配的直接块。

调用 bfree 将块标记为空闲，并清空 inode 中的指针。

释放一级间接块

读取一级间接块（bread），其内容是一个数组，存储实际数据块的地址。

遍历该数组，释放所有已分配的数据块。

释放一级间接块本身。

释放二级间接块

读取二级间接块（bread），其内容是一级间接块的地址数组。

遍历每个一级间接块：

读取一级间接块（bp2），释放其指向的所有数据块。

释放一级间接块本身。

释放二级间接块本身。

至此，我们完成了Large File部分。

**Symbolic links**

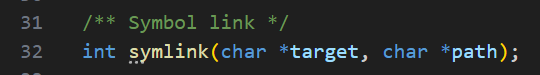
可以简单的将它理解为一个存有目标路径的文件，在Windows中可以将其理解为快捷方式。symbolic link文件有两种打开方式，其一是FOLLOW方式，即打开symbolic link文件内容中的路径；其次是NOFOLLOW，即将symbolic link文件视为普通文件，直接打开它。

首先和syscall一样创建sys\_symlink系统调用，相关代码如下：

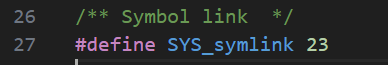
user/usys.pl



user/user.h



kernel/syscall.h



kernel/syscall.c



在kernel/stat.h中添加T\_SIMLINK来表示symbolic link文件

在kernel/fcntl.h中添加O\_NOFOLLOW标志，表示将symbolic link文件视为普通文件打开

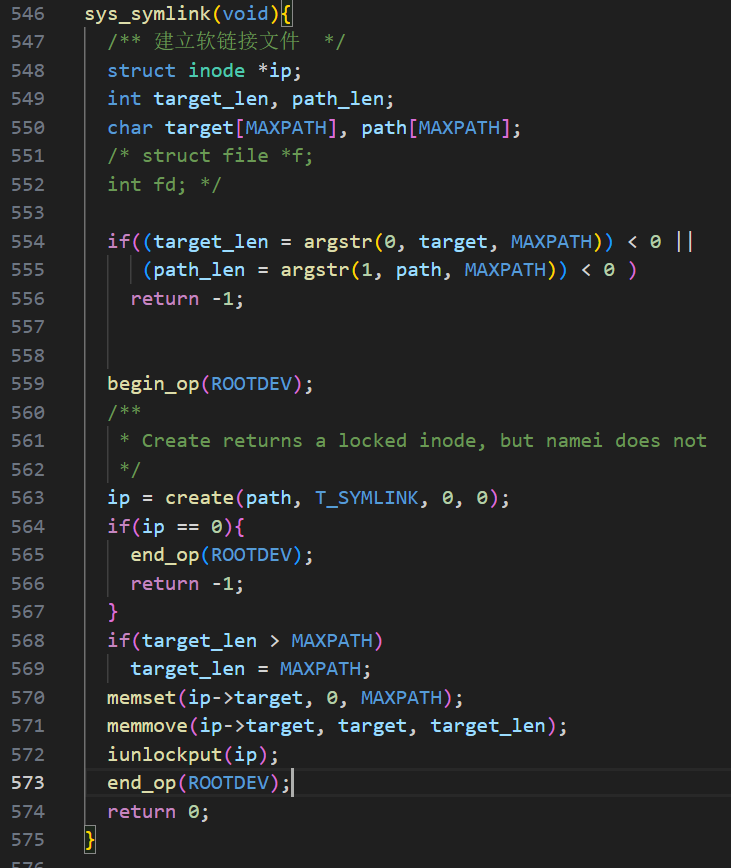
实现symlink(target,path)系统调用来创建一个symbolic link

修改open函数，使之能够处理T\_SIMLINK类型的文件；另外还需要判断标志O\_NOFOLLOW已决定是否FOLLOW。

如果在标志不为O\_NOFOLLOW的情况下，symbolic link链接的文件还是symbolic link文件，则应该递归寻找直到找到不是symbolic link的文件，将其打开。

其它系统调用不能FOLLOWsymbolic link 文件

在本次实验中不用处理指向目录的symbolic link文件



实现sys\_symlink函数。

输入：

target：符号链接指向的目标路径。

path：要创建的符号链接文件路径。

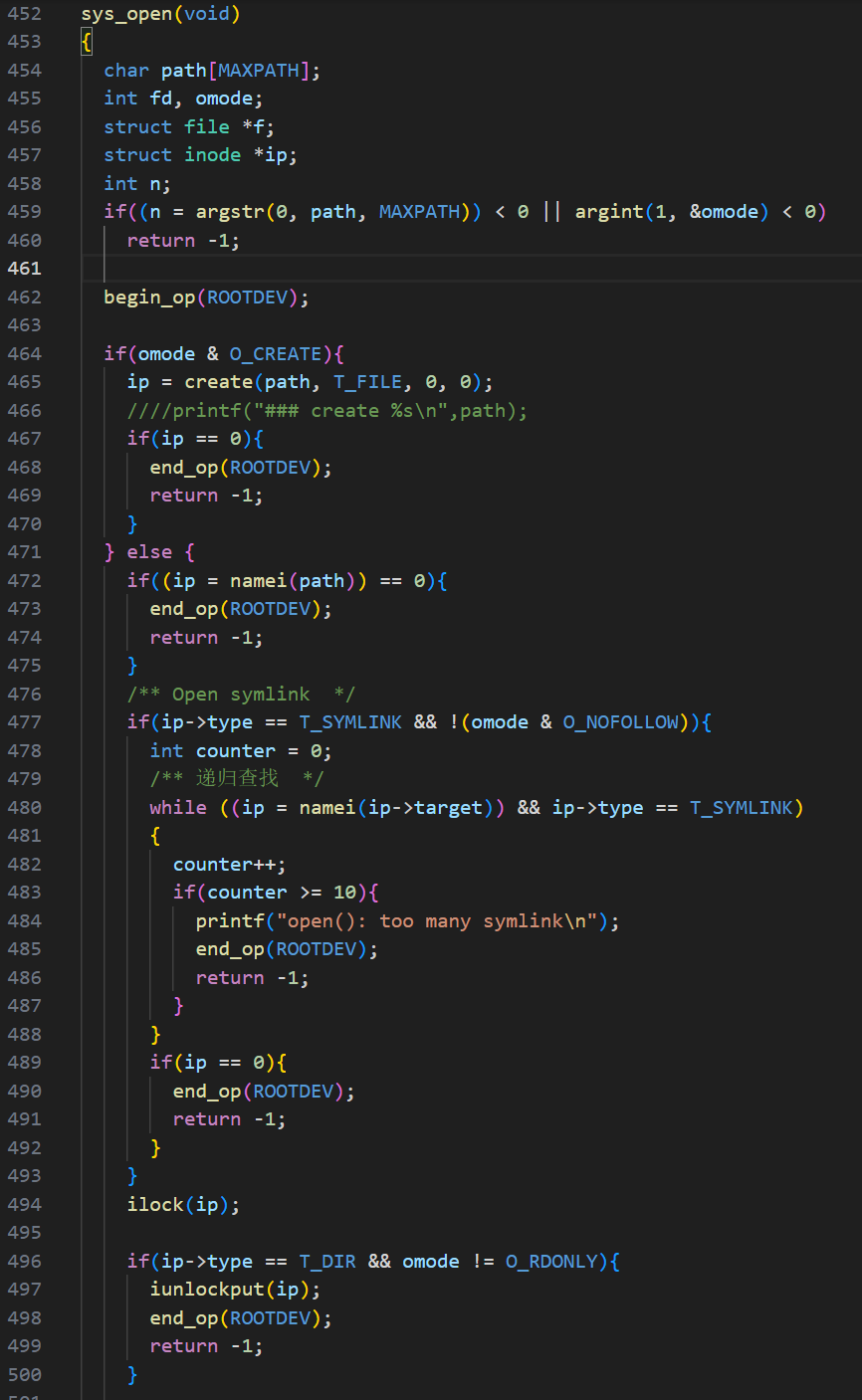
输出：

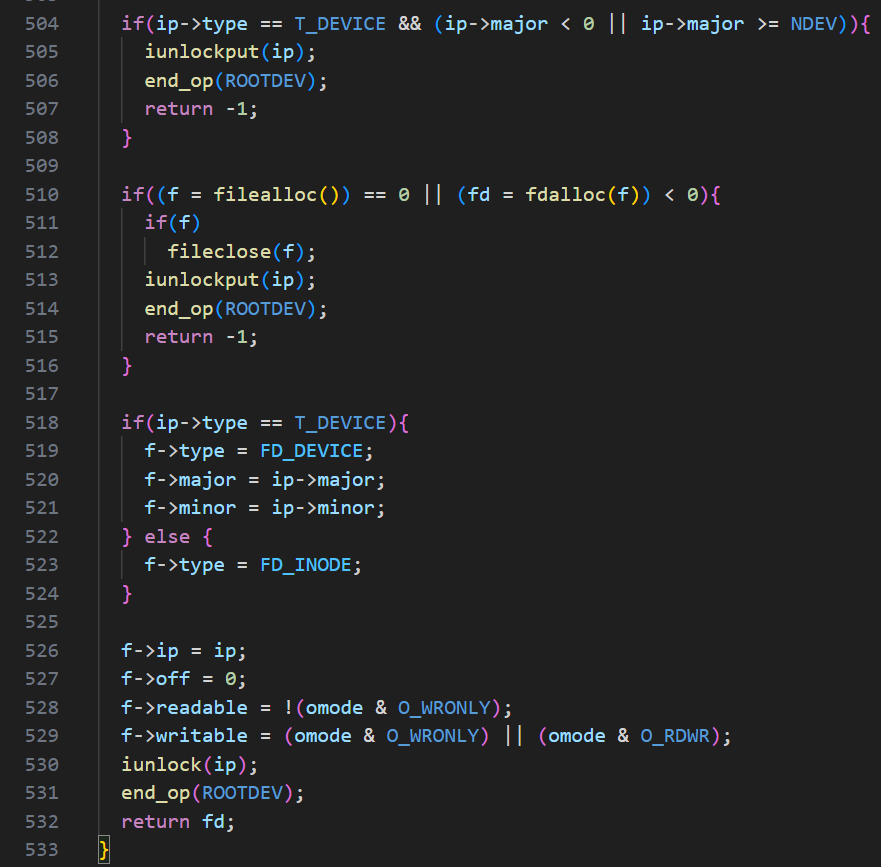
成功返回 0，失败返回 -1。

作用：

创建一个类型为 T\_SYMLINK 的文件，并将目标路径存储在该文件中。

首先从用户态获取两个参数：target：符号链接的目标路径。path：符号链接文件的路径。然后调用 create() 创建一个类型为 T\_SYMLINK 的文件。如果创建失败（如路径已存在），回滚事务并返回错误。检查目标路径长度，避免溢出。将目标路径（target）复制到 inode 的 target 字段中。最后释放 inode 锁并减少引用计数。提交文件系统事务。



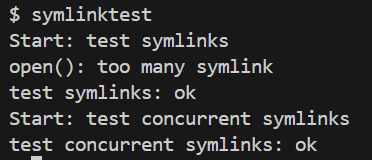


接下来，我们修改sys\_open函数，使之能够支持打开symbolic link文件。

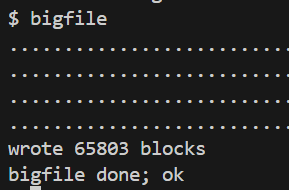
如果文件是符号链接（T\_SYMLINK）且未指定 O\_NOFOLLOW，递归解析目标路径。

限制最大递归深度（10 层），避免循环链接。

**实验现象：**



Symlinktest通过



Bigfile测试通过、

**实验中遇到的问题和解决办法：**问题：启动make qemu时报错

在检查后发现对inode定义的修改有遗漏，应当同时修改内存中和磁盘中的inode定义，即修改fs.h和file.h 两者不一致的话会出现问题

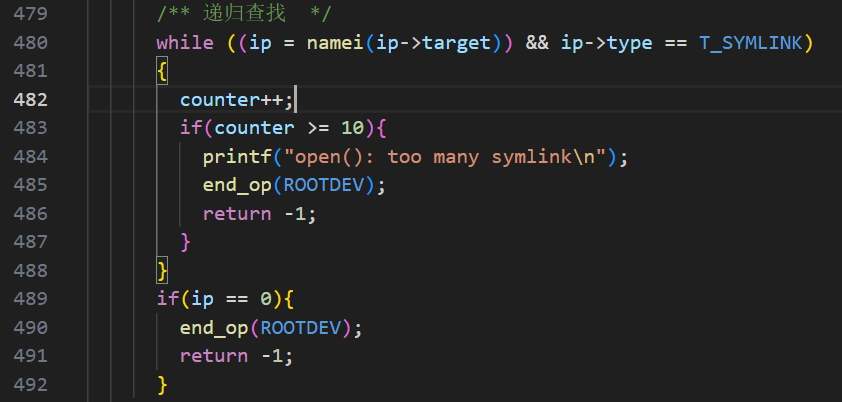
问题描述

在实现 sys\_open() 的符号链接支持时，测试 symlinktest 发现系统卡死。调试发现当符号链接形成循环（如 a -> b -> a）时，内核陷入无限递归。

解决过程

定位问题：通过 printf 打印解析路径，确认递归未终止。

添加深度限制：在符号链接解析循环中增加计数器，超过阈值（如10层）返回错误。



**实验心得：**

通过本次文件系统实验，我深刻理解了操作系统核心组件的设计哲学与实现挑战。在实现符号链接时，递归解析引发的死循环问题让我意识到内核开发必须严格处理所有边界条件。日志系统的实现过程尤其让我印象深刻，它完美诠释了如何通过事务机制在性能与可靠性间取得平衡。这次实验不仅让我掌握了inode管理、缓存优化等关键技术，更培养了我以"故障驱动"的思维设计系统——通过精心构造的测试用例（如磁盘空间耗尽、循环链接等）来验证系统的鲁棒性。这些经验为后续学习分布式文件系统打下了坚实基础，也让我更深刻地理解了"简单性"与"完备性"之间的权衡艺术。