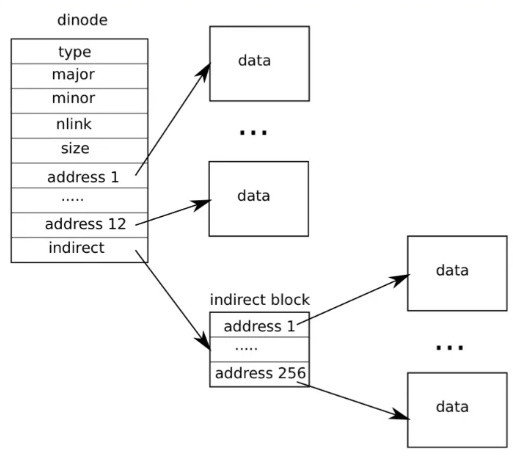
仓库链接： https://github.com/zhaoyihui233/mit-6.S081-lab-2021/tree/main/lab9%20File%20system

# 1 Large file

## 1.1实验内容

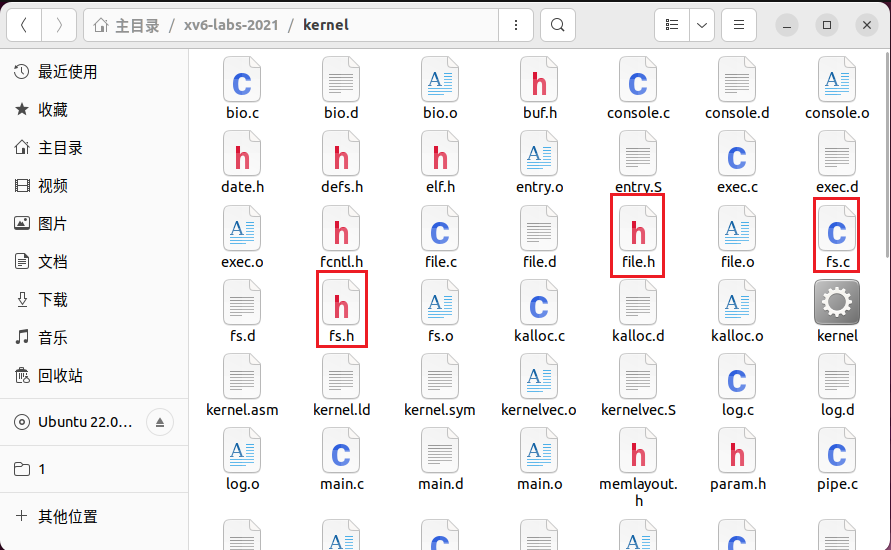
本实验中，我们需要为xv6实现存储表示大文件的功能。在xv6中，一个文件的数据在磁盘中的位置通过inode结构体组织，下图是inode组织记录磁盘块的形式：

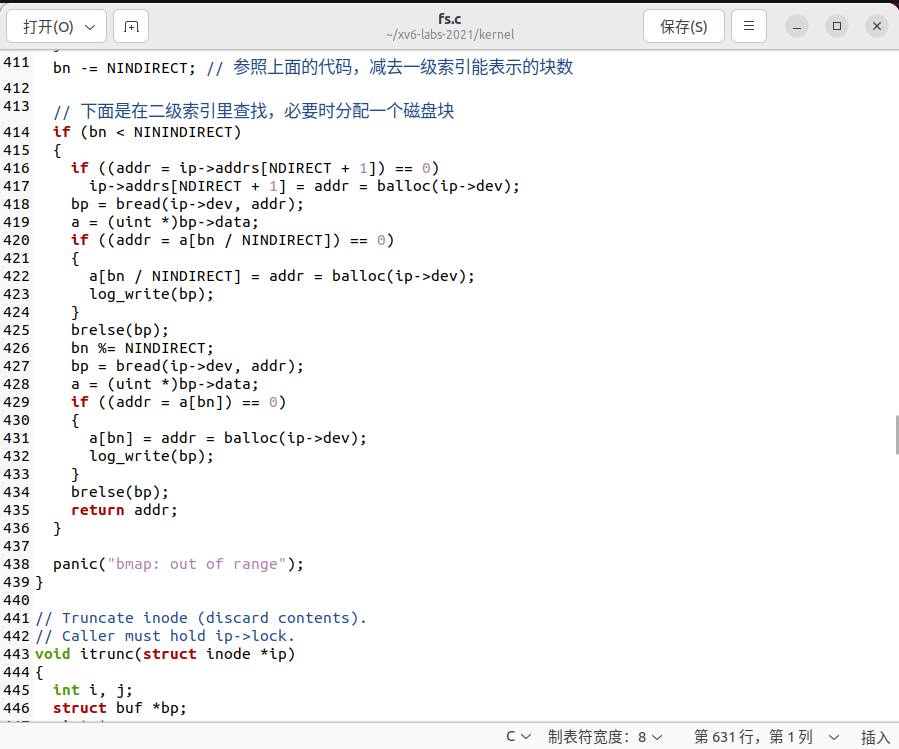


可知一个inode结构体最多可以记录12+256（因为一个地址指向一个块，一个块大小为1KB，一个地址长度为4B，因此一个块可以储存1KB/4B=256个地址）个磁盘块，每个磁盘块大小为1KB，那么该代码能表示的最大文件为268KB，一旦文件超过这个大小，xv6就无法表示。

我们的任务，就是改进这个索引方式，在该基础上再加一级索引，增大系统的表示范围。这里我们采用多级索引的方式，不改变inode的大小，使inode结构体中有11个地址直接表示磁盘块，1个地址是一级索引，指向256个地址，可以表示256个磁盘块，剩下一个地址是二级索引，指向的块包含256个地址，256个地址中的每一个又能指向一个包含256个地址的块。于是此时能够极限表示的文件大小为(11+256+256×256)KB。

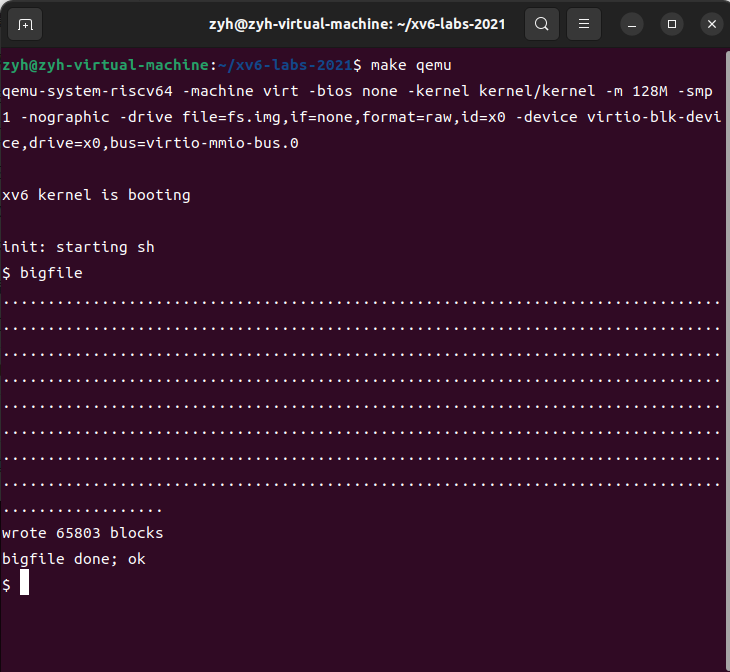
## 1.2代码位置与截图



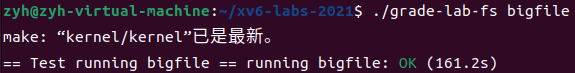


## 1.3实验结果

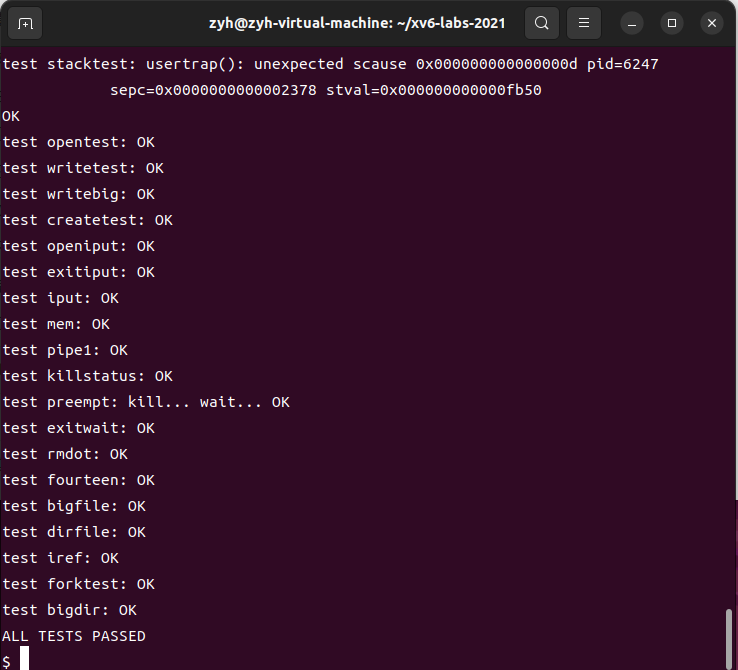
make qemu -> bigfile



./grade-lab-fs bigfile



usertests:



## 1.4代码实现

### 1.4.1实现思路

我们开始着手实现多级索引。首先是简单修改一个inode结构体的定义。在kernel/fs.h文件里修改、增加一些常量：

#define NDIRECT 11 // 直接索引的块数

#define NINDIRECT (BSIZE / sizeof(uint)) // 一级索引的块数

#define NDINDIRECT ((BSIZE / sizeof(uint)) \* (BSIZE / sizeof(uint))) // 二级索引的块数

#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT) // 系统能表示的文件的最大块数

#define NADDR\_PER\_BLOCK (BSIZE / sizeof(uint)) // 一个块中的地址数量

其中NDIRECT常量表示直接索引的数目，因此inode和dinode结构体里的uint addrs[NDIRECT + 2];属性也要简单地改改。

然后我们需要修改kernel/fs.c文件里的bmap函数和itrunc函数。bmap函数接收一个inode结构体参数ip和一个整数参数bn，返回ip索引的第bn个磁盘块的地址。如果没有这样的块，bmap会分配一个并返回地址。itrunc函数则是释放inode索引的空间。显然，这两个函数原来的代码都只能处理直接索引和一级索引的地址，我们需要添加处理二级索引的代码。

对于bmap函数，我们可以参考一级索引的代码，嵌套之后便是二级索引的代码。

int level2\_idx = bn / NADDR\_PER\_BLOCK;

int level1\_idx = bn % NADDR\_PER\_BLOCK;

level2\_idx是目的块在一级块下的编号，level1\_idx是目标块所在的一级块的编号。下面的代码就可以参考一级索引的代码了。需要注意的是bn需要减去一级索引能表示的块的数量，这样算出来的level1\_idx和level2\_idx才是正确的。lab的test针对这个有所检测，如果忘记减了，是会超出表示范围且报错的。

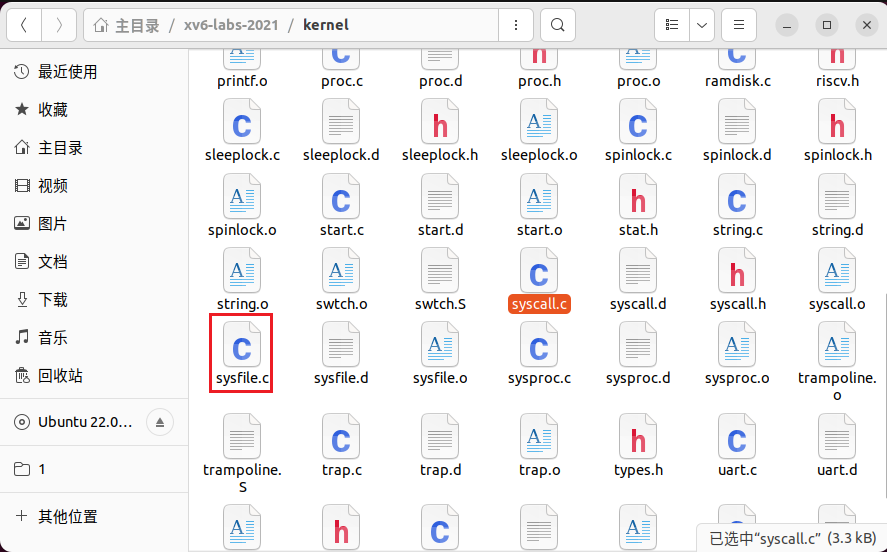
itrunc函数的修改与bmap类似，可以参考一级索引的代码。

# 2 Symbolic links

## 2.1实验内容

在本实验中，我们需要编译一个系统调用symlink，它的功能类似linux上的symlink软链接功能，比如执行symlink('a', 'b')后，open('b')这个指令会实际上索引到文件'a'所对应的文件磁盘块。

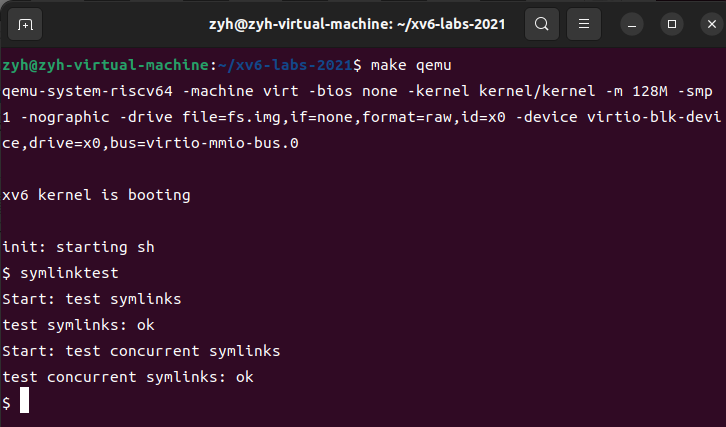
## 2.2代码位置与截图



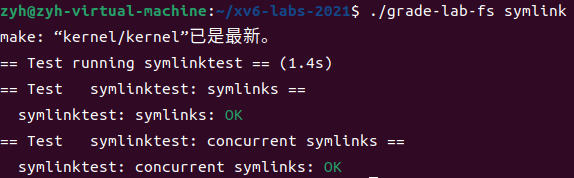


## 2.3实验结果

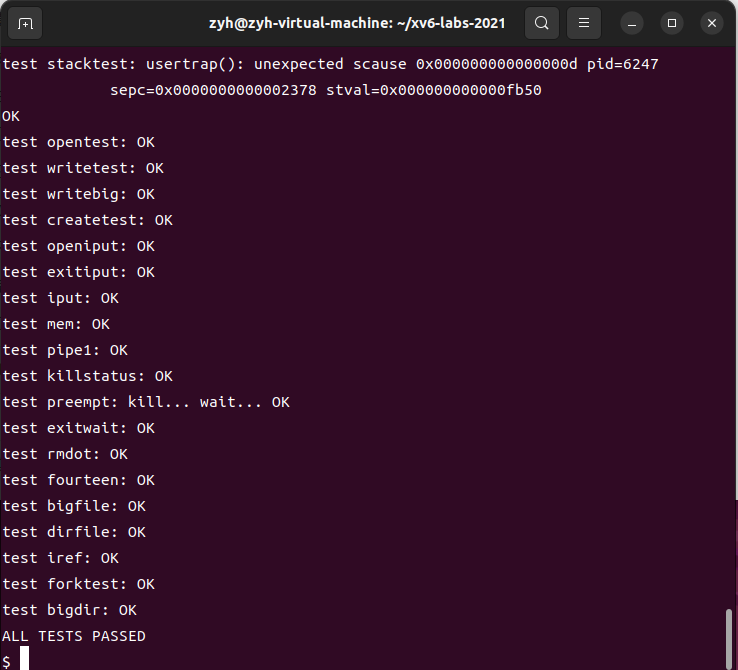
make qemu -> symlinktest



./grade-lab-fs symlink



usertests:



## 2.4代码实现

### 2.4.1实现思路

首先，是创建一个新的系统调用，我们要为symlink创建一个新的系统调用号（22），在user/usys.pl、user/user.h中添加一个条目，并在kernel/sysfile.c中实现一个空的sys\_symlink。接着，我们要向kernel/stat.h添加新的文件类型（T\_SYMLINK）以表示符号链接。然后，我们在kernel/fcntl.h中添加一个新标志（O\_NOFOLLOW），该标志可用于 open系统调用。传递给open的标志使用按位或运算符组合，因此新标志不应与任何现有标志重叠，这里定义为0x800与其他的区分开。别忘了把user/symlinktest.c添加到Makefile中，才可以编译它。

我们要实现的symlink(target, path)系统调用，是在path处创建一个新的指向target的符号链接。系统调用的成功不需要target文件已经存在。

我们要选择存储符号链接目标路径的位置，代码中把它储存在了inode的数据块中。symlink应返回一个表示成功（0）或失败（-1）的整数，类似于link和unlink。

另外，我们要修改open系统调用以处理路径指向符号链接的情况。当进程向open传递O\_NOFOLLOW标志时，open应打开符号链接（而不是跟随符号链接）。

如果链接文件也是符号链接，则必须递归地跟随它，直到到达非链接文件为止。如果链接形成循环，则必须返回错误代码。根据提示，我通过以下方式估算是否存在循环：通过在递归深度达到某个阈值（代码中是10）时返回错误代码。