Lab4 实验报告

姓名：李雨朔　学号：20232131090　班级：计科一班　日期：2025.11.21

一．　实验环境准备

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

二．　实验 4.1 对 xv6 文件系统添加“大文件”的支持

1. 在 kernel/fs.h 和 kernel/file.h 中修改宏定义和数据结构：

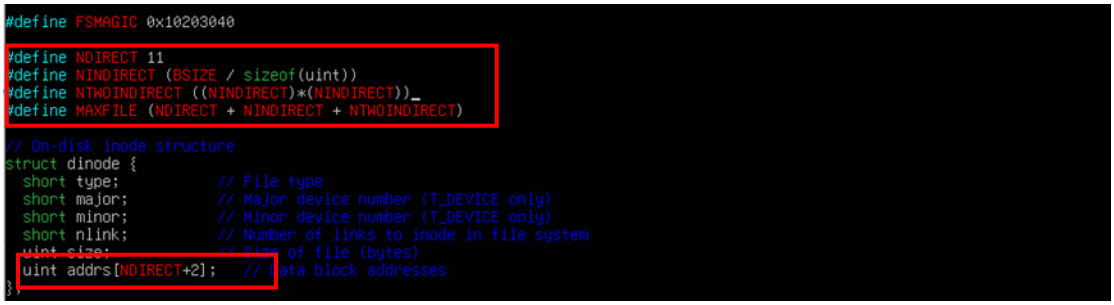
```
//kernel/fs.h
#define NDIRECT 11                  // 减少一个直接块用于二级间接块
#define NINDIRECT (BSIZE / sizeof(uint)) // 每个间接块包含 256 个地址
#define NTWOINDIRECT ((NINDIRECT)*(NINDIRECT)) // 二级间接块包含 65536 个地址
#define MAXFILE (NDIRECT + NINDIRECT + NTWOINDIRECT)
// 最大文件块数：11+256+65536=65803

struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];  // 11 个直接块 + 1 个一级间接块 + 1 个二级间接块
};

//kernel/file.h
struct inode {
  uint dev;
  uint inum;
  int ref;
  struct sleeplock lock;
  int valid;

  short type;
```

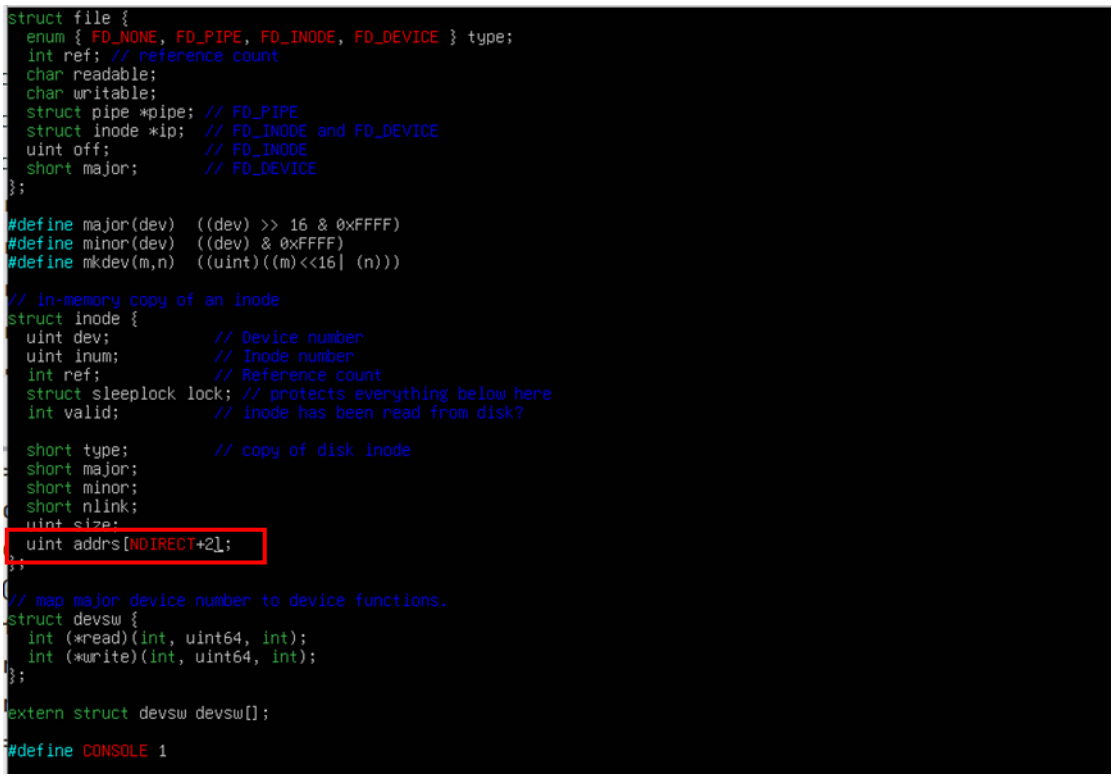
```
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];  // 与磁盘 inode 结构保持一致
};
```



```
#define FSMAGIC 0x10203040

#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NTWOINDIRECT ((NINDIRECT)*(NINDIRECT))
#define MAXFILE (NDIRECT + NINDIRECT + NTWOINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};
```



```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};

#define major(dev)  ((dev) >> 16 & 0xFFFF)
#define minor(dev)  ((dev) & 0xFFFF)
#define mkdev(m,n)  ((uint)((m)<<16| (n)))

// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];
};

// map major device number to device functions.
struct devsw {
  int (*read)(int, uint64, int);
  int (*write)(int, uint64, int);
};

extern struct devsw devsw[];

#define CONSOLE 1
```

2. 在 kernel/fs.c 中修改 bmap 函数，添加二级间接块支持:

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
```

```
if(bn < NDIRECT)
{
 //获得直接块的地址，若为 0 就分配
 if((addr = ip->addrs[bn]) == 0)
  ip->addrs[bn] = addr = balloc(ip->dev);
 return addr;
}
bn -= NDIRECT;
if(bn < NINDIRECT)
{
 //获得一级间接块的地址，若为 0 就分配
 if((addr = ip->addrs[NDIRECT]) == 0)
  ip->addrs[NDIRECT] = addr = balloc(ip->dev);
 bp = bread(ip->dev, addr); //根据地址读一级间接块
 a = (uint*)bp->data; //一级间接块的数据指针
 if((addr = a[bn]) == 0)
 {
   //根据指针从相应偏移量处获得数据块地址，若为 0 就分配
  a[bn] = addr = balloc(ip->dev);
  log_write(bp);
 }
 //获得地址后释放一级间接块并返回
 brelse(bp);
 return addr;
}
bn -= NINDIRECT;
struct buf *bp_2;
uint *a_2;
if(bn < NTWOINDIRECT)
{
 //获得二级间接块的地址，若为 0 就分配
 if((addr = ip->addrs[NDIRECT+1]) == 0)
  ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
 bp_2 = bread(ip->dev, addr); //根据地址读二级间接块
 a_2 = (uint*)bp_2->data; //二级间接块的数据指针
 if((addr = a_2[bn/NINDIRECT]) == 0)
 {
  //根据指针从相应偏移量处获得一级间接块地址，若为 0 就分配
  a_2[bn/NINDIRECT] = addr = balloc(ip->dev);
  log_write(bp_2);
 }
 //获得一级间接块地址后释放二级间接块
 brelse(bp_2);
 bp = bread(ip->dev, addr);//读一级间接块
```

```
    a = (uint*)bp->data;  //一级间接块的数据指针
    if((addr = a[bn%NINDIRECT]) == 0)
    {
      //根据指针从相应偏移量处获得数据块地址，若为 0 就分配
     a[bn%NINDIRECT] = addr = balloc(ip->dev);
     log_write(bp);
    }
    //获得数据块地址后释放一级间接块并返回
    brelse(bp);
    return addr;
  }
  panic("bmap: out of range");
}
```

```
  if((addr = ip->addrs[bn]) == 0){
    //addr = balloc(ip->dev);
    ip->addrs[bn] = addr = balloc(ip->dev);
    //if(addr == 0)
      //return 0;
    //ip->addrs[bn] = addr;
  }
  return addr;
  //}
}
bn -= NDIRECT;

if(bn < NINDIRECT){
  // Load indirect block, allocating if necessary.
  if((addr = ip->addrs[NDIRECT]) == 0){
    ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    //addr = balloc(ip->dev);
    //if(addr == 0)
      //return 0;
    //ip->addrs[NDIRECT] = addr;
  }
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[bn]) == 0){
    a[bn] = addr = balloc(ip->dev);
    //addr = balloc(ip->dev);
    //if(addr){
      //a[bn] = addr;
    log_write(bp);
```

```
uint *a_2;

if(bn < NTWOINDIRECT) {
  //uint level1_index = bn / NINDIRECT;
  //uint level2_index = bn % NINDIRECT;

  if((addr = ip->addrs[NDIRECT+1]) == 0){
    ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
  }
  bp_2 = bread(ip->dev, addr);
  a_2 = (uint*)bp_2->data;
  if((addr = a_2[bn/NINDIRECT]) == 0) {
    a_2[bn/NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp_2);
  }
  brelse(bp_2);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[bn%NINDIRECT]) == 0) {
    a[bn%NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  return addr;

}

//printf("bmap panic: bn=%d, NDIRECT=%d, NINDIRECT=%d, NDINDIRECT=%d\n", (int)(bn+NDIRECT+NINDIRECT), (int)NDIRECT, (int)

panic("bmap: out of range");
}
```

3. 在 kernel/fs.c 中修改 itrunc 函数，添加二级间接块的释放逻辑：

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp,*bp_2;
  uint *a,*a_2;
  for(i = 0; i < NDIRECT; i++)
  {
    if(ip->addrs[i])
    {
      //遍历所有直接块地址，若不为 0 就释放然后设为 0
     bfree(ip->dev, ip->addrs[i]);
     ip->addrs[i] = 0;
    }
  }
  if(ip->addrs[NDIRECT])
  {
    //一级间接块不为 0
   bp = bread(ip->dev, ip->addrs[NDIRECT]); //读一级间接块
   a = (uint*)bp->data; //一级间接块的数据指针
   for(j = 0; j < NINDIRECT; j++)
   {
      //释放一级间接块下的所有数据块
     if(a[j])
       bfree(ip->dev, a[j]);
   }
    //释放一级间接块本身
   brelse(bp);
   bfree(ip->dev, ip->addrs[NDIRECT]);
   ip->addrs[NDIRECT] = 0;
  }
  if(ip->addrs[NDIRECT+1])
  {
    //二级间接块不为 0
   bp_2 = bread(ip->dev, ip->addrs[NDIRECT+1]); //读二级间接块
   a_2 = (uint*)bp_2->data; //二级间接块的数据指针
   for(i = 0; i < NINDIRECT; i++)
   {
     //遍历二级间接块下的所有一级间接块
     if(a_2[i])
     {
      //该一级间接块不为空
       bp = bread(ip->dev, a_2[i]);
```

```
        a = (uint*)bp->data;
        for(j = 0; j < NINDIRECT; j++)
        {
          if(a[j])
            bfree(ip->dev, a[j]);
        }
        //释放一级间接块本身
        brelse(bp);
        bfree(ip->dev, a_2[i]);
        a_2[i] = 0;
       }
      }
       //释放二级间接块本身
      brelse(bp_2);
      bfree(ip->dev, ip->addrs[NDIRECT+1]);
      ip->addrs[NDIRECT+1] = 0;
    }
    ip->size = 0;
    iupdate(ip);
  }
```

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp, *bp2;
  uint *a, *a2;

  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]) {
    bp2 = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a2 = (uint*)bp2->data;
    for(i = 0; i<NINDIRECT; i++) {
      if(a2[i]) {
        bp = bread(ip->dev, a2[i]);
        a = (uint*)bp->data;
        for(j = 0; j<NINDIRECT; j++) {
          if(a[j]) {
            bfree(ip->dev, a[j]);
          }
        }
        brelse(bp);
        bfree(ip->dev, a2[i]);
        a2[i] = 0;
      }
    }
    brelse(bp2);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
```

```
  }

  ip->size = 0;
  iupdate(ip);
}
```

4. 编译和测试：

```
make clean
make qemu

$ bigfile
```

```
$ bigfile
..........
..........
..........
..........
..........
wrote 65803 blocks
reading bigfile
..........
..........
..........
..........
..........
bigfile done; ok
```

三． 实验 4.2 符号链接

1. 添加系统调用号和声明：

```
// kernel/syscall.h
// 添加系统调用号
#define SYS_symlink  22

//kernel/syscall.c
// 在 extern 声明部分添加
extern uint64 sys_symlink(void);
// 在系统调用函数指针数组中添加
[SYS_symlink]   sys_symlink,

//user/user.h
// 在系统调用声明部分添加
int symlink(const char *target, const char *path);

//user/usys.pl
# 在入口列表中添加
entry("symlink");
```

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_symlink 22
```

```
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_symlink(void);
```

```
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_symlink] sys_symlink,
};
```

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(const char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int symlink(const char *target, const char *path);
```

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("symlink");
```

2. 定义符号链接类型和标志位:

```
//kernel/stat.h
#define T_SYMLINK 4   // 符号链接文件类型

//kernel/fcntl.h
#define O_NOFOLLOW 0x800  // 不跟随符号链接
```

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
#define T_SYMLINK 4

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x800
```

3. 在 kernel/sysfile.c 文件中添加 sys_symlink 函数:

```
uint64
sys_symlink(void)
{
  char target[MAXPATH];
  char path[MAXPATH];
  struct inode *ip;
  // 从用户空间获取参数
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0){
    return -1;
  }
```

```
    // 开始文件系统操作
    begin_op();
    // 创建符号链接文件
    ip = create(path, T_SYMLINK, 0, 0);
    if(ip == 0)
    {
      end_op();
      return -1;
    }
    // 将目标路径写入符号链接文件的数据块中
    if(writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH)
    {
      // 写入失败，需要清理
      iunlockput(ip);
      end_op();
      return -1;
    }
    iunlockput(ip);
    end_op();
    return 0;
  }
```

```
uint64
sys_symlink(void)
{
  char target[MAXPATH];
  char path[MAXPATH];
  struct inode *ip;

  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
    return -1;
  }

  begin_op();

  ip = create(path, T_SYMLINK, 0, 0);
  if(ip == 0) {
    end_op();
    return -1;
  }

  if(writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip);
    end_op();
    return -1;
  }

  iunlockput(ip);
  end_op();
  return 0;
}
```

4. 在 kernel/sysfile.c 的 sys_open 函数中添加符号链接处理逻辑：

```
    // 符号链接处理：如果文件是符号链接且没有设置 O_NOFOLLOW 标志
    if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW))
    {
      // 这是个符号链接文件并且确定要 follow
      int depth = 10; // 最大递归深度
```

```
  char target[MAXPATH];
  for(int i = 0; i < depth; i++)
  {
    // 从符号链接文件的数据块中读取出路径 target
    if(readi(ip, 0, (uint64)target, 0, MAXPATH) != MAXPATH)
    {
      iunlockput(ip);
      end_op();
      return -1;
    }
    // 原本的符号链接文件已经不需要了，释放它的锁
    iunlockput(ip);
    // 得到了 target，解析路径得到目标文件的 inode
    if((ip = namei(target)) == 0)
    {
      end_op();
      return -1;
    }
    ilock(ip);
    // 如果目标文件不再是符号链接，停止循环
    if(ip->type != T_SYMLINK)
      break;
    // 检查是否超过最大递归深度
    if(i == depth - 1)
    {
      iunlockput(ip);
      end_op();
      return -1;
    }
  }
}
```

```
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;

  argint(1, &omode);
  if((n = argstr(0, path, MAXPATH)) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);

    if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
      int depth = 10;
      char target[MAXPATH];
      for(int i=0; i<depth; i++) {
        if(readi(ip, 0, (uint64)target, 0, MAXPATH) != MAXPATH) {
          iunlockput(ip);
          end_op();
          return -1;
        }
        iunlockput(ip);
        if((ip = namei(target)) == 0) {
          end_op();
          return -1;
        }
        ilock(ip);
        if(ip->type!=T_SYMLINK) break;
```

```
        if(i == depth-1) {
          iunlockput(ip);
          end_op();
          return -1;
        }
      }
    }

    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }
```

5. 修改 Makefile，在 UPROGS 部分添加：

$U/_symlinktest

```
UPROGS=\
        $U/_cat\
        $U/_echo\
        $U/_forktest\
        $U/_grep\
        $U/_init\
        $U/_kill\
        $U/_ln\
        $U/_ls\
        $U/_mkdir\
        $U/_rm\
        $U/_sh\
        $U/_stressfs\
        $U/_usertests\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_symlinktest\
```

6. 编译和测试：

make clean

make qemu

$ symlinktest

```
xv6 kernel is booting

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

四． 实验主观心得

本次实验中，在和 AI 智斗了很久后，终于走出 AI，然后发现 AI 世界外的代码清晰很多（这次的实验代码都来自于网上的大佬，自己的代码基础实在是太差了，基本很难独立写出代码）。AI 给我提供的很多解决方案在理论上可行，但实际上却无法出结果。然后去网上一查，再 结合 AI 解释，发现好几个问题其实用简单的思路就能解决（大佬写得代码还清晰易懂， AI 的代码就没这种效果）。

仍然感谢 AI 的帮助。