操作系统课程设计 – 选做作业（文件系统实现）

09017227 卓旭

本次实验的主要内容是实现一个简单的文件系统 naivefs。**受限于个人水平，该实验没有完全成功。**作为课程设计的选做项目，在此总结实现过程中的关键点，以及遇到的问题和解决方法。这里采用自顶向下的方式来讲述问题。

（一）文件系统基本概念

在进行该实验前，最好温习 Linux 文件系统相关的各个概念：

① 块：磁盘上的一定大小的区域，一个文件可以由多个块组成；

② inode：索引结点，每个文件都有一个 inode，包含文件的基本信息，以及组成文件的所有块的指针；

③ superblock：超级块，文件系统用来存自身基本元信息的一或多个块。

（二）自定义超级块、inode 结构

尽管 Linux 系统自带 super_block、inode 两种结构体，但其上的属性有限，不足以支持文件系统的定制化需求。因此，大多数文件系统都会自定义超级块和 inode 结构，然后把它挂到原生结构体的 private 域（sb->s_fs_info、inode->private）下。

根据需求，我们自定义的结构如下：

```
// 自定义超级块
// 考虑到 naivefs 基本不做异常处理，省略了很多没有用到的属性
struct naive_super_block {
  int magic;                // 魔数
  int inode_total;          // inode 的总量
  int block_total;          // 块的总量
  int inode_table_block_no; // inode 表块起始位置
  int data_block_no;        // 数据块起始位置
  // 补齐到一个块
  _Byte _padding[(NAIVE_BLOCK_SIZE - 5 * sizeof(int))];
};

// 自定义 inode
struct naive_inode {
  int mode;                         // mode
  int i_ino;                        // ino
  int block_count;                  // 该 inode 负责几个块
  int block[NAIVE_BLOCK_PER_FILE]; // 负责哪几个块，用第 0 块首部存 dir_record
  // 对于文件，记录文件大小；对于目录，记录目录下项目数
  union {
    int file_size;
    int dir_children_count;
  };
  // TODO: 下面这些属性原生 inode 上也有，先记着，没作用后续删掉
```

```
  int i_uid;
  int i_gid;
  int i_nlink;
  int i_atime;
  int i_ctime;
  int i_mtime;
  // FIXME: 补齐方便 naive_get_inode 计算，但是没必要
  _Byte _padding[(NAIVE_BLOCK_SIZE - (10 + NAIVE_BLOCK_PER_FILE) * sizeof(
int))];
};

// 目录下的项目的记录
struct naive_dir_record {
  int i_ino;                                // 所属目录的 inode 编号
  char filename[NAIVE_MAX_FILENAME_LEN]; // 文件名
};
```

可以看到，自定义的数据结构允许我们记录更多的信息，实现更灵活的文件系统。另外，为方便计算，这里暂时将每个自定义 inode 大小都补足到一个块的大小。

（三）格式化工具 mkfs.naive

naivefs 的分区布局如下图：

引 导 块	超 级 块	数据块位图	索引结点位图	索引结点表	数 据 块
	一个块	一个块	一个块	*n* 个块	*n* 个块

图 10-4　naive 分区的物理布局

为了可以正确挂载 naivefs，需要有格式化工具按该布局策略进行格式化。格式化策略如下：

- 引导块占一个块，留空不用
- 超级块占一个块，其中填充一个 naïve_super_block 结构
- 块位图（bmap）占一个块，对应于数据块之前的 bit 全部置 1，禁止用户存放
- inode 位图（imap）占一个块，前两 bit 置 1，用于支持根目录的.和..
- inode 表和数据块所占部分大小根据磁盘大小动态计算
- inode 表中应有根 inode 的记录，即根目录的支持
- 数据块首部要有.和..的 dir_record

格式化时，一些参数如下：

```
#define NAIVE_BLOCK_SIZE 512        // 块大小 512B
#define NAIVE_MAGIC 990717          // 魔数
#define NAIVE_BLOCK_PER_FILE 8      // 每个文件最多占多少块
#define NAIVE_MAX_FILENAME_LEN 128 // 文件名最大长度
#define NAIVE_BOOT_BLOCK 0          // 引导块块号
```

```
#define NAIVE_SUPER_BLOCK_BLOCK 1  // 超级块块号
#define NAIVE_BMAP_BLOCK 2         // 块位图块号
#define NAIVE_IMAP_BLOCK 3         // inode 位图块号
#define NAIVE_ROOT_INODE_NO 0      // 根 inode 编号
#define NAIVE_SUPER_BLOCK_SIZE sizeof(struct naive_super_block)
#define NAIVE_INODE_SIZE sizeof(struct naive_inode)
#define NAIVE_DIR_RECORD_SIZE sizeof(struct naive_dir_record)
```

Linux 允许我们像访问文件一样访问设备，因此只需调用 open 拿到 fd 后，利用 write 进行写入，必要时使用 lseek 调整指针即可。

（四）文件系统模块声明

内核模块分为两个类型：直接编译到内核的，和可动态插拔的。naivefs 采用第二种方式。为此，需要书写插拔函数，并利用 module_init 和 module_exit 注册：

```
// 将文件系统作为可插拔模块注册到系统
static int __init init_naivefs(void) {
  return register_filesystem(&naive_fs_type);
}
// 拔出文件系统模块
static void __exit exit_naivefs(void) {
  unregister_filesystem(&naive_fs_type);
}
// 声明插拔函数
module_init(init_naivefs)
module_exit(exit_naivefs)
```

（五）file_system_type 声明

注意到上面用到 naive_fs_type 变量。这是一个类型为 file_system_type 的结构体，表明了文件系统类型的基本信息。

```
// 文件系统类型定义
static struct file_system_type naive_fs_type = {
    .owner = THIS_MODULE,
    .name = "naivefs",
    .fs_flags = FS_REQUIRES_DEV,
    .get_sb = naive_get_sb,
    .kill_sb = kill_block_super, // 销毁超级块用自带的方法即可
};
```

（六）naïve_get_sb，获取超级块

该函数声明了如何获取一个超级块，主要借助系统自带的 get_sb_bdev。它需要我们定义 naive_fill_super 来说明如何填充超级块。

```
// 该函数声明了如何获取一个超级块
// 获取超级块借助自带的 get_sb_bdev 即可，注意 naive_fill_super 说明了如何填充超级块
```

```
// 这里和指导书不一样，跟进代码后发现 2.6.21.7 下 get_sb_nodev 原型返回值为 int，且增加
mnt 参数
static int naive_get_sb(struct file_system_type *fs_type, int flags,
                        const char *dev_name, void *data,
                        struct vfsmount *mnt) {
  return get_sb_bdev(fs_type, flags, dev_name, data, naive_fill_super, mnt
);
}
```

（七）naïve_fill_super，填充超级块信息

我们在磁盘上存放的超级块，是自定义超级块 naïve_super_block 类型，而不是原生超级块。为了实现从磁盘上读取自定义超级块，有如下两个步骤：

```
// 从磁盘上读块，主要借助 buffer_head 指针和 sb_bread 来完成
struct buffer_head *bh = sb_bread(sb, NAIVE_SUPER_BLOCK_BLOCK);
// 我们在超级块的 b_data 中放的是自定义超级块信息
struct naive_super_block *nsb = (struct naive_super_block *)bh->b_data;
```

naïve_fill_super 要求我们向它的参数 struct super_block *sb 填充系统侧需要的基本信息，这在我们拿到 naïve_super_block 后不是件难事：

```
  sb->s_magic = nsb->magic; // 魔数
  sb->s_op = &naive_sops;   // sops
  sb->s_maxbytes =
      NAIVE_BLOCK_SIZE * NAIVE_BLOCK_PER_FILE; // 声明每个文件的最大大小
  sb->s_fs_info = nsb; // 将自定义 super_block 结构放到私有域

  // 我们还需要拼装一个根目录的 inode，也叫根 inode，这个 inode 要关联到超级块
  // 利用 new_inode 方法可以取到一个可用的空 inode
  struct inode *root_inode = new_inode(sb);
  // 填一些信息
  my_inode_init_owner(root_inode, NULL, 0755 | S_IFDIR);
  struct naive_inode *root_ninode = naive_get_inode(sb, NAIVE_ROOT_INODE_N
O);
  root_inode->i_ino = NAIVE_ROOT_INODE_NO;
  root_inode->i_sb = sb;
  root_inode->i_mode = root_ninode->mode;
  // FIXME: 把目录下文件数作为 i_size 是否合适？
  root_inode->i_size = root_ninode->dir_children_count;
  // add，modify，create
  root_inode->i_atime = root_inode->i_mtime = root_inode->i_ctime =
      CURRENT_TIME;
  root_inode->i_nlink++;
  // i_op 是 inode 操作集，f_op 是文件对象操作集，由于 root_inode 对应一个目录，所以这
里给 fop 赋 dops
```

```
  root_inode->i_op = &naive_iops;
  root_inode->i_fop = &naive_dops;
  root_inode->i_private = root_ninode;

  // 最后关联根 inode 和超级块即可
  sb->s_root = d_alloc_root(root_inode);
```

（六）kill_sb

我们使用系统提供的 kill_block_super 来实现卸载文件系统时销毁超级块的动作。该方法需要我们在 super_block 的操作集（sops）中定义 put_super 的处理方式。主要是对超级块进行释放即可：

```
// 该函数说明了如何卸载文件系统，主要是做一些清理善后工作
static void naive_put_super(struct super_block *sb) {
  // 释放超级块即可
  struct naive_super_block *nfs = NAIVE_SB(sb);
  if (nfs == NULL)
    return;
  kfree(nfs);
  return;
}
```

（七）sops、iops、fops、dops、aops

在前面的代码中，我们已经看到许多种操作集（ops），它们说明了该对象可以具有的方法，是在 C 这种面向过程语言中引入面向对象因素的常用方法。在此，给出所有的 ops 定义：

```
// sops 实现了 inode 的读写和 naive 的卸载
static struct super_operations naive_sops = { // super_block ops
    .read_inode = naive_read_inode,
    .write_inode = naive_write_inode,
    .put_super = naive_put_super,
    .statfs = simple_statfs,
};
// iops 实现了常用的三个
static struct inode_operations naive_iops = { // inode ops
    .lookup = naive_lookup,
    .create = naive_create,
    .mkdir = naive_mkdir,
};
// fops 基本不需要特殊处理，全部靠系统自带完成即可
static struct file_operations naive_fops = { // file ops
    .llseek = generic_file_llseek,
    .read = do_sync_read,
    .aio_read = generic_file_aio_read,
```

```
    .write = do_sync_write,
    .aio_write = generic_file_aio_write,
    .mmap = generic_file_mmap,
    .sendfile = generic_file_sendfile,
};
static struct file_operations naive_dops = { // directory ops
    .read = generic_read_dir,
    .readdir = naive_readdir,
};
static struct address_space_operations naive_aops = { // address_space ops
    .readpage = simple_readpage,
    .sync_page = block_sync_page,
    .commit_write = generic_commit_write,
};
```

其中，不以 naïve 开头的项目均是调用系统自带功能完成，不必过于纠结。

（八）工具函数

```
// 根据 inode 编号在指定文件系统实例（一个超级块对应一个文件系统实例）的 inode 表中取出对应的自定义 inode
static struct naive_inode *naive_get_inode(struct super_block *sb, int ino
) {
  // 先取出自定义超级块信息
  struct naive_super_block *nsb = NAIVE_SB(sb);

  // naive 只有一个超级块、只有一个 BlockGroup
  // 而且为了简单，我们的 inode 每个占一块
  // 故下面这个简单的算式就可以算得 ino 对应 inode 在 inode 表中所在的块号
  int block_no_of_ino = nsb->inode_table_block_no + ino; // FIXME
  // 找一个 bh，读出整个块
  struct buffer_head *bh = sb_bread(sb, block_no_of_ino);
  struct naive_inode *ninode = (struct naive_inode *)bh->b_data;

  return ninode;
}

// 这个函数用来初始化 inode 很好用，但在 2.6.21.7 内核下还未提供，我们做个 polyfill
// inode_init_owner 的第二个参数是归属目录，根 inode 没有归属目录，给 NULL
// inode_init_owner 的第三个参数是 inode 的访问控制属性
void my_inode_init_owner(struct inode *inode, const struct inode *dir,
                         umode_t mode) {
  if (dir == NULL) {
    inode->i_uid = 0; // 这样做不一定准确
    inode->i_gid = 0; // 这样做不一定准确
```

```
  } else {
    inode->i_uid = dir->i_uid;
    inode->i_gid = dir->i_gid;
  }
  inode->i_mode = mode;
}

// 把变化的数据块写回盘，和下面写回 ninode 的差不多
static void write_back_block(/* 略 */) { /* 略 */ }

// 把自定义 inode 写回盘
static void write_back_ninode(struct super_block *sb,
                              struct naive_inode *ninode) {
  struct naive_super_block *nsb = NAIVE_SB(sb);
  // 应该放到 inode 表的哪个块
  int block_no = nsb->inode_table_block_no + ninode->i_ino;
  // 现在开始上盘
  struct buffer_head *bh = sb_bread(sb, block_no);
  // 块首指针
  struct naive_inode *block_head = (struct naive_inode *)bh->b_data;
  memcpy(block_head, ninode, NAIVE_INODE_SIZE);
  // sb_bread 只是读出来放到内存，前面更改的也是内存，并没有写回盘
  // 为了上盘，要建立映射，这是从 ext2 学的
  map_bh(bh, sb, block_no);
  brelse(bh);
}

// 获取一个可用的空 data_block 编号
static int naive_new_block_no(/* 略 */) { /* 略 */ }

// 获取一个可用的空 inode 编号，与自带的 new_inode 不同的是，该方法采用 bitmap 确定空闲
inode 编号
static int naive_new_inode_no(struct super_block *sb) {
  // 先取超级块
  struct naive_super_block *nsb = NAIVE_SB(sb);

  // 先把 imap 读进内存
  int imap_size = nsb->block_total / 8; // 1bit 每块，故除以 8 为字节数
  _Byte *imap = kmalloc(imap_size, GFP_KERNEL);
  // 根据布局图，inode_table_block 之前为 imap_block，且只有一块
  // naivefs 不考虑磁盘空间非常大，使得一块不足以支持 imap 存放的情况
  struct buffer_head *bh = sb_bread(sb, NAIVE_IMAP_BLOCK);
```

```
  memcpy(imap, (_Byte *)(bh->b_data), imap_size);

  // 现在，只需要在 imap 找到首个为 0（未使用）的位即可
  int res = NAIVE_ROOT_INODE_NO + 1;
  _Byte *imap_ptr = imap;
  while (*(imap_ptr++) == 0x00ff)
    res += 8;
  _Byte detector = 1;
  while ((*imap_ptr) & detector == 1) {
    detector <<= 1;
    res += 1;
  }

  brelse(bh);
  kfree(imap);
  return res;
}

// 把某块的 bmap 对应 bit 置值
static void set_bmap_bit(struct super_block *sb, int block_no, bool to) {
  // 取自定义超级块
  struct naive_super_block *nsb = NAIVE_SB(sb);
  // 读出整个 bmap
  // FIXME: 在不对 naivefs 数据块的数量进行限制的前提下，并不一定保证 bmap 只占一块
  int bmap_size = nsb->block_total / 8;
  _Byte *bmap = kmalloc(bmap_size, GFP_KERNEL);
  struct buffer_head *bh = sb_bread(sb, NAIVE_BMAP_BLOCK);
  memcpy(bmap, (_Byte *)(bh->b_data), bmap_size);

  int line = block_no / 8;   // 在第几个 Byte
  int offset = block_no % 8; // 在第 line 个 Byte 的第几 bit
  int bit = to == true ? 1 : 0;
  *(bmap + line) |= (bit << offset);

  // 改完拷回去
  // TODO: 有没有必要一次写整块？
  memcpy(bh->b_data, bmap, NAIVE_BLOCK_SIZE);
  map_bh(bh, sb, NAIVE_BMAP_BLOCK);

  brelse(bh);
}
```

```
// 把某块的 imap 对应 bit 置值
static void set_imap_bit(/* 略 */) { /* 略 */ }

// 用于取 super_block 上的私有域
static struct naive_super_block *NAIVE_SB(struct super_block *sb) {
  return sb->s_fs_info;
}
```

可以总结出如下几点：

- 当获得原生 super_block 时，相当于也拿到了 naïve_super_block，只需要调用 NAÏVE_SB 转换即可
- 从盘上读信息的方法是借助 buffer_head *bh 以及 sb_bread 函数，传入超级块和要读的块号，即可一次读出一块
- 如要进行从内存到盘上的回写，要使用 map_bh 函数，传入超级块、回写数据、回写块号，建立一个映射

（九）文件操作支持

本节讨论如何让 naivefs 支持文件（和目录）的创建。主要书写一个 naïve_mknod 函数将帮助我们创建目录和文件，考虑的是更普遍的情况——创建结点（mknod）。minix 文件系统中也是这样实现的。

mknod 的函数签名如下：

static int naive_mknod(struct inode *dir, struct dentry *dentry, int mode)

dir：要创建的文件所在的目录；dentry：新建的文件的目录项

mknod 的主要步骤如下：

- 为新文件分配 inode 号
- 拼装原生 inode 和 naïve_inode，填写相关属性
- 如果要创建的是目录，则添加.和..两条 dir_record，并且由于有 dir_record，必须立即分配数据块，这导致最后要回写 naïve_inode、block，同时要更新 bmap
- 如果要创建的是文件，建立新的文件（空文件）并不占据数据块，只需要回写 naïve_inode 即可
- 不管是什么类型，都要更新它们的所在目录（父目录）的 inode 信息，即增加 dir_record，并且回写 naïve_inode
- 使用 mark_inode_dirty 来标记需要更新的原生 inode
- 更新 imap，因为分配了新的 inode 号
- 使用 d_instantiate 关联 dentry 和 inode

当有了可靠的 mknod 函数后，mkdir 和 create 的实现就简单了：

```
// 创建文件
static int naive_create(struct inode *dir, struct dentry *dentry, int
mode,
                        struct nameidata *nd) {
```

```
  return naive_mknod(dir, dentry, mode);
}
// 创建目录
static int naive_mkdir(struct inode *dir, struct dentry *dentry, int m
ode,
                       struct nameidata *nd) {
  // 注意！根据资料，虽然 naive_mkdir 会被系统在创建文件夹时调用，
  // 但是调用时系统给的 mode 并不激活 IFDIR 位！这大概是一个 bug。
  // 不管怎样，手动与上 IFDIR。
  return naive_mknod(dir, dentry, mode | S_IFDIR);
}
```

（十）目录操作支持

目前仅编写了 readdir 的支持。readdir 说明了如何遍历一个目录，获取其中的文件信息。实现它，文件系统就支持 ls 命令。naïve_readdir 的原型如下：

static int naive_readdir(struct file *filp, void *dirent, filldir_t filldir)

注意，这里 filp 指向的是目录，而不是一个文件（这也是为什么它归在 dops 中），但在 linux 里万物皆文件，所以这里用的仍是 file*，这略有歧义，值得说明。

filldir_t 是一个函数指针类型，也就是说 filldir 是一个回调函数。通过在函数中调用 filldir，来告知系统目录下有哪些文件。

naïve_readdir 的主要逻辑如下：

- 通过 filp 取超级块，进而取到目录的 inode，转换为 naïve_inode
- 读出该 naïve_inode 存储 dir_record 的块
- 借助 dir_children_count，逐个解析 dir_record，用 filldir 告知系统

（十一）Makefile

为了正确编译内核模块和 mkfs.naïve，我们需要编写一个 Makefile。两者的主要区别是：

- mkfs.naïve 是一个面向用户的应用程序，工作在用户态，不需要引入与内核相关的头文件，因此使用 gcc 单独编译即可
- naivefs 工作在内核，需要与内核联合编译，方可生成.ko 文件，供 insmod 使用

综上，Makefile 如下：

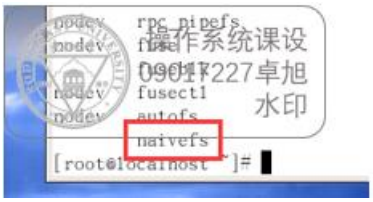
```
obj-m := naivefs.o
KERNEL_DIR := /lib/modules/$(shell uname -r)/build

default: # naivefs itself
  $(MAKE) -C ${KERNEL_DIR} M=$(PWD) modules

mkfs: # mkfs tool
  gcc mkfs.naive.c -o mkfs.naive
```

```
clean: # clean both
  rm -rf *.ko *.o *.mod.o *.mod.c *.symvers .*.cmd .tmp_versions
  rm -rf mkfs.naive
```

（十二）测试流程

为测试文件系统能否正确使用，按照实验指导书，总结测试流程如下：

- **编译：**在项目目录下
  make mkfs
  make
- **使用环回设备模拟磁盘：**在合适目录下
  dd if =/dev/zero of=tmpfile bs=1k count=200
  losetup /dev/loop0 tmpfile
- **格式化：**在项目目录下
  ./mkfs.naive /dev/loop0
  注意，stat 对环回设备取 size 时，会为 0 值，这将导致格式化计算不正确。若有此情况，可以对 tmpfile 进行格式化：
  ./mkfs.naive <some_path>/tmpfile
- **注册模块：**在项目目录下
  insmod naivefs.ko
  cat /proc/filesystems
  可看到 naivefs 已经被添加：



- **挂载：**选用/mnt/naïve 作挂载点
  mkdir /mnt/naïve
  mount -t naivefs /dev/loop0 /mnt/naïve
- **测试：**若正确无误，则 cd /mnt/naïve 后，此处则工作 naivefs，可使用各项文件操作命令进行进一步测试

（十三）总结

本次实验并没有取得成功。出过许多状况，如段错误、内核 dump 出寄存器、mount 提示非目录、ls 提示.不是目录、文件不可保存等等问题。每当调节好一个问题后，另一个问题便随之而来。最后，也没能实现一个可读写的 on-disk 文件系统。

究其原因，首先是自身对 Linux 文件系统相关知识确实不太熟悉，相关 API、相关结构体下挂的属性是什么含义，都不敢说是清楚明白的。这也导致很多时候调试是在摸黑、排列组合进行尝试，而没有一个顺畅的思路指引。

另外，文件系统是一个精密的整体，整个体系不能有一丝错误。任何一个错误的算式，哪怕是多加了 1，或是少加了 1，都会带来完全不同的结果。更何况会经常涉及二进制位运算、内存地址计算等等。

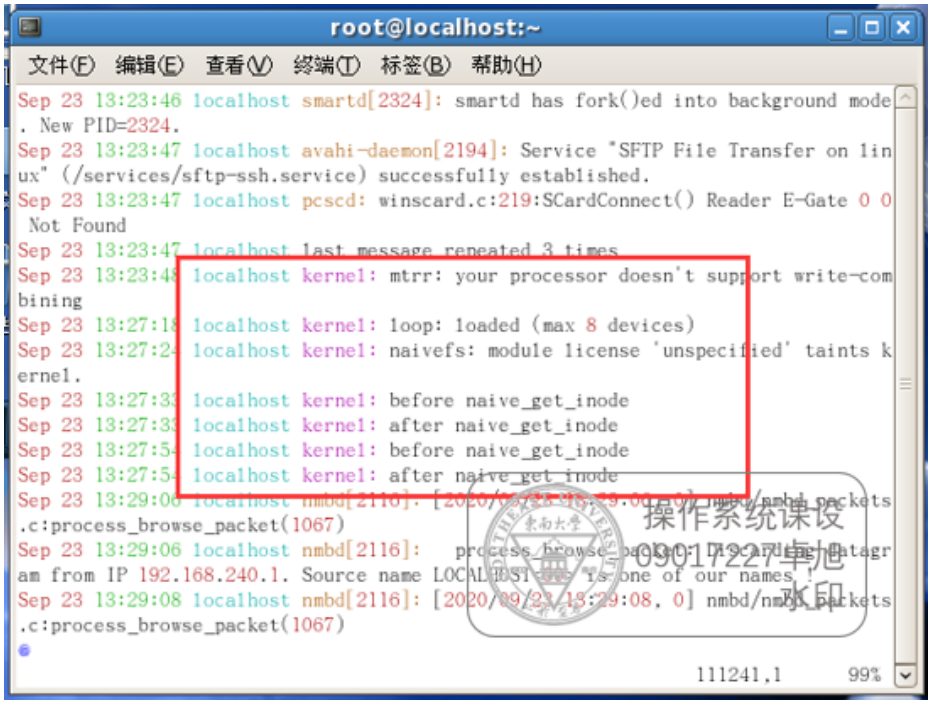
Linux 内核的相关资料也偏少，即便是在谷歌上，有时也很难找到完整的函数参数含义。这也加大了实验难度。

不管如何，最后还是附上 naivefs 当前的源代码：

https://github.com/z0gSh1u/naivefs 。

（十四）调试技巧

对内核级的调试相当困难。我最常用是 printk 函数。它与 printf 基本类似，但 printf 不能在内核级输出到 stdout，而 printk 可在内核级进行输出。调用 printk 后，其输出在 /var/log/messages 文件中，以 localhost kernel: 打头。这可以帮助我们打出所需的变量，辅助调试。如下图所示：



（十五）参考资料

本实验的报告和代码都参考了大量的资料，在它们的帮助下逐渐写成。我承认代码中有许多它们的影子，因此在此进行一个归纳：

项目	说明
linux-2.6.21.7\fs\ext2	Linux 内核，ext2 文件系统实现
linux-2.6.21.7\fs\minix	Linux 内核，minix 文件系统实现
https://elixir.bootlin.com/linux/v2.6.21.7/C/ident/	查询某个 Linux 标识符（函数、宏，或结构定义）在内核源码中的位置
实验指导书	—
http://www2.comp.ufscar.br/~helio/fs/rkfs.html	rkfs，圣卡洛斯联邦大学文件系统作业，只有一个文件的文件系统实现

https://github.com/psankar/simplefs	simplefs，一名印度程序员实现的完善的 on-disk 文件系统，但很复杂
https://github.com/yangoliver/lktm/tree/master/fs	samplefs，IBM Linux 技术中心的文件系统教学课作业，但是 on-ram 的
https://github.com/karelzak/util-linux/tree/master/disk-utils	mkfs 的官方实现
https://github.com/ZhangShurong/simple-filesystem	HUST_fs，华中科技大学操作系统实验，但有较多 bug