# Virtual File System (VFS) Implementation in Linux

Tushar B. Kute,
http://tusharkute.com

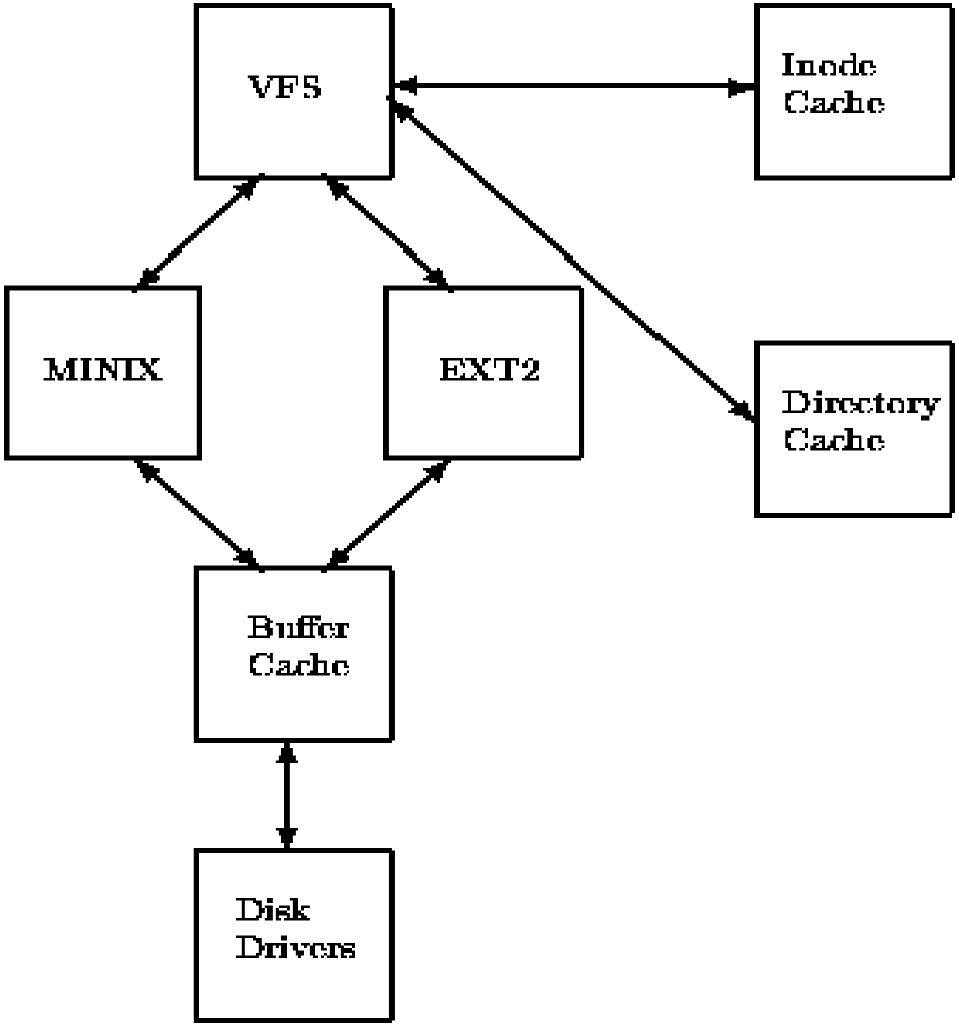
# Virtual File System

- The Linux kernel implements the concept of Virtual File System (VFS, originally Virtual Filesystem Switch), so that it is (to a large degree) possible to separate actual "low-level" filesystem code from the rest of the kernel.
- This API was designed with things closely related to the ext2 filesystem in mind. For very different filesystems, like NFS, there are all kinds of problems.

# Virtual File System- Main Objects

- The kernel keeps track of files using in-core **inodes** ("index nodes"), usually derived by the low-level filesystem from on-disk inodes.
- A file may have several names, and there is a layer of **dentries** ("directory entries") that represent pathnames, speeding up the lookup operation.
- Several processes may have the same file open for reading or writing, and **file** structures contain the required information such as the current file position.
- Access to a filesystem starts by mounting it. This operation takes a filesystem type (like ext2, vfat, iso9660, nfs) and a device and produces the in-core **superblock** that contains the information required for operations on the filesystem; a third ingredient, the mount point, specifies what pathname refers to the root of the filesystem.

# Virtual File System



# The /proc filesystem

- The /proc filesystem contains a illusionary filesystem.
- It does not exist on a disk. Instead, the kernel creates it in memory.
- It is used to provide information about the system (originally about processes, hence the name).
- The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures.  It is commonly mounted at /proc.
- Most of it is read-only, but some files allow kernel variables to be changed.
- The /proc filesystem is described in more detail in the proc manual page.

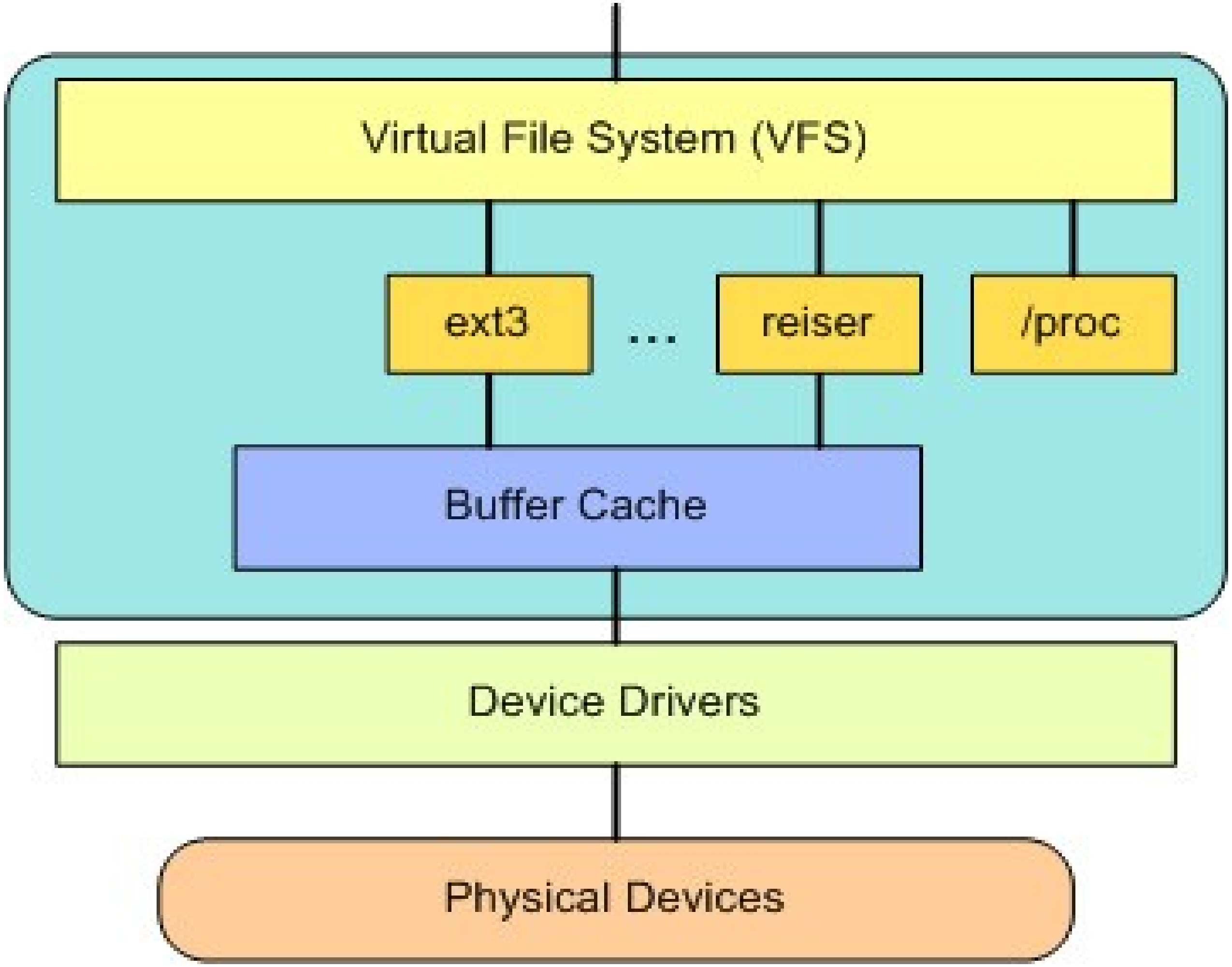
# Some /proc

- **/proc/1**
  - A directory with information about process number 1. Each process has a directory below /proc with the name being its process identification number.
- **/proc/cpuinfo**
  - Information about the processor, such as its type, make, model, and performance.
- **/proc/devices**
  - List of device drivers configured into the currently running kernel.

# Some /proc

- **/proc/filesystems**
  - Filesystems configured into the kernel.
- **/proc/ioports**
  - Which I/O ports are in use at the moment.
- **/proc/meminfo**
  - Information about memory usage, both physical and swap.
- **/proc/version**
  - The kernel version.

# VFS in Linux


Virtual File System (VFS)
ext3
...
reiser
/proc
Buffer Cache
Device Drivers
Physical Devices

# Create filesystem as a module

- Write a hello_proc.c program.
- Create a Makefile.
- The program and Makefile should be kept in a single folder.
- Change directory to this folder and execute following:
  - **`make`**
  - **`insmod hello_proc.ko`**
  - **`dmesg`** (see the kernel buffer contents, reads the kernel log file /var/log/syslog)
  - **`lsmod`**
  - **`rmmod hello_proc.ko`**

# hello_proc.c

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
static int hello_proc_show(struct seq_file *m, void *v)  {
  seq_printf(m, "Hello proc!\n");
  return 0;
}
static int hello_proc_open(struct inode *inode, struct  file *file) {
  return single_open(file, hello_proc_show, NULL);
}
static const struct file_operations hello_proc_fops = {
  .owner = THIS_MODULE,
  .open = hello_proc_open,
  .read = seq_read,
  .llseek = seq_lseek,
  .release = single_release,
};
```

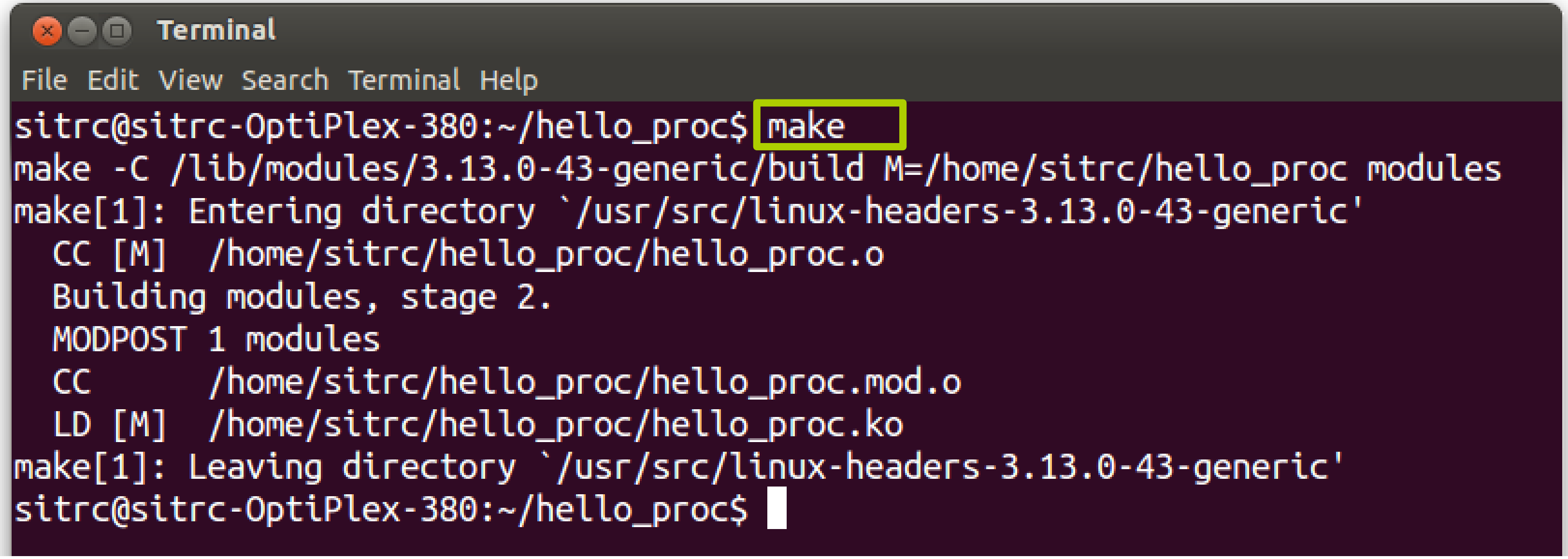
# hello_proc.c

```
static int __init hello_proc_init(void)
{
  proc_create("hello_proc", 0, NULL,
         &hello_proc_fops);
  return 0;
}
static void __exit hello_proc_exit(void)
{
  remove_proc_entry("hello_proc", NULL);
}
MODULE_LICENSE("GPL");
module_init(hello_proc_init);
module_exit(hello_proc_exit);
```

# Makefile

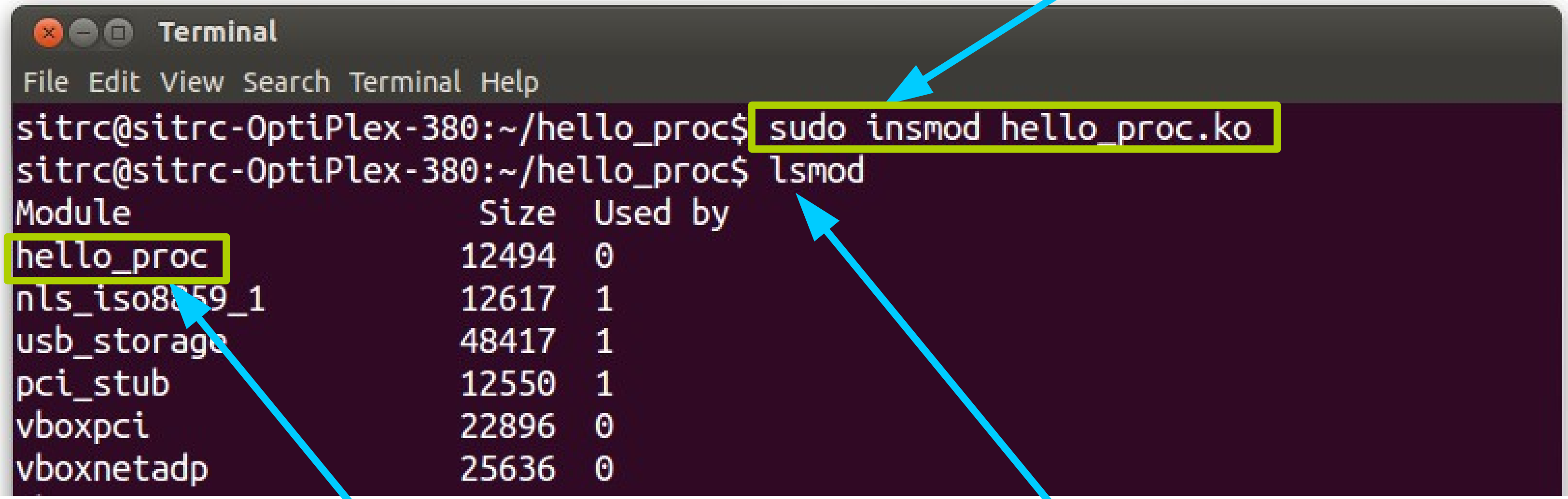
```
obj-m += hello_proc.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

# Make



```
sitrc@sitrc-OptiPlex-380:~/hello_proc$ make
make -C /lib/modules/3.13.0-43-generic/build M=/home/sitrc/hello_proc modules
make[1]: Entering directory `/usr/src/linux-headers-3.13.0-43-generic'
  CC [M]  /home/sitrc/hello_proc/hello_proc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/sitrc/hello_proc/hello_proc.mod.o
  LD [M]  /home/sitrc/hello_proc/hello_proc.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-43-generic'
sitrc@sitrc-OptiPlex-380:~/hello_proc$ 
```

# Insert and list



```
sitrc@sitrc-OptiPlex-380:~/hello_proc$ sudo insmod hello_proc.ko
sitrc@sitrc-OptiPlex-380:~/hello_proc$ lsmod
Module                  Size  Used by
hello_proc             12494  0
nls_iso8859_1          12617  1
usb_storage            48417  1
pci_stub               12550  1
vboxpci                22896  0
vboxnetadp             25636  0
```

Insert

List

hello_proc module

# See the entry

- `ls -l /proc`

- `cat /proc/hello_proc`

# Functions used

- **proc_create**
  - It creates a virtual file in the /proc directory.
- **remove_proc_entry**
  - It removes a virtual file from the /proc directory.
- **hello_proc_show()**
  - It shows the output.
- **seq_printf**
  - It uses sequential operations on the file.
- **hello_proc_open()**
  - This is the open callback, called when the proc file is opened.
- **single_open()**
    - All the data is output at once.

# The file_operations structure

- The file_operations structure holds pointers to functions defined by the driver that perform various operations on the device.
- Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

# Syntax of file_operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,  loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,  loff_t *);
  };
```

# Structure used in program

```
struct file_operations hello_proc_fops = {
  .owner = THIS_MODULE,
  .open = hello_proc_open,
  .read = seq_read,
  .release = single_release,
};
```

# Limitations to /proc file system

- Our module cannot output more than one page of data to the pseudo-file at once.
- A page is a pre-defined amount of memory, typically 4096 bytes (4K defined by processor), and is available in the PAGE_SIZE macro.
- This limitation is bypassed by using sequence files.

# Thank you

**Web Resources**
http://tusharkute.com

**Blogs**
http://digitallocha.blogspot.in
http://kyamputar.blogspot.in

**tushar@tusharkute.com**