KERNEL LINUX - part 3

File systems

- Located in the *fs/* directory of the Linux sources, along with file related core.
- Can be built-in or loaded as modules.
- Provide a wide range of functionality:
    - *Journals*
    - *Compression*
    - *Snapshots*
    - *Redundancy*
- Two general purposes of file system:
    - *File storage, e.g., ext4, zfs or btrfs.*
      Pseudo file systems, e.g., procfs, sysfs or
    - *devtmpfs, …*

The VFS

- A common interface to all the file systems.
- A hierarchy of directories, starting with /.
- Each Linux distribution chooses how to organize it.

    -

      *Linux foundations hierarchy: <**http://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.txt**>*
      Systemd’s modernized version is described
    - *in file-hierarchy(7).*

- Userspace interacts with the VFS using a set of syscalls: *open(2)*, *creat(2)*, *unlink(2)*, *mount(2)*, *chroot(2)*, *symlink(2)*, etc.

Changing /

- *chroot(2)* can be used to change the root of the filesystem hierarchy of the

current process. This is the syscall used by *chroot(1)* and *switch_root(8)*.

- Linux change root from initial ramdisk and true storage disk

```
int chroot(const char *path);
```

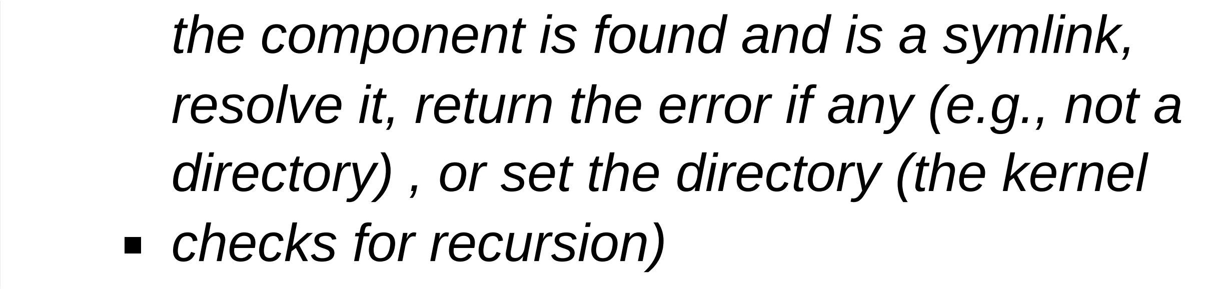
```
$ switch_root newroot init
```

Namespaces

- *clone(2)* with the *CLONE_NEWNS* flag: the child process get a modifiable copy of the parent’s mount points. This is called changing of mount namespace.
- *clone(2)* with the *CLONE_FS* flag: the child process get a strict copy of the parent’s mount points. Calls to *chroot(2)* or *chdir(2)* from the parent or the child also affect other processes.
- The list of filesystems currently mounted for the current namespace can be seen in */proc/[pid]/mounts*.
- More namespaces exist: *PID*, *USER*, *NET*, *IPC* and *UTS*.
- For more information, see *namespaces(7)*

Path resolution

- A kernel operation.
- Is the path absolute (starting with a /) or relative? This defines the starting lookup directory.
- For each nonfinal component of the pathname, look it up the current directory, if:
  - *the process does not have search permission, return EACCES*
  - *the component is not found, return ENOENT*
  - *the component is found, but not a directory, return ENOTDIR*
  - *the component is found and is a directory, set the current lookup directory to that directory and go to the next component*



- *the component is found and is a symlink, resolve it, return the error if any (e.g., not a directory) , or set the directory (the kernel checks for recursion)*

- Depending on the syscall, the final entry can be a directory or something else.

Mounts

- The *mount(1)* utility can be used to display the list of mount points in the current process’ namespace, and create new mount entries.
- The underlying syscall is *mount(2)*.
- A mount is composed of :

    - *Filesystem type*
    - *Source (may be unused)*
    - *Mount point*
    - *Mount flags*
    - *Mount data/options (optional)*

- Some options
- *MS_BIND*: the *bind* mount that makes a file or directory visible at another point within a filesystem.
- *MS_MOVE*: moves a mount point.

Anatomy of the VFS

Definition

- Mostly oriented toward traditional Unix filesystems: other filesystems must map their internal implementation with the Linux VFS structures.
- The VFS is composed of four main objects:
    - *superblock, information about a mounted filesystem*
    - *dentry, information about a directory entry*
    - *inode, information about a specific file on a filesystem*
    - *file, information about an opened file*
- Historically very similar to the ext2 filesystem.
- FAT / VFAT / NTFS filesystems do not have inodes

Caches

Multiple caches help avoiding immediate access to the underlying hardware, or other internal data structures.

- dentry cache (*dcache*): a view of the currently accessed paths.
- inode caches: if an inode can be uniquely identified by an *int*, make a cache out of it.
- block device caches (page cache): keep pages of data in RAM instead of fetching them at each request.

VFS doc in kernel

Adding a file system in Linux

Why creating a new file system?

- Adding entries to */proc* is discouraged.
- A single misc device may not be enough for what you want to do.
- Adding “new syscalls” using *ioctl(2)* is also discouraged.

Registering a new (pseudo) filesystem

```
/* my_fs.c */
static struct file_system_type fs_type = {
        .owner    = THIS_MODULE,
        .name     = "fs",
        .mount    = fs_mount,
        .kill_sb  = fs_kill_sb,
        /* Can be mounted by userns root */
        .fs_flags = FS_USERNS_MOUNT,
};

static int __init fs_init(void) {
        return register_filesystem(&fs_type);
}
```

fs_mount()

Setups the super block, returns the root *dentry*.

```
/* my_fs.c */
static struct dentry *fs_mount(
                     struct file_system_type *fs_type,
                     int flags,
                     const char *dev_name,
                     void *data)
{
        struct super_block *sb;

        /* find or create a superblock */
        sb = sget(fs_type, NULL, set_anon_super, flags,
                  NULL);
        fs_super_fill(sb);
        return dget(sb->s_root);
}
```

- No error handling!
- set_anon_super: no underlying block device, this is a pseudo device

fs_mount() (refactored)

The kernel sources have many shortcuts already written…

```
static struct dentry *fs_mount(
                     struct file_system_type *fs_type,
                     int flags, const char *dev_name,
                     void *data)
{
        /* Better */
        return mount_nodev(fs_type, flags, data,
                           fs_super_fill);
}
```

fs_super_fill()

Setup the filesystem metadata, create root inode.

```
#define MYFS_SUPER_MAGIC 0x12345678

int fs_super_fill(struct super_block *sb,
                  void *data, int silent)
{
        struct inode *inode;

        sb->s_blocksize = PAGE_CACHE_SIZE;
        sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
        sb->s_magic = MYFS_SUPER_MAGIC;
        sb->s_op = &super_sops;

        inode = fs_inode_get(sb, NULL, NULL);
        sb->s_root = d_make_root(inode);
        if (!sb->s_root)
                return -ENOMEM;
        return 0;
}
```

struct super_operations

Define in *include/linux/fs.h*
File system wide configuration. Default operations from *fs/libfs.c* can be used.

```
static const struct super_operations super_ops = {
        .statfs = simple_statfs,
        /* sync_fs */
        /* {alloc,destroy,dirty,drop,evict}_inode */
        /* show_{options,devname,path,stats} */
        /* ... */
};
```

fs_inode_get() (1/2)

Creates a new inode for *alloc_inode*.

```
struct inode *fs_inode_get(struct super_block *sb,
               const struct inode *dir,
               umode_t mode)
{
        /* For cache usage: iget_locked(sb, inode_id) */
        struct inode *inode = new_inode(sb);

        if (!inode)
                return ERR_PTR(-ENOMEM);

        inode->i_ino = get_next_ino();
        inode_init_owner(inode, dir, S_IFDIR);
        /* No address space mapping operations. */
        inode->i_mapping->a_ops = &empty_aops;
        inode->i_atime = inode->i_ctime = \
            inode->i_mtime = CURRENT_TIME;
```

fs_inode_get() (2/2)

```
switch (mode) {
default:
        init_special_inode(inode, mode,  0);
        break;
case S_IFDIR:
        inode->i_op = &simple_dir_inode_operations;
        inode->i_fop = &simple_dir_operations;
        /* directory inodes starts with i_nlink == 2
         * (for "." entry) */
        set_nlink(inode, 2);
        break;
case S_IFREG: case S_IFLNK:
        /* here for regular file or symlink ... */
        break;
}
```

```
    return inode;
}
```

Going further

- This file system can be mounted, but does not have any useful feature.
- We must fill all the necessary *{file,inode,dir,symlink}_operations* callbacks in order to make it do something useful.

Where to see what to do?

  - *fs/ramfs/inode.c* is one of the simplest file system in Linux.
  - *fs/overlayfs/* is also pretty small.

fs/libfs.c **contains a wide range of general purpose, default behavior**

-
  - operations.

FUSE: Filesystem in Userspace

- A filesystem in which data and metadata are provided by an ordinary userspace process. The filesystem can be accessed normally through the kernel interface.
- It consists of a kernel module (*fuse.ko*), a userspace library (*libfuse.so*) and mount utilities (*fusermount*, *mount.fuse*).
- Binding are available for other languages, eg. Python.
- Projects using FUSE:
  - *gdrivefs (Google Drive)*
  - *rar2fs (rar archives)*
  - *encfs (encrypted files)*
  - *ftpfs (ftp directly in the VFS)*
  - *sshfs (remove filesystem using ssh)*
- <**http://fuse.sourceforge.net/**>

CUSE: Character device in Userspace

- Userspace can create char devices!
- Based on the fuse kernel code (*fs/fuse/cuse.c*).
- Simplify driver development.
- But slow and limited.