Advanced Operating Systems
MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Virtual file system internals:

1. VFS basic concepts
2. VFS design approach and architecture
3. Device drivers
4. The Linux case study

File system: representations

- In RAM
  - Partial/full representation of the current structure and content of the File System
- On device
  - (non-updated) representation of the structure and of the content of the File System
- Data access and manipulation
  - FS independent part: interfacing-layer towards other subsystems within the kernel
  - FS dependent part: data access/manipulation modules targeted at a specific file system type

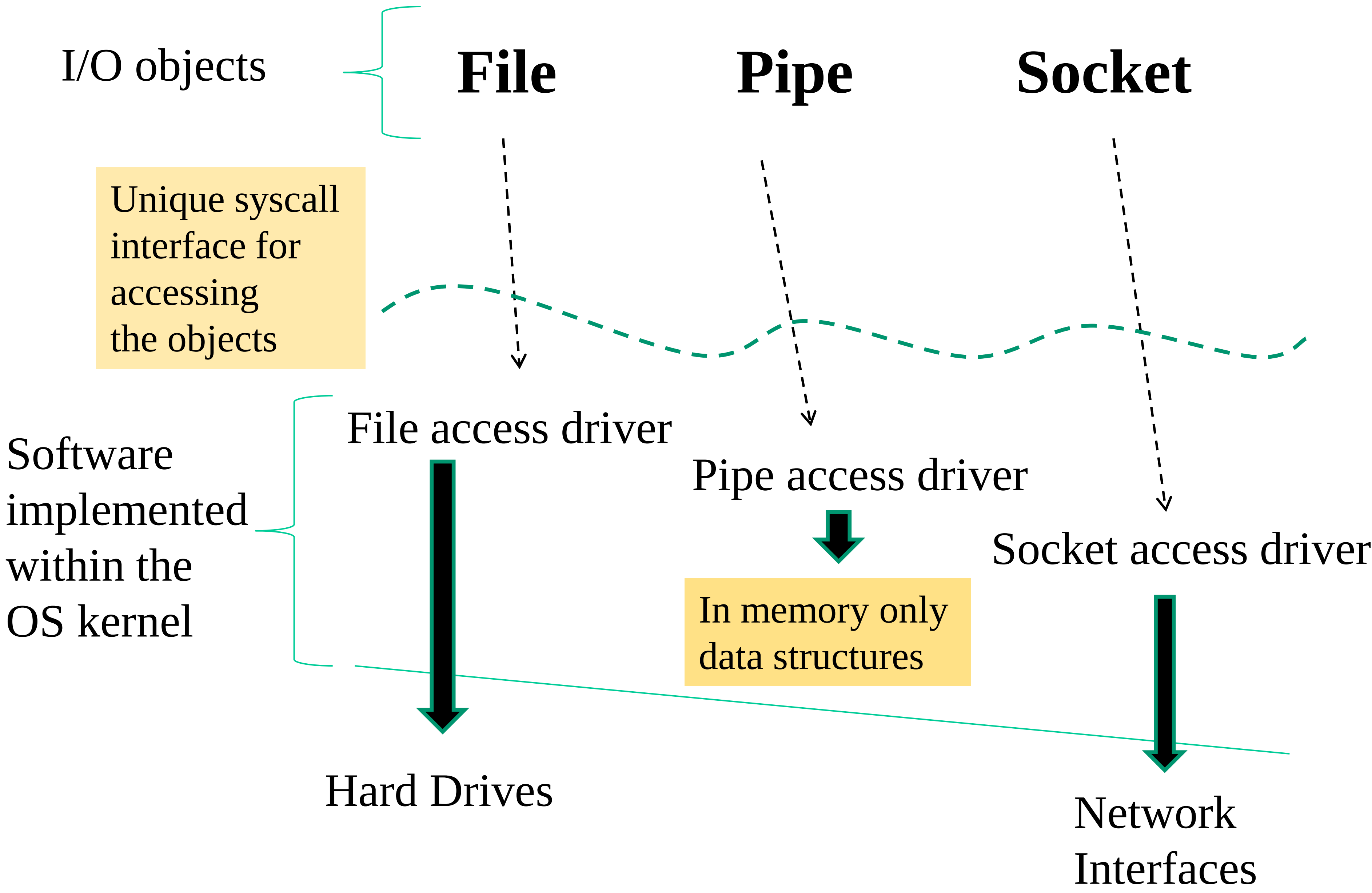
Connections

- Any FS object (dir/file) is represented in RAM via specific data structures
- The object keeps a reference to the module instances for its own operations
- The reference is accessed in a File System independent manner by any overlying kernel layer
- This is achieved thanks to multiple different instances of a same function-pointers’ (drivers’) table

VFS hints

- **Devices can be seen as files**
- What we drive, in terms of state update, is the structure use to represent the device in memory
- Then we can also reflect such state somewhere out of memory (on a hardware component)
- Classical devices we already know of
  - ✓ Pipes and FIFO
  - ✓ sockets

An overall scheme



Lets' focus on the true files example

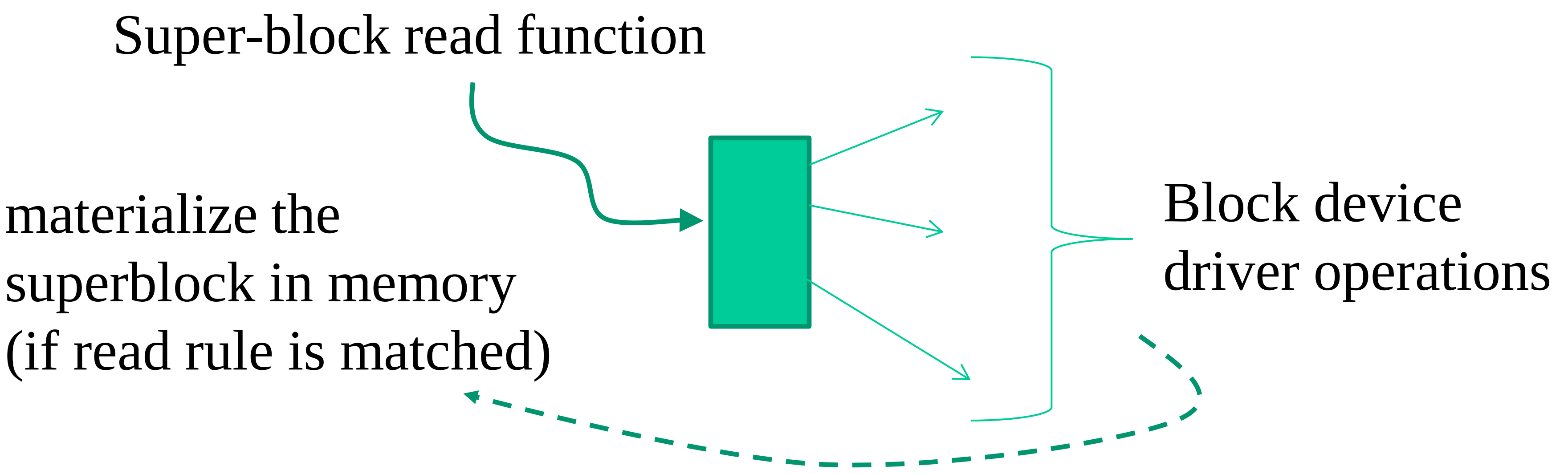
- These are backed by data on a hard drive
- What **software modules do we need** for managing files on that hard drive in a well shaped OS-kernel
  1. A function to read the device superblock for determining what files exist and where their data are
  2. A function to read device blocks for bringing them into a buffer cache
  3. A function to flush updated blocks back to the device
  4. A set of functions to actually work on the in-memory cached data and to trigger the activation of the above functions

Block vs char devices

- The first three points in the previous slide are linked to the notion of block device and **block device drivers**
- The last point (number 4) is linked to the notion of char device and **char device driver**
- These drivers are essentially <u>tables of function pointers</u>, pointing to the actual implementation of the operations that can be executed on the target object
- **The core point is therefore how to allow a VFS supported system call to determine what is the actual driver to run when a given system call is called**

File system types in Linux

- To be able to manage a file system type we need a **superblock read-function**
- This function relies on the block-device driver of a device to instantiate the corresponding file system superblock in memory
- Each file system type has a superblock that needs to match its read function



The VFS startup

- This is the minmal startup path:
  - `vfs_caches_init()`
    - `mnt_init()`
      - `init_rootfs()`
      - `init_mount_tree()`

This tells we are instantiating at least one FS type – the **Rootfs**

- Typically, at least two different FS types are supported
  - Rootfs (file system in RAM)
  - Ext (in the various flavors)
- However, in principles, the Linux kernel could be configured such in a way to support no FS
- In this case, any task to be executed needs to be coded within the kernel (hence being loaded at boot time)

File system types data structures

- The description of a specific FS type is done via the structure `file_system_type` defined in `include/linux/fs.h`

- This structure keeps information related to
  - ➢ The actual file system type
  - ➢ A pointer to a function to be executed upon mounting the file system (superblock-read)

Moved to the `mount` filed in newer kernel versions

```
struct file_system_type {
   const char *name;
   int fs_flags;
   ……
   struct super_block *(*read_super) (struct
    super_block *, void *, int);
   struct module *owner;
   struct file_system_type * next;
   struct list_head fs_supers;
   ……
};
```

… newer kernel version alignment

```
struct file_system_type {
const char *name;
int fs_flags;
  …
  …
  struct dentry *(*mount) (struct file_system_type *,
      int, const char *, void *);
  void (*kill_sb) (struct super_block *);
  struct module *owner;
  struct file_system_type * next;
  …
  …
}
```

Beware this!!

Rootfs and basic fs-type API

- Upon booting, an instance of the structure `file_system_type` is allocated to keep meta-data for the **Rootfs**
- This file system only lives in main memory (hence it is re-initialized each time the kernel boots)
- The associated data act as initial “inspection” point for reaching additional file systems (starting from the root one)
- We can exploit kernel macros/functions in order to allocate/initialize a `file_system_type` variable for a specific file system, and to link it to a proper list
- The linkage one is

```
int register_filesystem(struct file_system_type *)
```

- Allocation of the structure keeping track of **Rootfs** is done statically within `fs/ramfs/inode.c`
- The linkage to the list is done by the function `init_rootfs()` defined in the same source file
- The name of the structured variable is `rootfs_fs_type`

```
int __init init_rootfs(void){

 return register_filesystem(&rootfs_fs_type);

}
```

Kernel 4.xx instance

```
static struct file_system_type rootfs_fs_type = {
        .name           = "rootfs",
        .mount          = rootfs_mount,
        .kill_sb        = kill_litter_super,
};

int __init init_rootfs(void)
{
        int err = register_filesystem(&rootfs_fs_type);

        if (err)
                return err;

        if (IS_ENABLED(CONFIG_TMPFS) && !saved_root_name[0] &&
                (!root_fs_names || strstr(root_fs_names, "tmpfs"))) {
                err = shmem_init();
                is_tmpfs = true;
        } else {
                err = init_ramfs_fs();
        }

        if (err)
                unregister_filesystem(&rootfs_fs_type);

        return err;
}
```

Creating and mounting the Rootfs instance

- Creation and mounting of the **Rootfs** instance takes place via the function `init_mount_tree()`
- The whole task relies on manipulating 4 data structures
  - `struct vfsmount`
  - `struct super_block`
  - `struct inode`
  - `struct dentry`
- The instances of `struct vfsmount` and `struct super_block` keep file system proper information (e.g. in terms of relation with other file systems)
- The instances of `struct inode` and `struct dentry` are such that one copy exits for any file/directory of the specific file system

The structure `vfs_mount` (still in place in 3.xx)

```
struct vfsmount
{
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent;      /*fs we are mounted on */
    struct dentry *mnt_mountpoint;    /*dentry of mountpoint */
    struct dentry *mnt_root;          /*root of the mounted tree*/
    struct super_block *mnt_sb;       /*pointer to superblock */
    struct list_head mnt_mounts;      /*list of children, anchored
                                                        here */
    struct list_head mnt_child;      /*and going through their
                                                        mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname;                /* Name of device e.g.
                                                /dev/dsk/hda1 */
    struct list_head mnt_list;
};
```

…. now structured this way in 4.xx

```
struct vfsmount {
        struct dentry *mnt_root; /* root of the mounted tree */
        struct super_block *mnt_sb; /* pointer to superblock */
        int mnt_flags;
} __randomize_layout;
```

This feature is supported by the `randstruct` plugin
Let's look at the details …….

`randstruct`

- Access to any field of a structure is based on compiler rules when relying on classical ‘.’ or ‘`->`’ operators
- Machine code is therefore generated in such a way to correctly displace into the proper filed of a structure
- `__randomize_layout` introduces a reshuffle of the fields, with the inclusion of padding
- This is done based on pseudo random values selected at compile time
- Hence an attacker that discovers the address of a structure but does not know what’s the randomization, will not be able to easily trap into the target field
- Linux usage (stable since kernel 4.8):
  - ✓ on demand (via `__randomize_layout`)
  - ✓ by default on any struct only made by function pointers (a driver!!!)
  - ✓ the latter can be disabled with `__no_randomize_layout`

The structure super_block

```
struct super_block {
        struct list_head        s_list;   /* Keep this first */
        ……
        unsigned long           s_blocksize;
        ……
        unsigned long long      s_maxbytes;    /* Max file size */
        struct file_system_type        *s_type;
        struct super_operations        *s_op;
        ……
        struct dentry           *s_root;
        ……
        struct list_head        s_dirty;       /* dirty inodes */
        ……
        union {
                struct minix_sb_info   minix_sb;
                struct ext2_sb_info    ext2_sb;
                struct ext3_sb_info    ext3_sb;
                struct ntfs_sb_info    ntfs_sb;
                struct msdos_sb_info   msdos_sb;
                ……
                void                   *generic_sbp;
        } u;
        ……
};
```

The structure `dentry`

```
struct dentry {
        atomic_t d_count;
        ......
        struct inode  * d_inode;   /* Where the name belongs to */
        struct dentry * d_parent;  /* parent directory */
        struct list_head d_hash;   /* lookup hash list */
        ......
        struct list_head d_child;  /* child of parent list */
        struct list_head d_subdirs;       /* our children */
        ......
        struct qstr d_name;
        ......
        struct dentry_operations  *d_op;
        struct super_block * d_sb; /* The root of the dentry tree */
        unsigned long d_vfs_flags;
        ......
        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

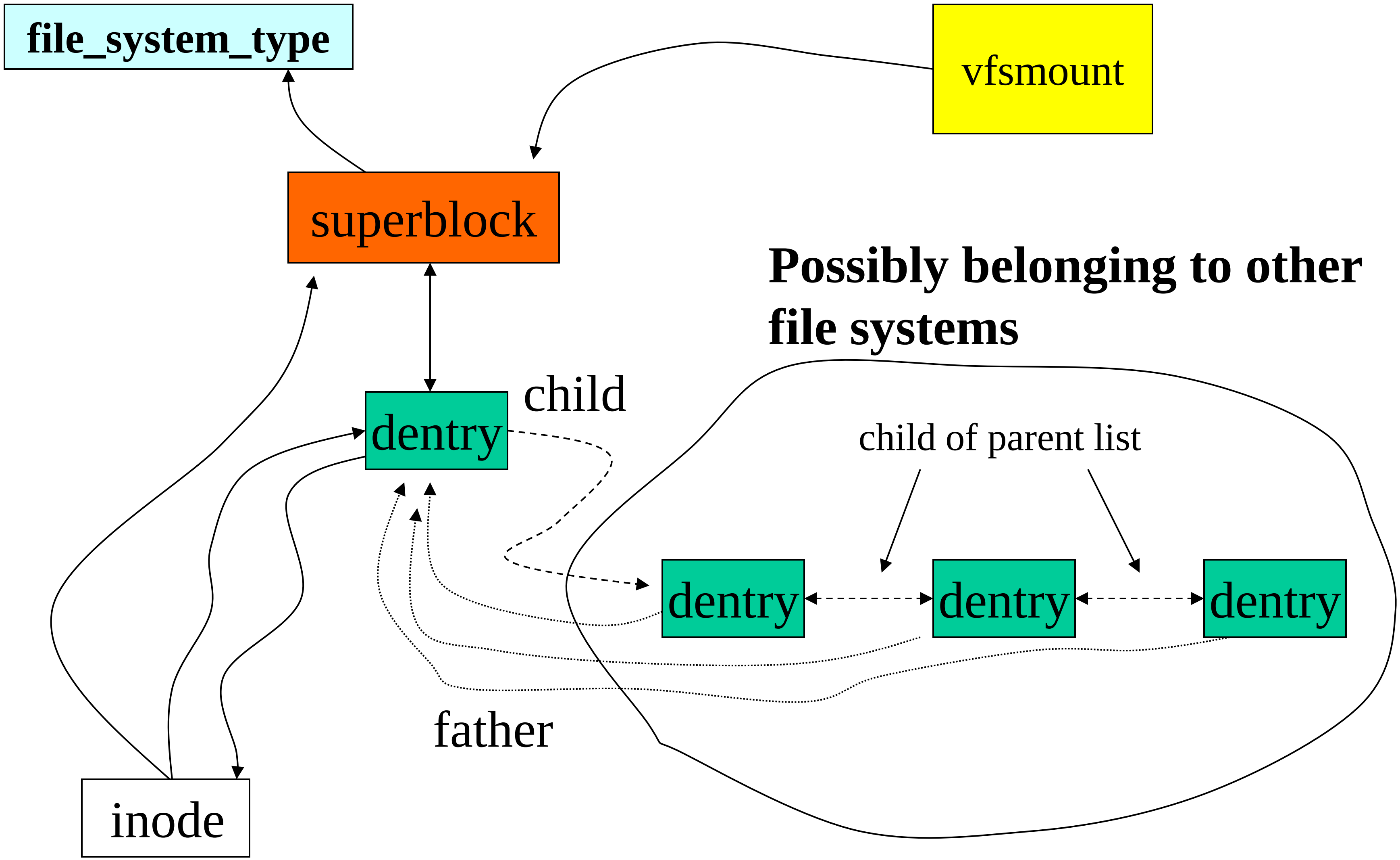
This is for “short” names

The structure `inode` (a bit more fields are in kernel 4.xx)

```
struct inode {
        ......
        struct list_head i_dentry;
        ......
        uid_t                       i_uid;
        gid_t                       i_gid;
        ......
        unsigned long               i_blksize;
        unsigned long               i_blocks;
        ......
        struct inode_operations     *i_op;
        struct file_operations              *i_fop;
        struct super_block                  *i_sb;
        wait_queue_head_t           i_wait;
        ......
        union {
                ......
                struct ext2_inode_info              ext2_i;
                struct ext3_inode_info              ext3_i;
                ......
                struct socket                       socket_i;
                ......
                void                                *generic_ip;
        } u;
};
```

Beware this!!

Overall scheme



Initializing the Rootfs instance

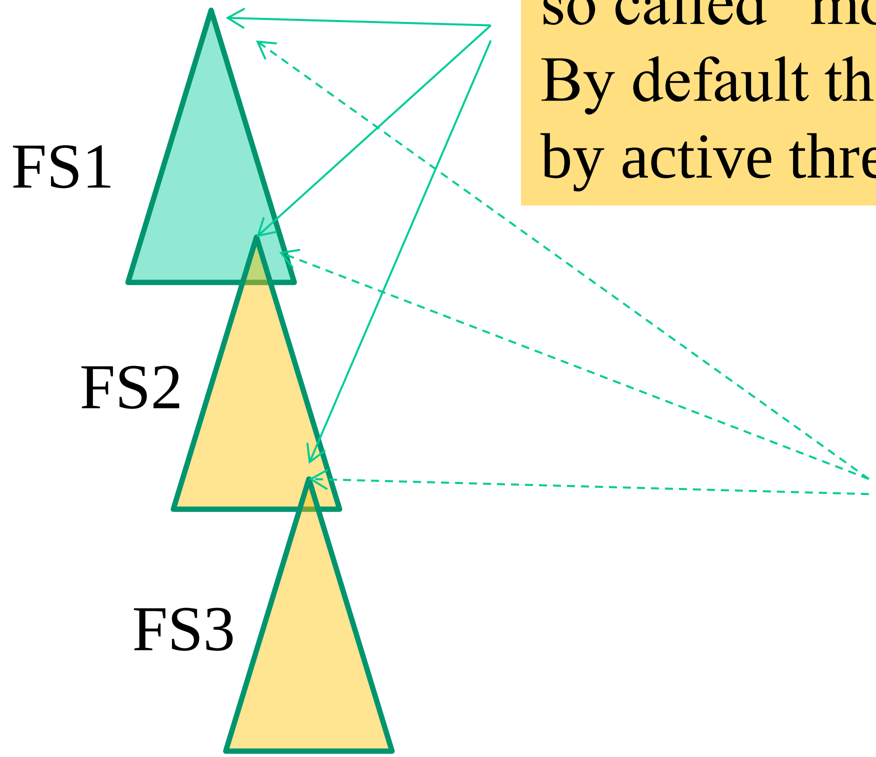
- The main tasks, carried out by `init_mount_tree()`, are
  1. Allocation of the 4 data structures for **Rootfs**
  2. Linkage of the data structures
  3. Setup of the name “/” for the root of the file system
  4. Linkage between the IDLE PROCESS and Rootfs
- The first three tasks are carried out via the function `do_kern_mount()` which is in charge of invoking the execution of the super-block read-function for **Rootfs**
- Linkage with the IDLE PROCESS occurs via the functions `set_fs_pwd()` and `set_fs_root()`

```
static void __init init_mount_tree(void)
{
        struct vfsmount *mnt;
        struct namespace *namespace;
        struct task_struct *p;

        mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
        if (IS_ERR(mnt))
                panic("Can't create rootfs");
        .........

        set_fs_pwd(current->fs, namespace->root,
                                    namespace->root->mnt_root);
        set_fs_root(current->fs, namespace->root,
                                    namespace->root->mnt_root);
}
```

…. very minor changes of this function are in kernel 4.xx

FS mounting and namespaces

The list of mount points along the three is a so called “mount namespace”
By default the “initial namespace” is seen by active threads



We can make a thread start working with a new mount namespace which is initially a copy of another one

Moving to another mount namespace makes `mount/unmount` operations only acting on the current namespace (except if the mount operation is tagged with SHARED)

Actual system calls for mount namespaces

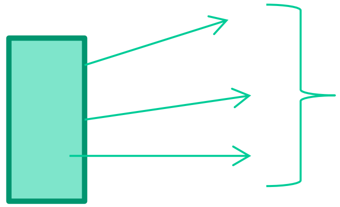
`clone(... int flags ...)`

`CLONE_NEWNS`

`unshare(int flags)`

An overall view

Super operations

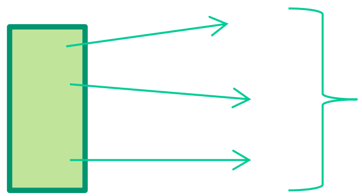


Update superblock (an flush on device)

Get superblock info (e.g. `statfs/fstatfs`)

Manage i-nodes (read/write them from/ to superlock)

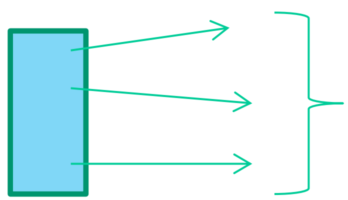
Dentry operations



Allocate/deallocate dentries

Link them to other data structures

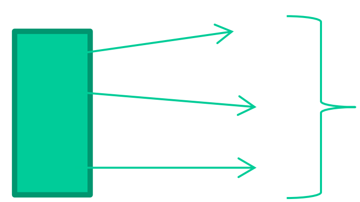
i-node operations



`creat/link/unlink`

Lookup

file operations



Actual operations on data