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

Virtual file system

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

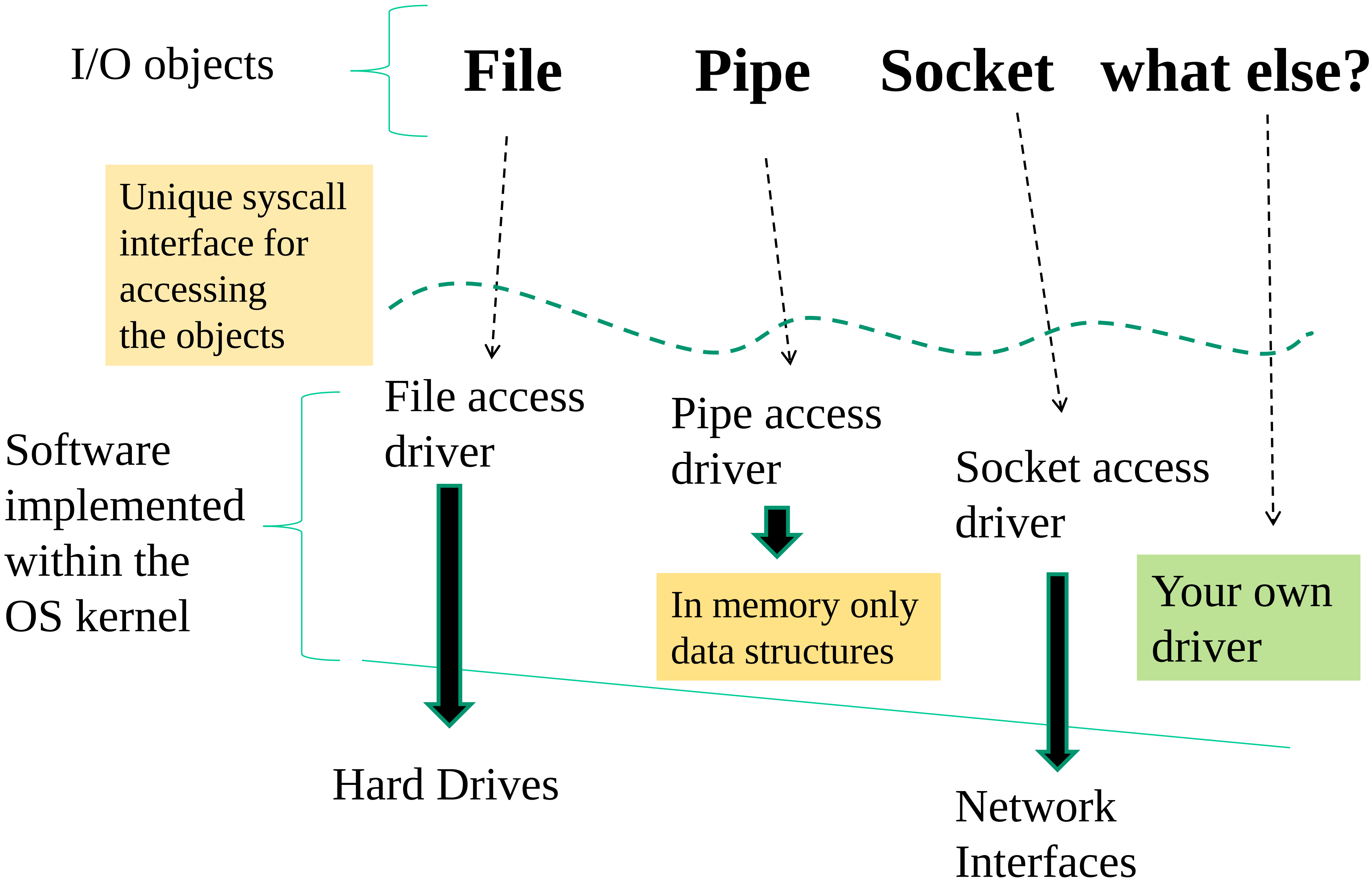
Connections

- 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

- **<u>Devices can be seen as files</u>**
- What we drive, in terms of state update, is <u>the structure used to represent the device in memory</u>
- 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

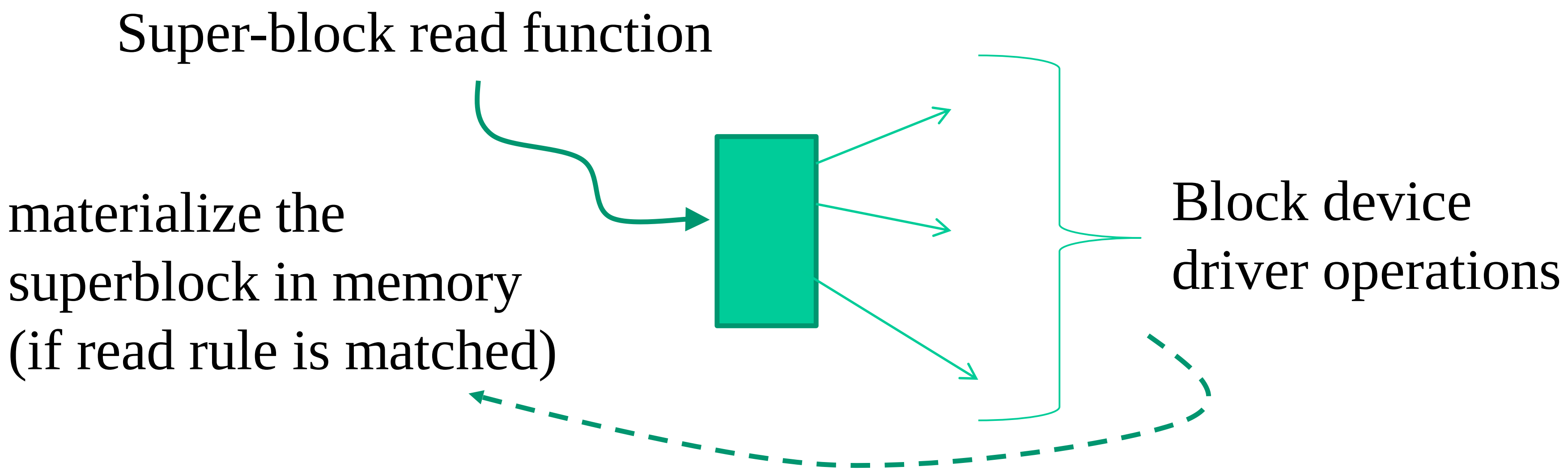
- Files are backed by data on a hard drive
- What **software modules do we need** for managing files on that hard drive in a well shaped OS-kernel??
    1. A function to read the device superblock for determining what files exist and where their data are
    2. A function to read device blocks for bringing 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 device drivers

- 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 tables of function pointers, 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**

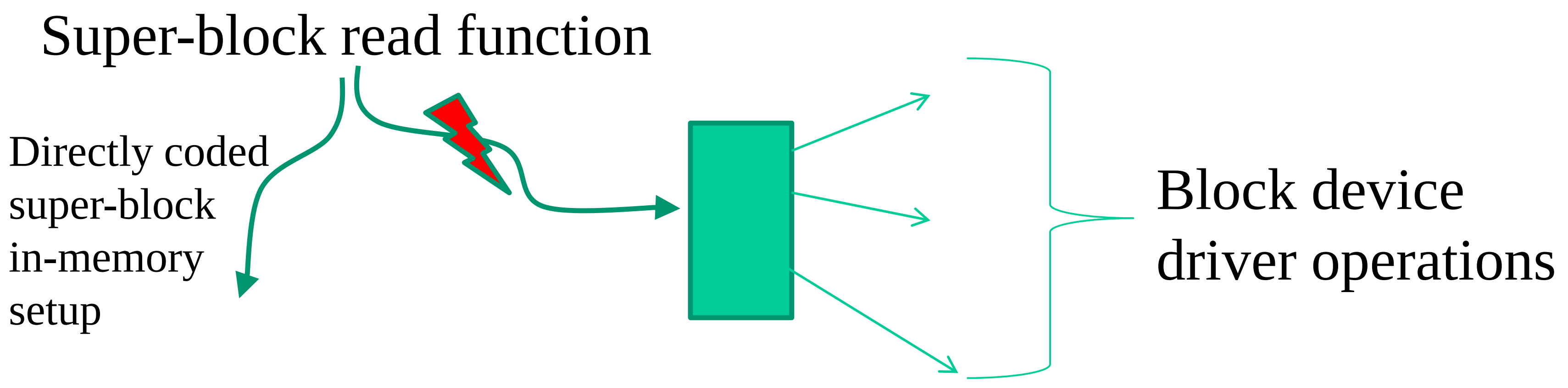
File system types in Linux

- To be able to manage a file system type we need a **<u>superblock read function</u>**
- 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


Super-block read function
materialize the superblock in memory (if read rule is matched)
Block device driver operations

What about RAM file systems?

- These are file systems whose data disappear at system shutdown
- On the basis of what described before, these file systems **<u>do not have an on-device</u>** representation
- Their superblock read function does not really need to read blocks from a device
- It typically relies on in-memory instantiation of a fresh superblock representing the new incarnation of the file system


Super-block read function
Directly coded super-block in-memory setup
Block device driver operations

The VFS startup

- This is the minimal 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` field 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, a compile time defined instance of the structure `file_system_type` keeps 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

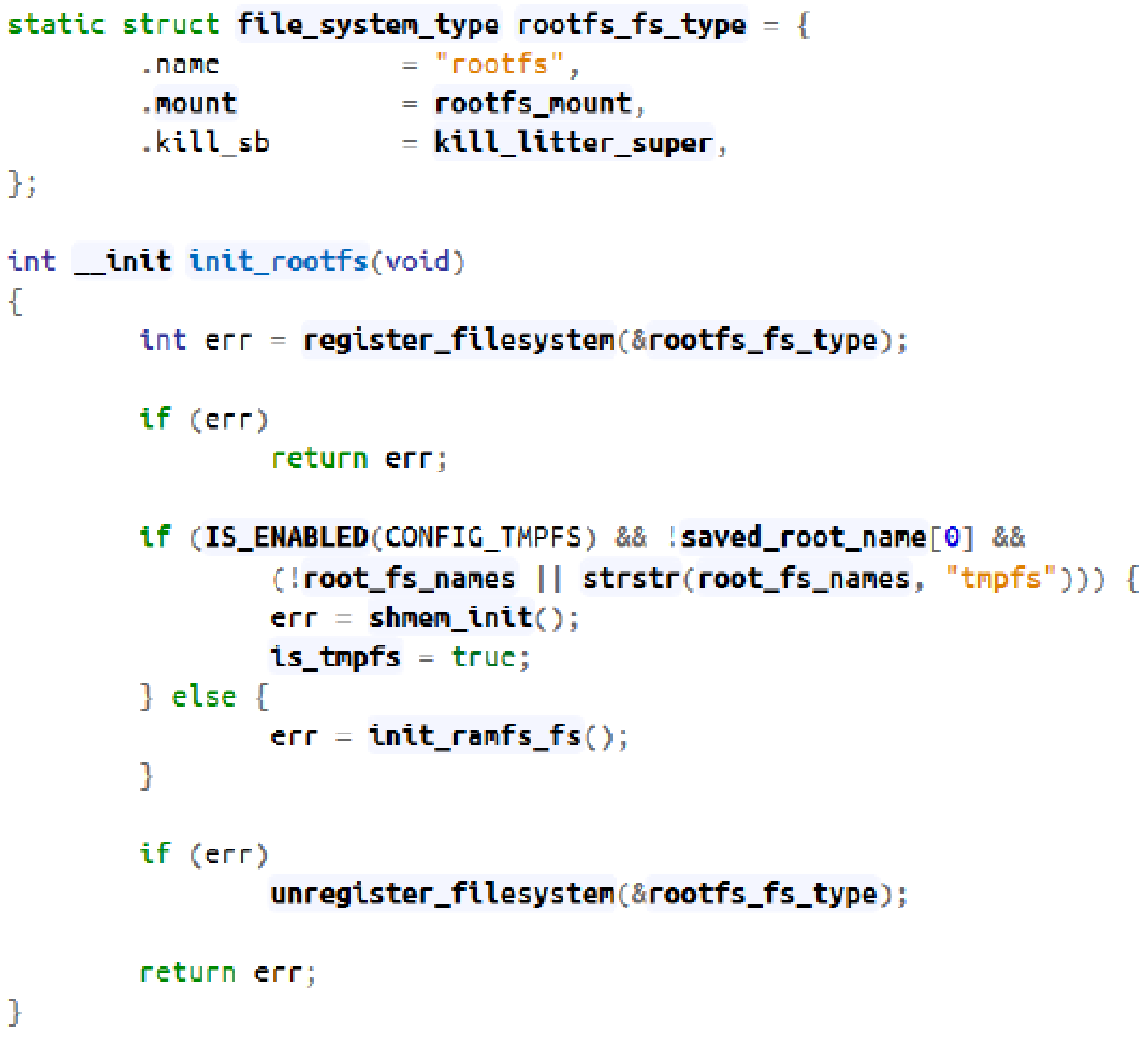
```
int register_filesystem(struct file_system_type *)
```

- Allocation of the structure keeping track of **Rootfs** is done statically (compile time)
- The linkage to the list is done by the function `init_rootfs()`
- The name of the structured variable is `rootfs_fs_type`

```
int __init init_rootfs(void){

    ...
    register_filesystem(&rootfs_fs_type);
    ...
}
```

A few modifications in the structure of init_rootfs() are in kernel 5

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 `vfsmount` (still in place in kernel 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 kernel 4.xx or later

```
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 field 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** (a few variants in very recent kernels)

```
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` (a few minor variants in very recent kernels)

```
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 */
};
```

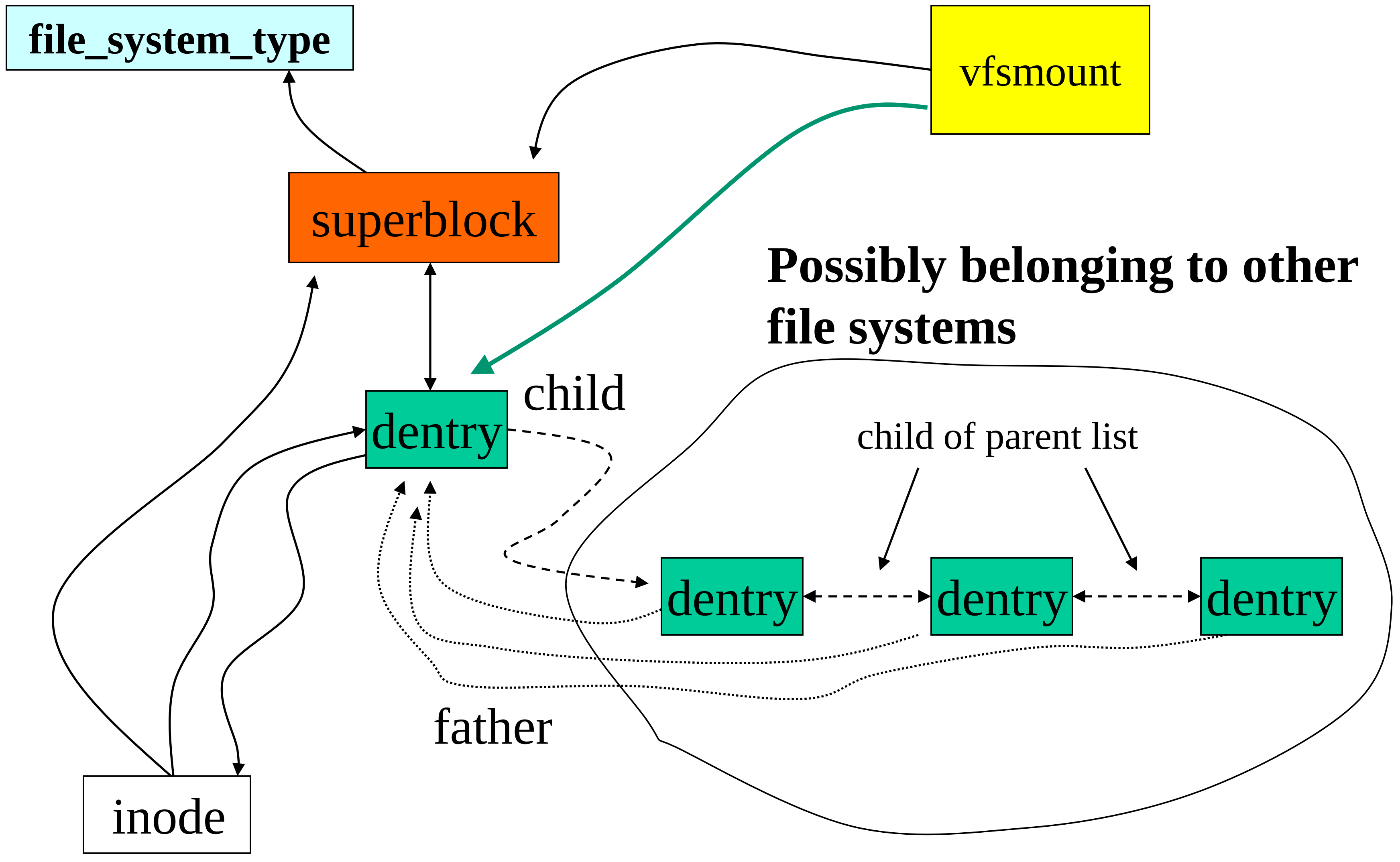
This is for “short” names

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

```
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



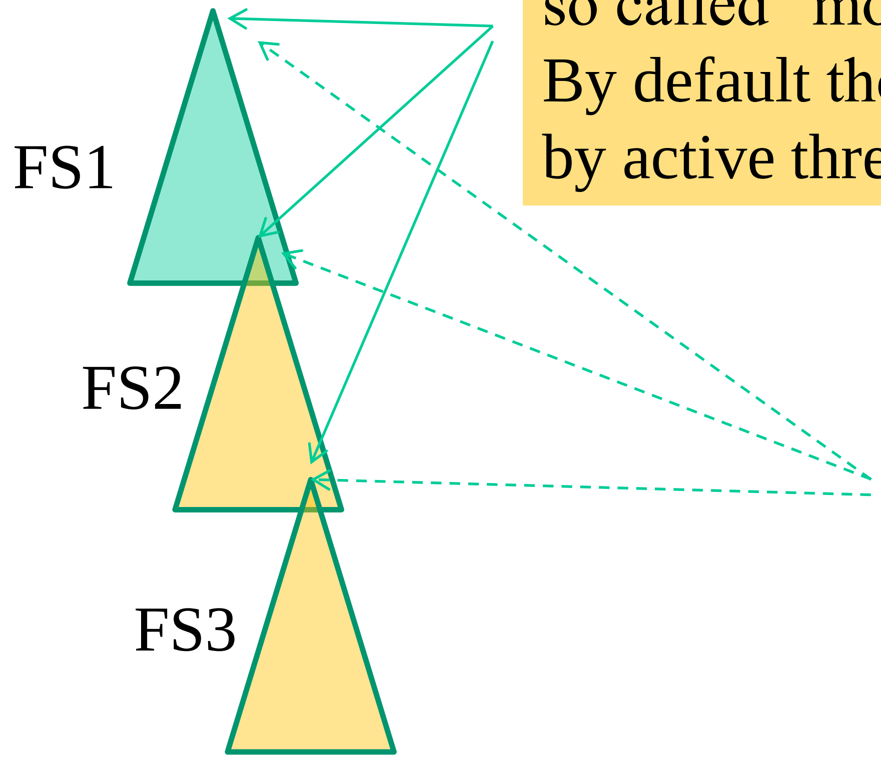
Initializing the Rootfs instance

- 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()` or `vfs_kern_mount()`, which are 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/5.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

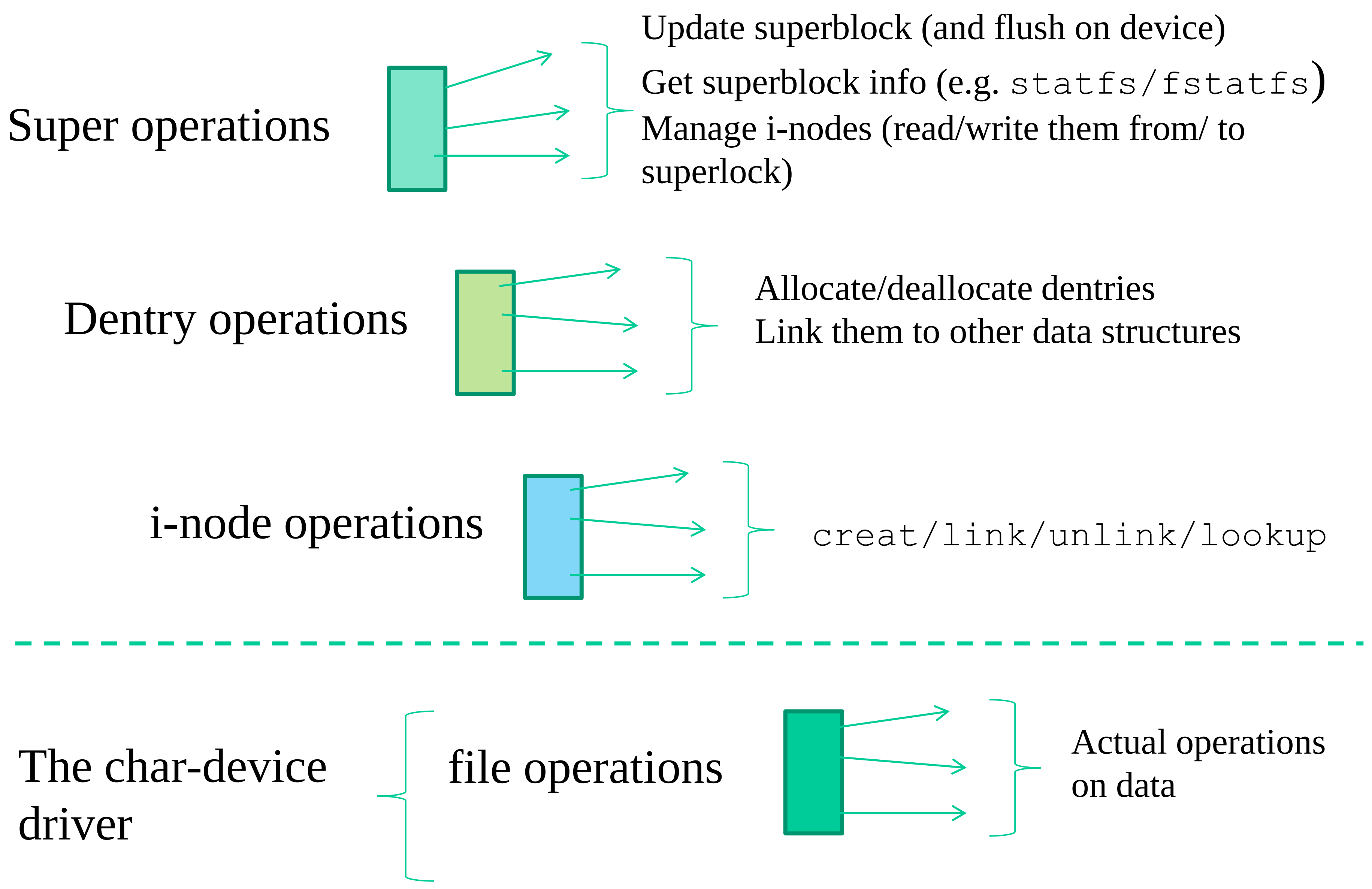
We can make a thread start working with a new mount namespace which is initially a copy of another one

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

$$\begin{tikzcd}
\texttt{clone(... int flags ...)} & \\
& \texttt{CLONE\_NEWNS} \arrow[ul] \arrow[dl] \\
\texttt{unshare(int flags)} &
\end{tikzcd}$$

An overall view



Super operations

- Update superblock (and 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`

The char-device driver

file operations

- Actual operations on data

VFS vs TCBs (2.4 style)

- The TCB keeps the field `struct fs_struct *fs` pointing to information related to the current directory and the root directory for the associated process
- `fs_struct` is defined as follows in `include/fs_struct.h`

```
struct fs_struct {
    atomic_t count;
    rwlock_t lock;
    int umask;
    struct dentry * root, * pwd, * altroot;
    struct vfsmount * rootmnt, * pwdmnt,
                                * altrootmnt;
};
```

3.xx/4.7 kernel style

See include/linux/fs_struct.h

```
struct fs_struct {
        int users;
        spinlock_t lock;
        seqcount_t seq;
        int umask;
        int in_exec;
        struct path root, pwd;
};
```

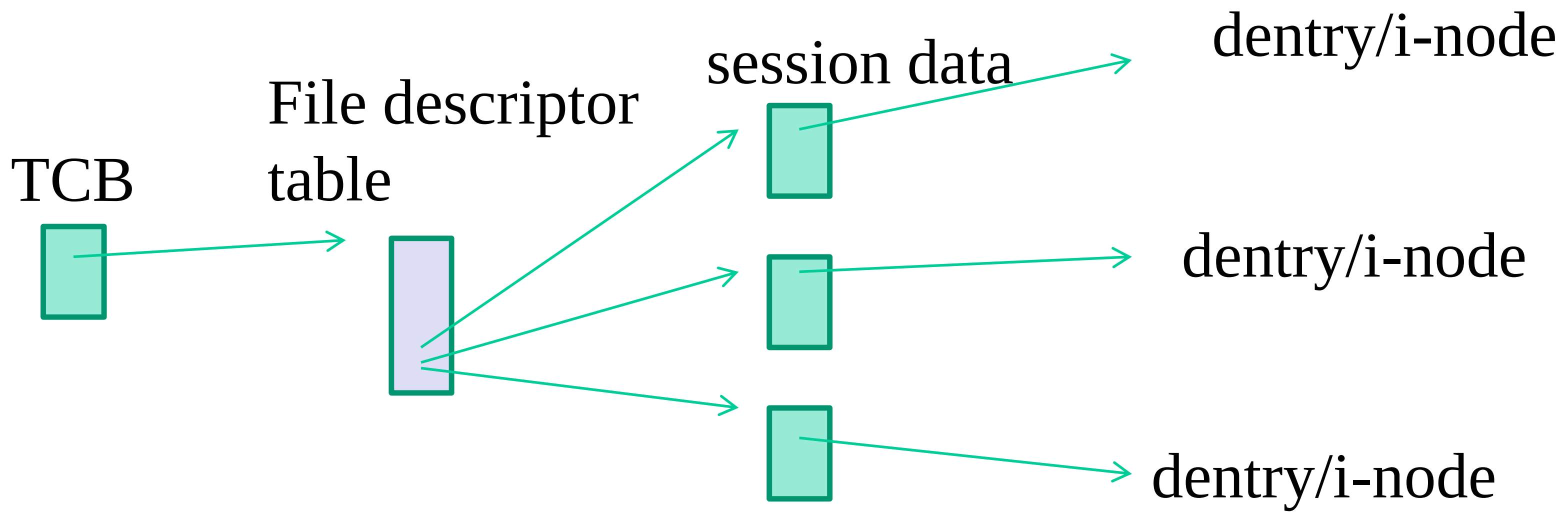
… and then 4.8 or later style

```
struct fs_struct {
        int users;
        spinlock_t lock;
        seqcount_t seq;
        int umask;
        int in_exec;
        struct path root, pwd;
} __randomize_layout;
```

Towards more security

File descriptor table

- It builds a **<u>relation between an I/O channel</u>** (a numerical ID code) and **<u>an I/O object</u>** we are currently working with along an I/O session
- Hence it enables fast search of the data structures used to represent I/O objects and I/O sessions
- The search is based on the channel ID as the key
- The actual implementation of the layout for the file descriptor table is clearly system specific
- In Linux we have the below scheme



File descriptor table (a few variations in very recent kernel versions)

- TCB keeps the field `struct files_struct *files` which points to the descriptor table

- This table is defined in as

```
struct files_struct {
    atomic_t count;
    rwlock_t file_lock; /* Protects all the below
                           members.  Nests
                    inside tsk->alloc_lock */
    int max_fds;
    int max_fdset;
    int next_fd;
    struct file ** fd;  /* current fd array */
    fd_set *close_on_exec;
    fd_set *open_fds;
    fd_set close_on_exec_init;
    fd_set open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};
```

$\leftarrow$ bitmap for close on exec flags (close_on_exec)

$\leftarrow$ bitmap identifying open fds (open_fds)

The session data: `struct file` (the very classical shape)

```
struct file {
  struct list_head   f_list;
  struct dentry      *f_dentry;
  struct vfsmount   *f_vfsmnt;
  struct file_operations   *f_op;
  atomic_t           f_count;
  unsigned int       f_flags;
  mode_t             f_mode;
  loff_t             f_pos;
  unsigned long      f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct fown_struct f_owner;
  unsigned int       f_uid, f_gid;
  int                f_error;
  unsigned long      f_version;
  /* needed for tty driver, and maybe others */
  void               *private_data;
  /* preallocated helper kiobuf to speedup O_DIRECT */
  struct kiobuf      *f_iobuf;
  long               f_iobuf_lock;
};
```

3.xx/4.xx/5.xx style (quite similar to 2.4)

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head          fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
        struct inode            *f_inode;       /* cached value */
        const struct file_operations     *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t            f_count;
        unsigned int             f_flags;
        fmode_t                  f_mode;
        struct mutex             f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct       f_owner;
        const struct cred        *f_cred;
        struct file_ra_state     f_ra;

.........
..........  __randomize_layout;;
```

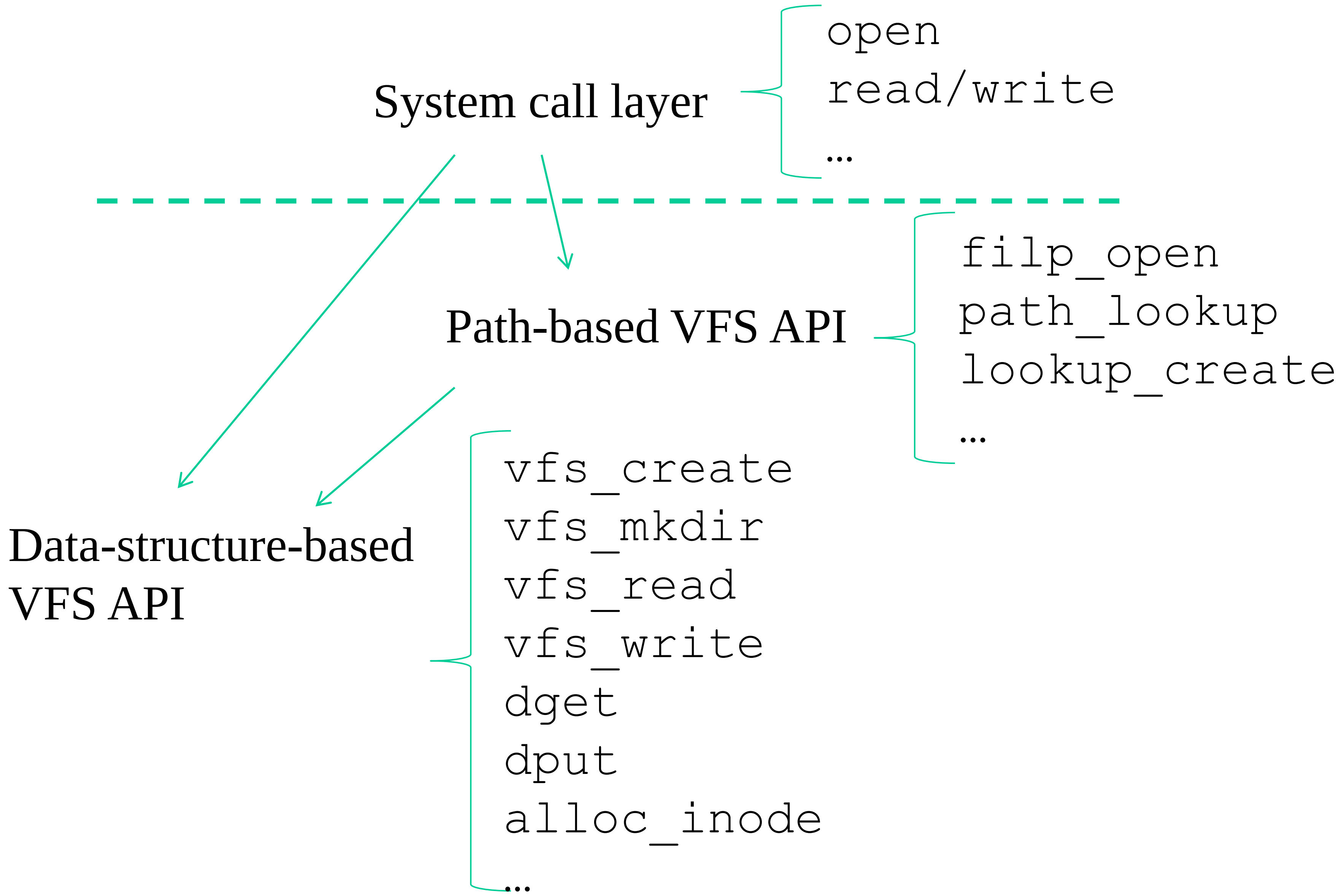
Now we have randomized layout and a few fields are moved to other pointed tables

Randomized from kernel 4.8

Linux VFS API layering

- System call layer
  - ✓ Session setup
  - ✓ Channel ID based data access/manipulation
- Path-based VFS layer
  - ✓ Do something on file system based on a path passed as parameter
- Data structure based VFS layer
  - ✓ Do something on file system based on pointers to data structures

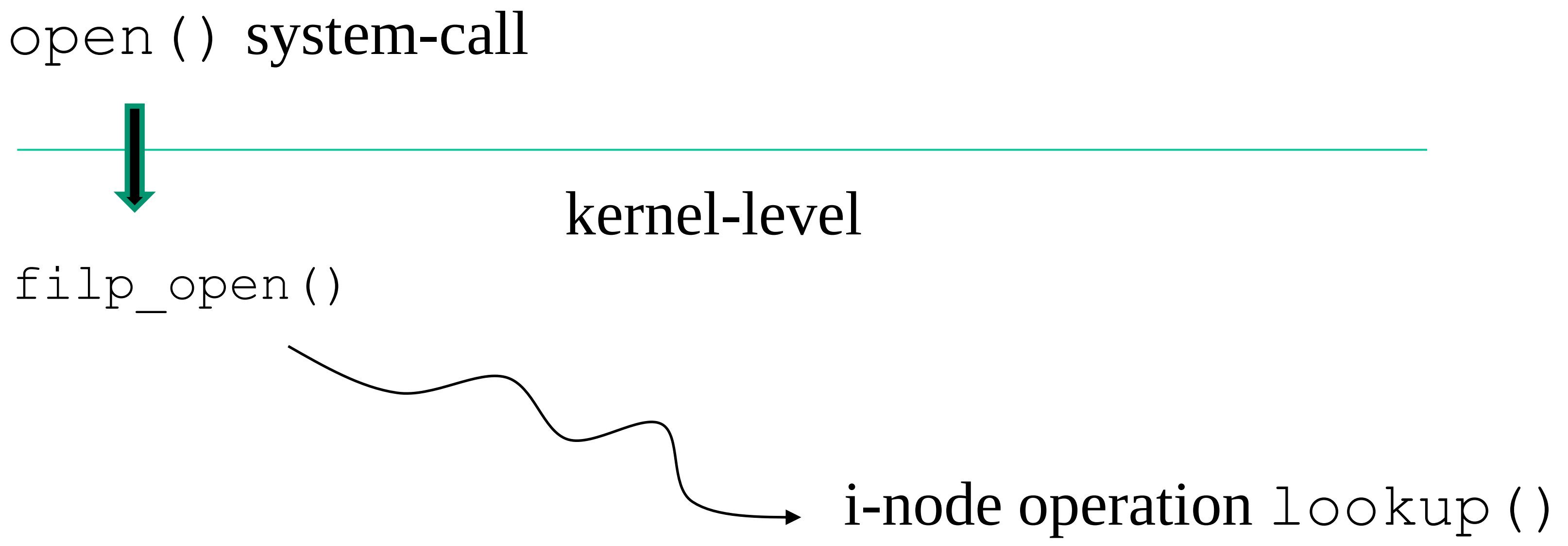
Relations


System call layer
open
read/write
...
Path-based VFS API
filp_open
path_lookup
lookup_create
...
Data-structure-based VFS API
vfs_create
vfs_mkdir
vfs_read
vfs_write
dget
dput
alloc_inode
...

Path-based API examples

```
struct file *filp_open(const char * filename, int
flags, int mode)
```

returns the address of the `struct file` associated with the opened file



In the end we pass trough dentry/i-node/char-dev/superblock drivers

Data-structure based API examples

```
int vfs_mkdir(struct inode *dir, struct dentry  *dentry,
 int mode)
```

Creates an i-node and associates it with `dentry`. The parameter `dir` is used to point to a parent i-node from which basic information for the setup of the child is retrieved. `mode` specifies the access rights for the created object

```
int vfs_create(struct inode *dir, struct dentry *dentry,
                  int mode)
```

Creates an i-node linked to the structure pointed by `dentry`, which is child of the i-node pointed by `dir`. The parameter `mode` corresponds to the value of the permission mask passed in input to the open system call. Returns 0 in case of success (it relies on the i-node-operation `create`)

```
static __inline__ struct dentry * dget(struct   dentry
 *dentry)
```

Acquires a dentry (by incrementing the reference counter)

```
void dput(struct dentry *dentry)
```

Releases a dentry (this module relies on the dentry operation `d_delete`)

… still on data-structure based API examples

```
ssize_t vfs_read(struct file *file, char __user *buf,
size_t count, loff_t *pos)

ssize_t vfs_write(struct file *file, char __user
*buf, size_t count, loff_t *pos)
```

file operation `read(......)`
file operation `write(......)`

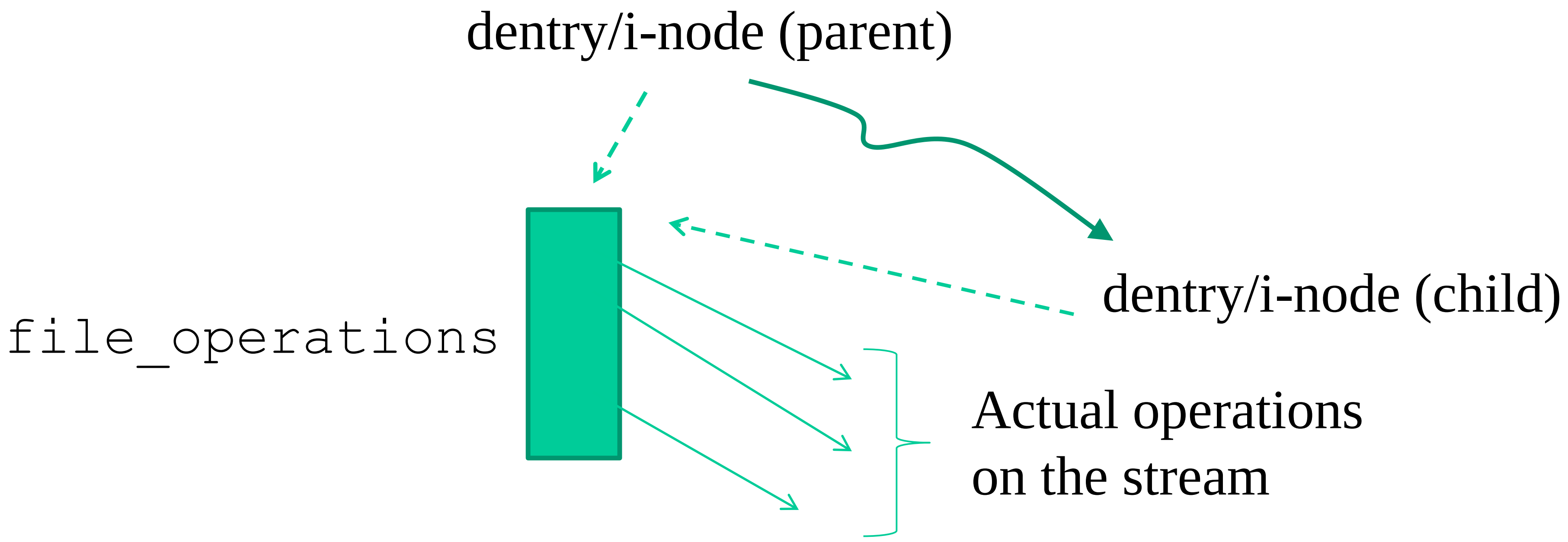
In the end we traverse dentry/i-node structures to retrieve the file operations table associated with that dentry

Relating I/O objects and drivers: the MAJOR number

- A driver (for either a block or a char device) is registered into so called device-drivers table
- The table is an array and the displacement into the array where the driver is registered is called MAJOR number
- Suppose we have to instantiate in memory the dentry/i-node of a file belonging to a specific file system type, then we need to:
  - ✓ Identify the char-dev driver for operating on the file (this will depend on where we registered the driver for files of that file system into the table)
  - ✓ Link the dentry/i-node to that driver (recall a char-device driver is a table of file-operations

Lets' simplify the job

- Suppose we instantiate in memory a dentry/i-node that depends on another one on the same file system
- They are "homogeneous"
- In this case we simply inherit the same char-device driver of the parent



What about data isolation?

- Generally the i-node identifies what data are touched by a call to a function in `file_operations`
- This might not be the case with generic I/O objects that are not regular files
- As an example, what about things that are not files??
- We may have an I/O object that
  - ✓ Can be managed by a given char-device driver
  - ✓ Can be an instance in a group of many that need to be driven by the same char-device driver (they are homogeneous but are not regular files)

VFS "nodes" and device numbers

- The field `umode_t i_mode` within `struct inode` keps an information indicating the type of the i-node, e.g.:
  - directory
  - file
  - char device
  - block device
  - (named) pipe
- The kernel function `sys_mknod()` allows creating an i-node associated with a generic type
- In case the i-inode represents a device, the operations for managing the device are retrieved via the device driver tables
- Particularly, the i-node keeps the field `kdev_t i_rdev` which logs information related to both **MAJOR and MINOR** numbers for the device

`The mknod()` system call

```
int mknod(const char *pathname, mode_t
                    mode, dev_t dev)
```

- `mode` specifies the permissions to be used and the type of the node to be created
- permissions are filtered via the `umask` of the calling process `(mode & umask)`
- several different macros can be used for defining the node type: `S_IFREG,  S_IFCHR,  S_IFBLK, S_IFIFO`
- when using `S_IFCHR` or `S_IFBLK`, the parameter `dev` specifies **MAJOR and MINOR numbers for the device file that gets created**, otherwise this parameter is a don’t care

Device numbers

- for x86 machines, device numbers are represented as bit masks
- MAJOR corresponds to the least significant byte within the mask
- MINOR corresponds to the second least significant byte within the mask
- The macro `MKDEV(ma,mi)`, which is defined in `include/linux/kdev_t.h`, can be used to setup a correct bit mask by starting from the two numbers

Usage of MINOR numbers in drivers

- The functions belonging to the driver take a pointer to `struct file` in input
- Therefore we know the session – the dentry – and the i-node ...
- …. hence we know the MINOR!
- …. and we can do stuff based on the MINOR!
- … as an example we might have that the driver manages an array of tables, each associated with the state of an I/O object with a given MINOR (an index)

The Linux block devices table (classical style)

```
static struct {
 const char *name;
 struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];
```

- In `fs/block_devices.c` we can find the below functions for registering/deregistering the driver

```
int register_blkdev(unsigned int major,
    const char * name, struct
    block_device_operations *bdops)

int unregister_blkdev(unsigned int major,
const char * name)
```

`struct block_device_operations`
(a bit more fields in very recent kernel versions)

```
struct block_device_operations {
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*ioctl) (struct inode *, struct file *,
                      unsigned, unsigned long);
   int (*check_media_change) (kdev_t);
   int (*revalidate) (kdev_t);
   struct module *owner;
};
```

Char devices table

```
struct device_struct {
        const char * name;
        struct file_operations * fops;
};
```

Device name

Device operations

```
static struct device_struct chrdevs[MAX_CHRDEV];
```

- in `fs/devices.c` we can find the following functions for registering/deregistering a driver

  `int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)`

  Registration takes place onto the entry at displacement MAJOR (0 means the choice is up to the kernel). The actual MAJOR number is returned

  `int unregister_chrdev(unsigned int major, const char * name)`

  Releases the entry at displacement MAJOR

struct file_operations
(a bit more fields in very recent kernel versions)

```
sruct 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 *);
 ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                                loff_t *, int);
 unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                        unsigned long, unsigned long, unsigned long);
};
```

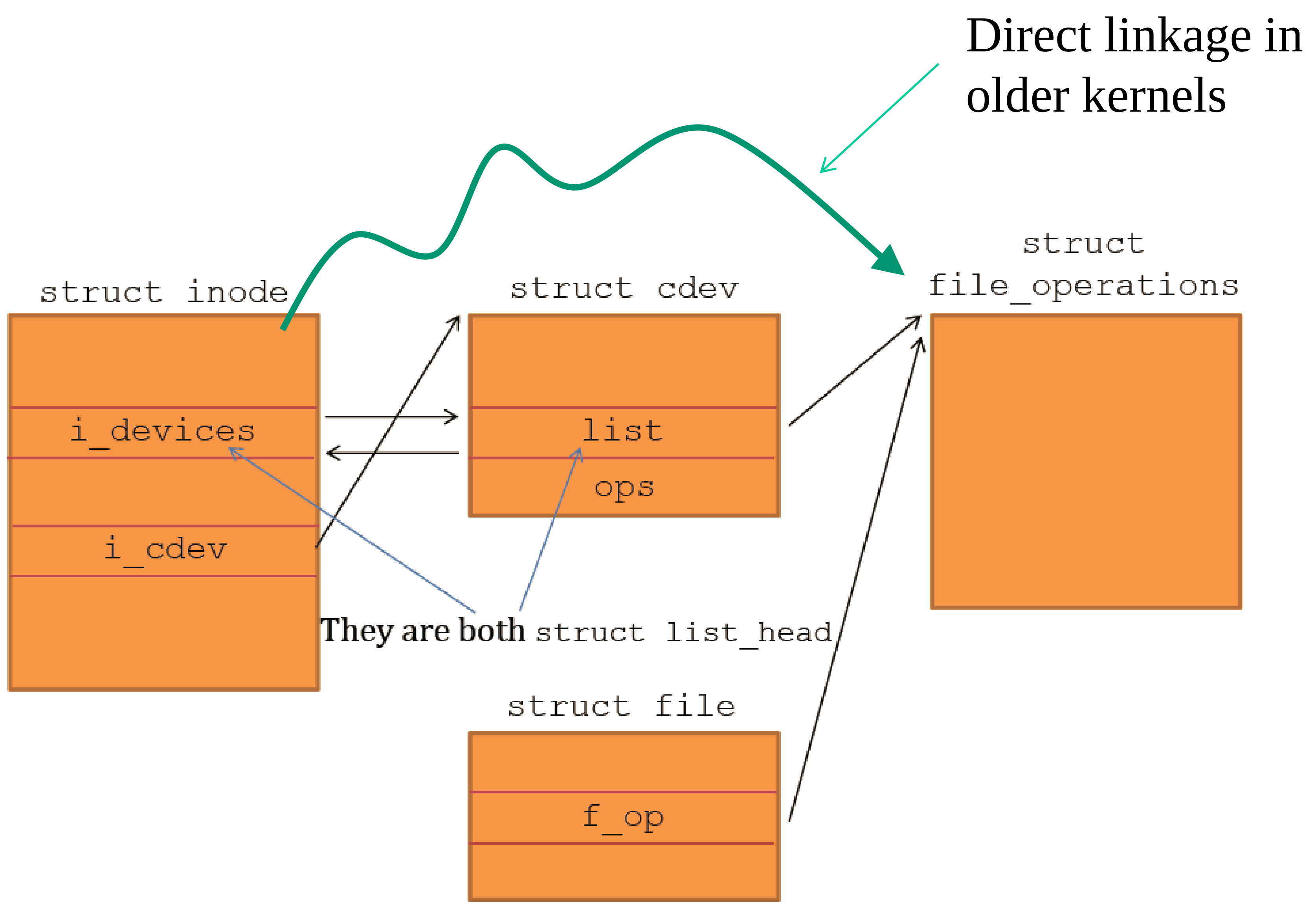
Kernel 3 or later: augmenting flexibility and structuring

```
#define CHRDEV_MAJOR_HASH_SIZE 255
static struct char_device_struct {
      struct char_device_struct *next;
      unsigned int major;
      unsigned int baseminor;
      int minorct;
      char name[64];
      struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Minor number ranges already indicated and flushed to the `cdev` table

Pointer to file-operations is here

A scheme on i-node to file operations mapping for kernel 3 or later



Operations remapping

```
int register_chrdev(unsigned int major, const
char *name, struct file_operations *fops)
```

New features

```
int __register_chrdev(unsigned int major,
unsigned int baseminor, unsigned int count,
const char *name, const struct file_operations
*fops)
```

```
int unregister_chrdev(unsigned int major, const char
*name)
```

```
void __unregister_chrdev(unsigned int major,
unsigned int baseminor, unsigned int count,
const char *name)
```

Final part of the boot (activating the INIT thread - 2.4 style)

- The last function invoked while running `start_kernel()` is `rest_init()` and is defined in `init/main.c`
- This function spawns INIT, which is initially created as a kernel level thread, and eventually activates the l'IDLE PROCESS function

```
static void rest_init(void)
{
    kernel_thread(init, NULL, CLONE_FS |
                CLONE_FILES | CLONE_SIGNAL);
    unlock_kernel();
    current->need_resched = 1;
    cpu_idle();
}
```

… and 3.xx or later style

see linux/init/main.c

```
static noinline void __init_refok rest_init(void)
{
            int pid;

            rcu_scheduler_starting();
            /*
             * We need to spawn init first so that it obtains pid 1, however
             * the init task will end up wanting to create kthreads, which, if
             * we schedule it before we create kthreadd, will OOPS.
*/
             kernel_thread(kernel_init, NULL, CLONE_FS);
            ...........
                numa_default_policy();
....... .
.... .
```

Switch off round-robin to first-touch

The `mount_root()` function

```
static void __init mount_root(void)
{
        ……
        create_dev("/dev/root", ROOT_DEV,
                                   root_device_name);
        ……
        mount_block_root("/dev/root", root_mountflags);
}

static int __init create_dev(char *name, kdev_t dev,
        char *devfs_name)
{
        void *handle;
        char path[64];
        int n;

        sys_unlink(name);
        if (!do_devfs)
                return sys_mknod(name, S_IFBLK|0600,
                                         kdev_t_to_nr(dev));
        ……
}
```

The function `init()`

- The `init()` function for INIT is defined in `init/main.c`
- This function is in charge of the following main operations
  - ➢ Mount of ext2 (or the reference root file system)
  - ➢ Activation of the actual INIT process (or a shell in case of problems)

```
static int init(void * unused){
  struct files_struct *files;
  lock_kernel();
  do_basic_setup();
  prepare_namespace();
  .........
  if (execute_command)  run_init_process(execute_command);
  run_init_process("/sbin/init");
  run_init_process("/etc/init");
  run_init_process("/bin/init");
  run_init_process("/bin/sh");
  panic("No init found.  Try passing init= option to
          kernel.");
}
```

registering drivers

The `prepare_namespace()` function (2.4 style - minor variations are in kernels 3/4/5)

```
void prepare_namespace(void){
 ......
 sys_mkdir("/dev", 0700);
 sys_mkdir("/root", 0700);
 sys_mknod("/dev/console", S_IFCHR|0600,
                              MKDEV(TTYAUX_MAJOR, 1));
 ......
 mount_root();
out:
 ......
 sys_mount(".", "/", NULL, MS_MOVE, NULL);
 sys_chroot(".");
 ......
}
```

The scheme

This is the typical state before calling `mount_root()`

$$\begin{tikzcd}
 & / \arrow[dl, bend right] \arrow[dr, bend left] & \\
\text{dev} \arrow[d, bend left] & & \text{root} \\
\text{console} & &
\end{tikzcd}$$

all in Rootfs (in RAM)

The function `mount_block_root()`

```
static void __init mount_block_root(char *name, int flags) {
       char *fs_names = __getname(); char *p;
       get_fs_names(fs_names);
retry:  for (p = fs_names; *p; p += strlen(p)+1) {
           int err = sys_mount(name, "/root", p, flags, root_mount_data);
            switch (err) {
                       case 0: goto out;
                       case -EACCES:  flags |= MS_RDONLY;  goto retry;
                       case -EINVAL:
                        case -EBUSY:  continue;
            }
       printk ("VFS: Cannot open root device \"%s\" or %s\n",
                       root_device_name, kdevname (ROOT_DEV));
       printk ("Please append a correct \"root=\" boot option\n");
       panic("VFS: Unable to mount root fs on %s", kdevname(ROOT_DEV));
       }
       panic("VFS: Unable to mount root fs on %s", kdevname(ROOT_DEV));
out:    putname(fs_names);
       sys_chdir("/root");
       ROOT_DEV = current->fs->pwdmnt->mnt_sb->s_dev;
       printk("VFS: Mounted root (%s filesystem)%s.\n",
              current->fs->pwdmnt->mnt_sb->s_type->name,
              (current->fs->pwdmnt->mnt_sb->s_flags & MS_RDONLY) ?
              " readonly" : "");
}
```

The `mount()` system call

```
int  mount(const  char  *source, const char *target,
           const char *filesystemtype, unsigned long mountflags,
              const void *data);
```

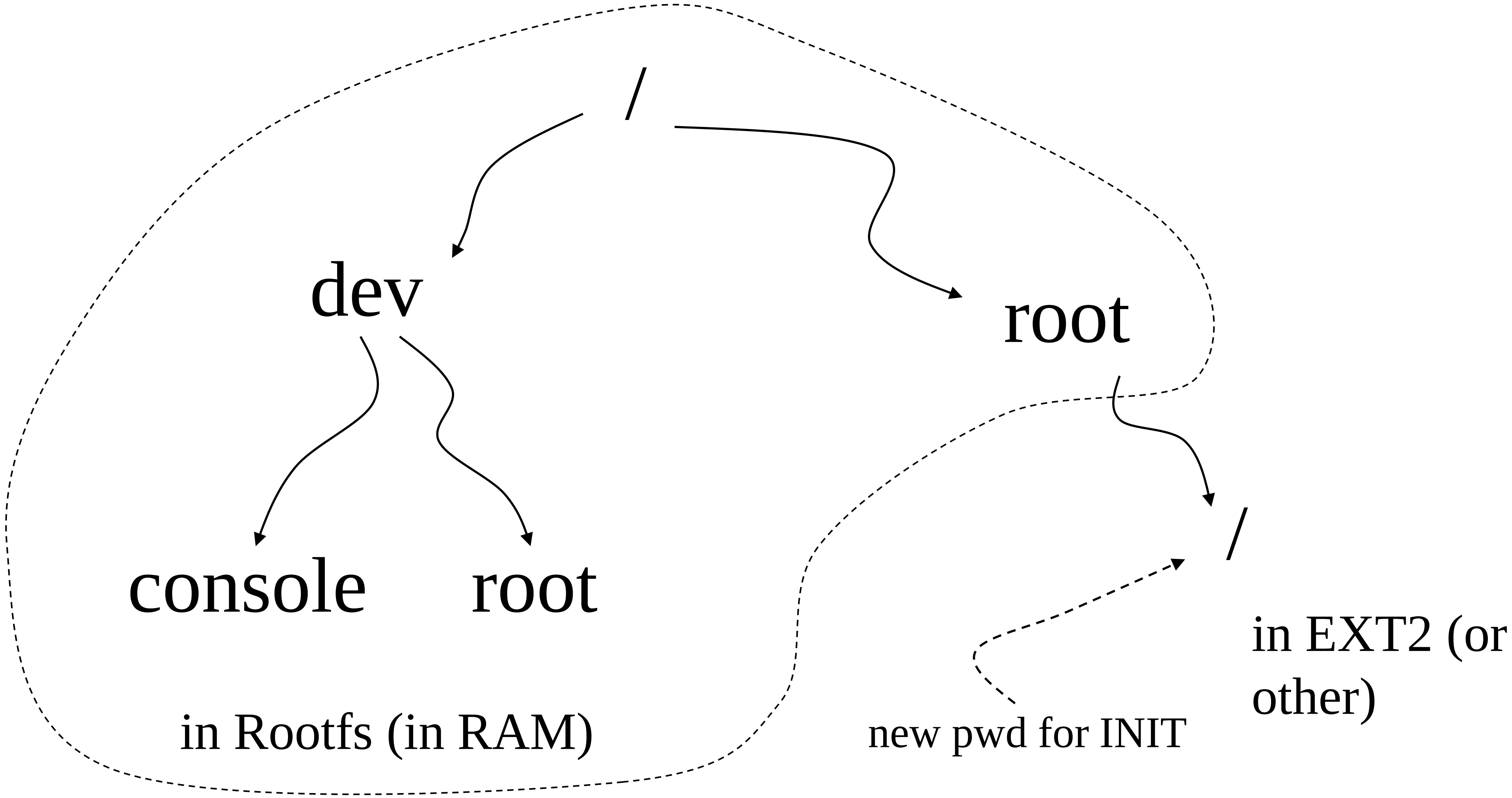
```
MS_NOEXEC     Do not allow programs to be executed from this file
              system.

MS_NOSUID     Do not honour set-UID and set-GID bits when execut-
              ing programs from this file system.

MS_RDONLY     Mount file system read-only.

MS_REMOUNT    Remount an existing mount.  This is allows  you  to
              change the mountflags and data of an existing mount
              without having to unmount and remount the file sys-
               tem.   source  and target should be the same values
               specified in the initial mount() call;  filesystem-
               type is ignored.

MS_SYNCHRONOUS Make  writes  on  this  file system synchronous
               (as though the O_SYNC flag to open(2) was specified
                for all file opens to this file system).
```

Mounting scheme

- The device to be mounted is used for accessing the driver (e.g. to open the device and to load the super-block)
- The superblock read function is identified via the device (file system type) to be mounted
- The super-block read-function will check whether the superblock is compliant with what expected for that device (i.e. file system type)
- In case of success, the 4 classical file system representation structures get allocated and linked in main memory
- **Note**: `sys_mount` relies on `do_kern_mount()`

The scheme

- This is the state at the end of the execution of `mount_root()`



Mount point

- **NOTE**: any directory selected as the target for the mount operation becomes a so called “mount point”
- `struct dentry` keeps the field `int d_mounted` to determine whether we are in presence of a mount point
- the function `path_lookup()` ignores the content of mount points (namely the name of the dentry) while performing pattern matching
- hence `sys_chroot(“.”)` (executed right after `prepare_namespace()`) brings INIT onto the root of the EXT2 file system (or any other root file system)
- the move takes place after repositioning EXT2 (or other) onto “/” of Rootfs

Description of an `open()` – kernel side

The steps

1. Get a free file descriptor (via `current->files->fd`)
2. Get the dentry via `filp_open()` (internally calls `file_operation` open)
3. Link the two things together

Description of a `close()` – kernel side

The steps

1. Release the dentry (by file descriptor) via `filp_close()` (internally calls `file_operation` close)
2. Release the file decriptor (via `current->files->fd`)

Description of a `read()`/`write()` – kernel side

The steps

1. Get reference to dentry via file descriptor
2. Get reference to `file_operations`
3. Call the associated interface in `file_operations`

`proc` file system

- It is an in-memory file system which provides information on
  - Active programs (processes)
  - The whole memory content
  - Kernel level settings (e.g. the currently mounted modules)
- Common files on `/proc` are
  - `cpuinfo` contains the information established by the kernel about the processor at boot time, e.g., the type of processor, including variant and features.
  - `kcore` contains the entire RAM contents as seen by the kernel.
  - `meminfo` contains information about the memory usage, how much of the available RAM and swap space are in use and how the kernel is using them.
  - `version` contains the kernel version information that lists the version number, when it was compiled and who compiled it.

- `net/` is a directory containing network information.
- `net/dev` contains a list of the network devices that are compiled into the kernel. For each device there are statistics on the number of packets that have been transmitted and received.
- `net/route` contains the routing table that is used for routing packets on the network.
- `net/snmp` contains statistics on the higher levels of the network protocol.
- `self/` contains information about the current process. The contents are the same as those in the per-process information described below.

- `pid/` contains information about process number *pid*. The kernel maintains a directory containing process information for each process.
- `pid/cmdline` contains the command that was used to start the process (using null characters to separate arguments).
- `pid/cwd` contains a link to the current working directory of the process.
- `pid/environ` contains a list of the environment variables that the process has available.
- *`pid`*`/exe` contains a link to the program that is running in the process.
- `pid/fd/` is a directory containing a link to each of the files that the process has open.
- `pid/mem` contains the memory contents of the process.
- `pid/stat` contains process status information.
- `pid/statm` contains process memory usage information

Registering the proc file system type

- Registration of the proc file system type occurs (if configured) in `start_kernel()` right before executing `rest_init()`
- It is configured via the macro `CONFIG_PROC_FS`, exploited as follows in `start_kernel()`

```
#ifdef CONFIG_PROC_FS
  proc_root_init();
#endif
```

- The function `proc_root_init()`, defined in `fs/proc/root.c`, is in charge of both registering `/proc` and creating the actual instance

proc features

- `stuct file_system_type` for the proc file system is initialized at compile time in `fs/proc/root.c` come segue

```
static DECLARE_FSTYPE(proc_fs_type,
   "proc", proc_read_super, FS_SINGLE);
```

- **NOTE:**
  - The flag `FS_SINGLE` is registered within the field `fs_flags` of the `proc_fs_type` variable
  - It indicates that this file system is managed as a single instance
  - Even though `/proc` is an in-RAM file system, it is completely different from Rootfs, in fact they have very different super-block read functions

Creation of the proc instance

- It occurs right after registering `proc` as a valid file system, and takes place in `proc_root_init()`
- Additional tasks by this function include creating some subdirs of proc such as
  - `net`
  - `sys`
  - `sys/fs`
- Creating a subdir in proc takes place via the kernel function `proc_mkdir()`

Core data structures for proc

```
struct proc_dir_entry {
       unsigned short low_ino;
       unsigned short namelen;
       const char *name;
       mode_t mode;
       nlink_t nlink;    uid_t uid;    gid_t gid;
       unsigned long size;
       struct inode_operations * proc_iops;
       struct file_operations * proc_fops;
       get_info_t *get_info;
       struct module *owner;
       struct proc_dir_entry *next, *parent, *subdir;
       void *data;
       read_proc_t *read_proc;
       write_proc_t *write_proc;
       atomic_t count;            /* use count */
       int deleted;        /* delete flag */
       kdev_t rdev;
    };
```

A few modifications occur while the kernel version changes

Properties of `struct proc_dir_entry`

- It fully describes any element of the proc file system in terms of
  - ➢name
  - ➢i-node operations (typically NULL)
  - ➢file operations (typically NULL)
  - ➢Specific read/write functions for the element
- We have specific functions to create proc entries, and to link the `proc_dir_entry` to the file system tree

Mounting proc

- The proc file system is not necessarily (depends on kernel version or config) mounted upon booting the kernel, it only gets instantiated if configured (see the macro `CONFIG_PROC_FS`)
- The proc file system gets mounted by INIT (if not before)
- This is done in relation to information provided by `/etc/fstab` or as a configured runtime task
- Typically, the root of the application level root-file-system keeps the directory `/proc` that is exploited as the mount point for the proc-file-system
- **NOTE:** given that the proc-file-system is single instance
    - ➢ No device needs to be specified for mounting proc, thus only the type of file system is required as parameter
    - ➢ Hence the `/etc/fstab` line for mounting proc does not specify any device

Handling proc (see `include/linux/proc_fs.h`)

```
struct proc_dir_entry *proc_mkdir(const char *name,
                 struct proc_dir_entry *parent);
```

Creates a directory called `name` within the directory pointed by `parent`. Returns the pointer to the new `struct proc_dir_entry`

```
static inline struct proc_dir_entry
*create_proc_read_entry(const char *name,
      mode_t mode, struct proc_dir_entry *base,
      read_proc_t *read_proc, void * data)
```

Creates a node called `name`, with type and permissions `mode`, linked to `base`, and where the reading function is set to `read_proc` end the data field to `data`. It returns the pointer to the new `struct proc_dir_entry`

```
struct proc_dir_entry *create_proc_entry(const char
*name, mode_t mode,  struct proc_dir_entry *parent)
```

Creates a node called `name`, with type and permissions `mode`, linked to `parent`. It returns the pointer to the new `struct proc_dir_entry`

```
static inline struct proc_dir_entry
*proc_create(const char *name, umode_t
mode, struct proc_dir_entry *parent, const
struct file_operations *proc_fops)
```

`name`: The name of the proc entry
`mode`: The access mode for proc entry
`parent`: The name of the parent directory under /proc
`proc_fops`: The structure in which the file operations for the proc entry will be created

Read/Write operations

- Read/write operations for proc have the same interface as for any file system handled by VFS

```
ssize_t (*read) (struct file *, char *,
                           size_t, loff_t *);

ssize_t (*write) (struct file *, const char *,
                     size_t, loff_t *);
```

- If not NULL, then actual read/write operations are those registered by the fields `read_proc_t *read_proc` and `write_proc_t *write_proc`

```
typedef int (read_proc_t)(char *page, char **start,
   off_t off, int count, int *eof, void *data);

typedef int (write_proc_t)(struct file *file, const
   char *buffer, unsigned long count, void *data);
```

An example with `read_proc_t`

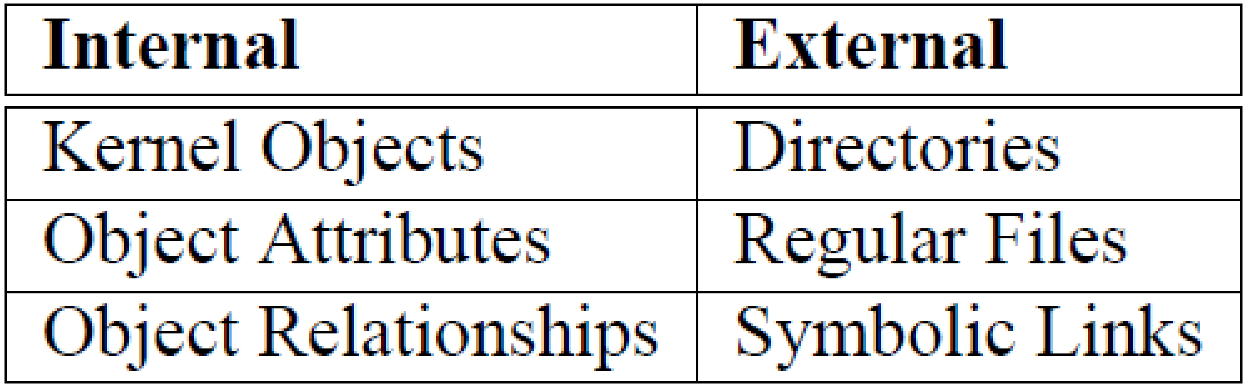
`char* page`	A pointer to a one-page buffer. (A page is `PAGE_SIZE` bytes big)
`char** start`	A pass-by-reference `char *` from the caller. It is used to tell the caller where is the data put by this procedure. (If you're curious, you can point the caller's pointer at your own text buffer if you don't want to use the page supplied by the kernel in `page`.)
`off_t off`	An offset into the buffer where the reader wants to begin reading
`int count`	The number of bytes after `off` the reader wants.
`int* eof`	A pointer to the caller's eof flag. Set it to 1 if the current read hits EOF.
`void* data`	Extra info you won't need
`return value`	Number of bytes written into `page`

We assume that the content of the proc-entry is within the buffer `pContent` and that it has size `N` bytes

```
int MyReadProc(char *page, char **start, off_t off, int
count, int *eof, void *data)
{
    int n;
    if (off >= N) {
        *eof = 1;
        return 0;
    }
    n = N-off;
    *eof = n>count ? 0 : 1;
    if (n>count)
        n=count;
    memcpy(page, pContent+off, n);
    *start = page;
    return n;
}
```

The sys file system (available since kernel 2.6)

- Similar in spirit to /proc
- It is an alternative way to make the kernel export information (or set it) via common I/O operations
- Very simple API
- More clear-cut structuring
- sysfs is compiled into the kernel by default depending on the configuration option CONFIG_SYSFS (visible only if CONFIG_EMBEDDED is set)



Internal	External
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

Baseline architectural concepts: kernel objects

- The `/sys` file system is based on data structures that play a more ample role within the Linux kernel
- This is the kernel object data structure architecture
- What is a kernel object
  - Something that allows to identity individual things
  - Something that allows to identify groups of things
  - Something that allows to identify the typology of things
  - Something that allows to associate the same typology to many
  - Something that allows to identify hierarchies

The kobject structure

```
Struct kobject{
     const char             *name;
     struct list_head   entry;
     struct kobject         *parent;
     struct kset            *kset;
     struct kobj_type       *ktype;
     struct kernfs_node *sd;
               /*sysfs directory entry*/
     struct kref            kref;
}
```

$\uparrow$

Reference counting

The kobj_type structure

We can have multiple attributes

```
struct kobj_type{
      void (*release)(structkobject*);
      struct sysfs_ops *sysfs_ops;
      struct attribute **default_attrs;
}
```

Called when reference counting reaches the vale zero

Actual operations to be executed on the object

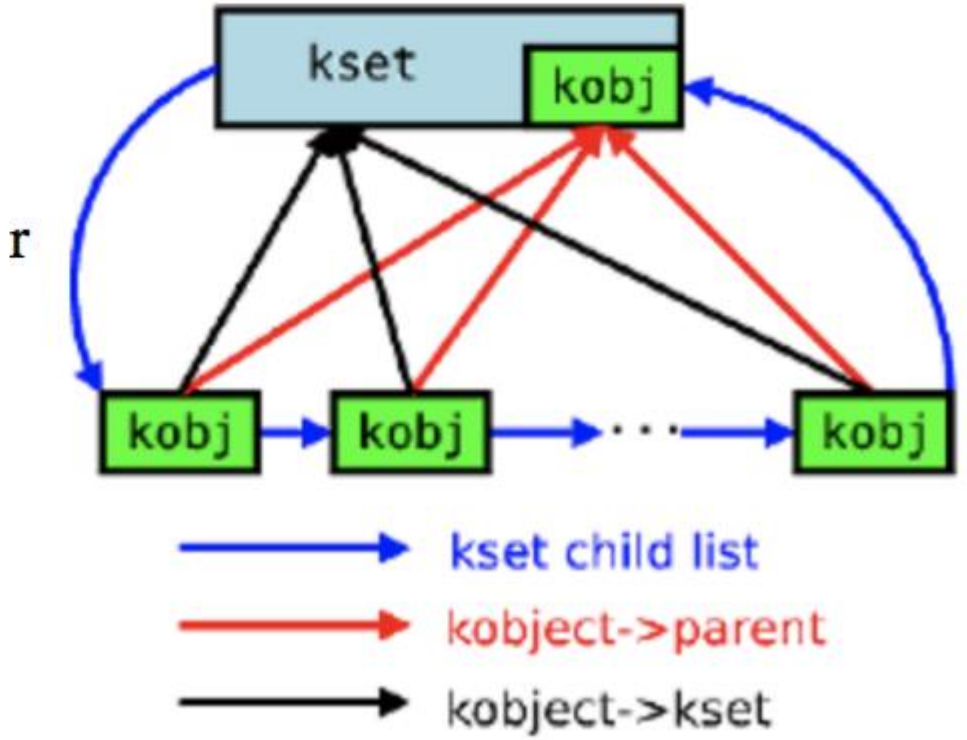
The specification of the read/write operations occurs via the sysfs_ops couple of functions

```
struct sysfs_ops {
        /* method invoked on read of a sysfs file */
        ssize_t (*show) (struct kobject *kobj,
                         struct attribute *attr,
                         char *buffer);

        /* method invoked on write of a sysfs file */
        ssize_t (*store) (struct kobject *kobj,
                          struct attribute *attr,
                          const char *buffer,
                          size_t size);
}
```

What can we do with kernel objects

- We can represent data that can be used by software to keep track of the current state of both logical and physical entities
- Examples are related to the representation of
  - ✓ The USB bus subsystem
  - ✓ The char devices subsystem
  - ✓ The block devices subsystem
- A kernel object may belong to only one subsystem!
- A subsystem must contain only identical kernel object elements!
- In Linux we use `struct kset` to group together all the kernel objects we want to have within the same subsystem

A representation of the linkage



Although it is not mandatory, we should keep all these kernel objects linked to the same type specification

File system linkage

- A `kset` element is associated with an I/O element of the `/sys` file system
- On the other hand, a kernel object can be either associated or not to an element of the `/sys` file system
  - ✓ it is associated if it is in `kset`
  - ✓ it can be out of the `/sys` file system if it is not inside a `kset`
- This also provides the importance of the kernel object reference counter

Baseline API

```
int kobject_add(struct kobject *kobj)
void kobject_del(struct kobject *kobj)
```

Add/remove from a pointed to `kset`

Baseline management

There is also

kobject_register, which is a combination of ***kobject_init*** and ***kobject_add***

kobject_unregister, which is a combination of ***kobject_del*** and ***kobject_put***

kset API

```
void kset_init(struct kset *kset)

int kset_register(struct kset *kset)

void kset_unregister(struct kset *kset)

struct kset *kset_get(struct kset *kset)

void kset_put(struct kset *kset)

kobject_set_name(my_set>kobj,“thename")
```

Event to user space

- It is used to notify that something has changed in relation to things that are handled by kernel objects
- The architecture is based on a function pointer that is called **`kobject_uevent`**
- This function pointer is recorded into the `kset` data structure
- The identified function is typically used to let the kernel start some user space application when something occurs at the kernel side
- The classical example is when inserting an USB drive, in this case a user space program is started to let the used know about the insertion (and to ask what to do)

sysfs core API for kernel objects

```
int sysfs_create_dir(struct kobject * k);

void sysfs_remove_dir(struct kobject * k);

int sysfs_rename_dir(struct kobject *, const char *new_name);
```

Main fields: `parent` – `name`

- it is possible to call `sysfs_create_dir` without `k->parent` set
- it will create a directory at the very top level of the sysfs file system
- this can be useful for writing or porting a new top-level subsystem using the `kobject/sysfs` model

sysfs core API for object attributes

```
int sysfs_create_file(struct kobject *, const struct attribute *);

void sysfs_remove_file(struct kobject *, const struct attribute *);

int sysfs_update_file(struct kobject *, const struct attribute *);
```

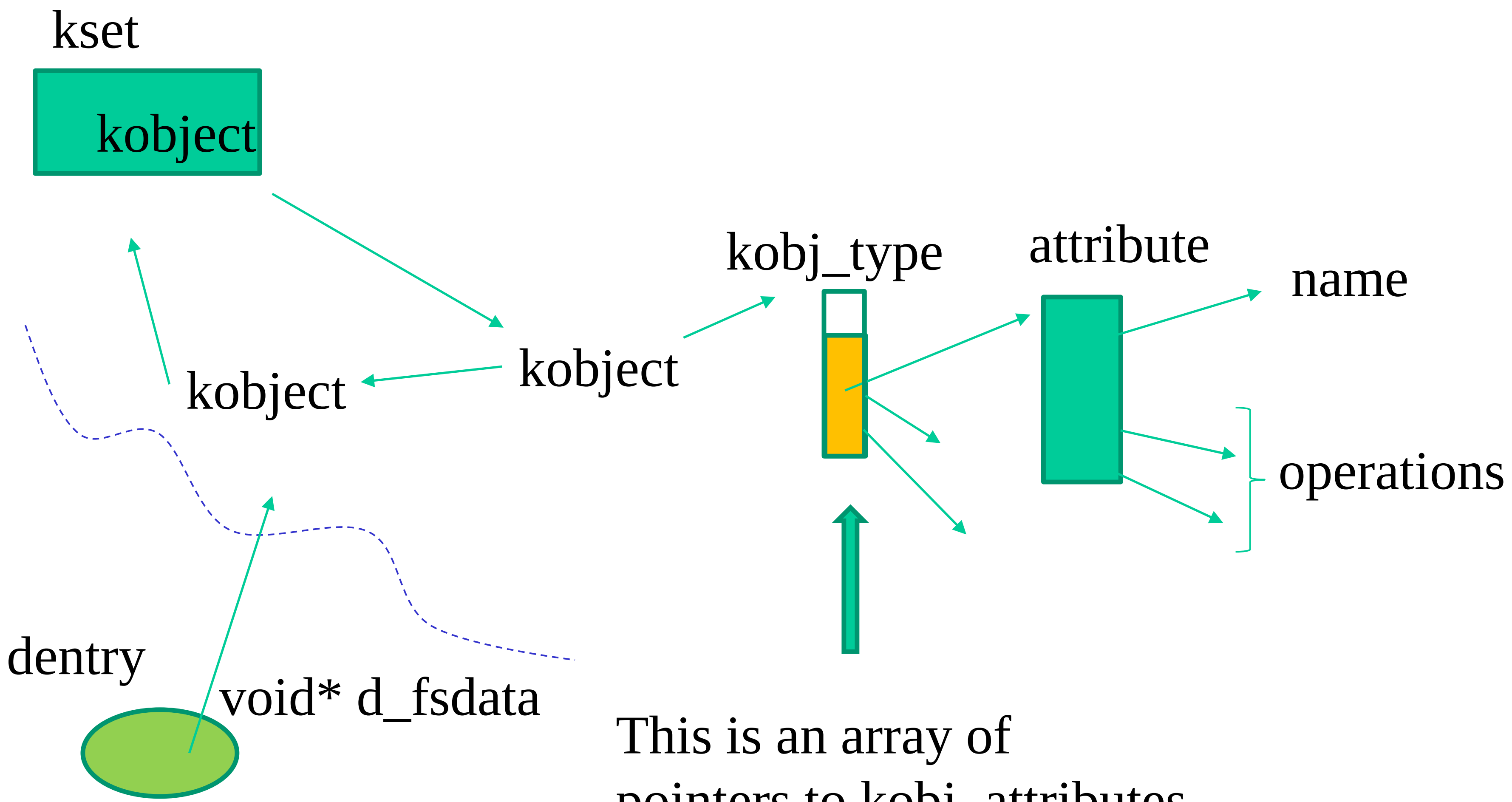
```
struct attribute {
    char                *name;
    struct module       *owner;
    mode_t              mode;
};
```

The owner field may be set by the caller to point to the module in which the attribute code exists

Actual object attributes

```
struct kobj_attribute {
      struct attribute attr;
      ssize_t (*show)(struct kobject *kobj,
            struct kobj_attribute *attr, char *buf);
      ssize_t (*store)(struct kobject *kobj,
                  struct kobj_attribute *attr,
                  const char *buf, size_t count);
}
```

Overall architecture



This is an array of pointers to kobj_attributes

Kernel API for creating devices in /sys

- `/sys/class` is a device file that internally hosts the reference to other device files

- To create a device file in this “directory” one can resort to:

```
static struct class* class_create(struct
 moudule* owner, char* class_name)

static struct class* device_create(static struct
 class* the_class, … kdev_t i_rdev, … char*
 name)
```

- There are similar API functions for destroying the device and the class