

Advanced Operating Systems (and System Security)

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 (namely of its I/O objects)
- 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

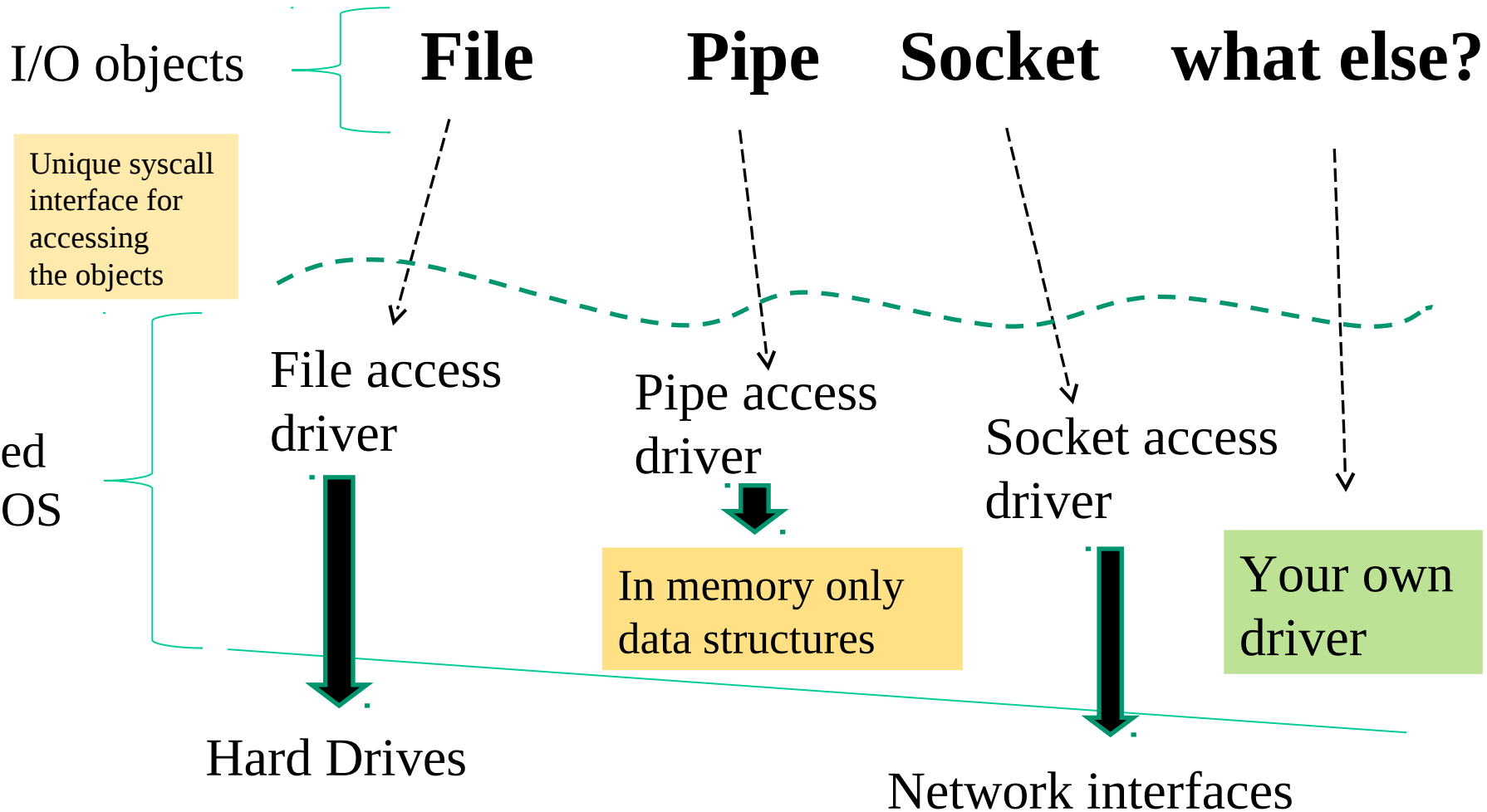
Connections

- 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 → the virtual file system (VFS)
- 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 used 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

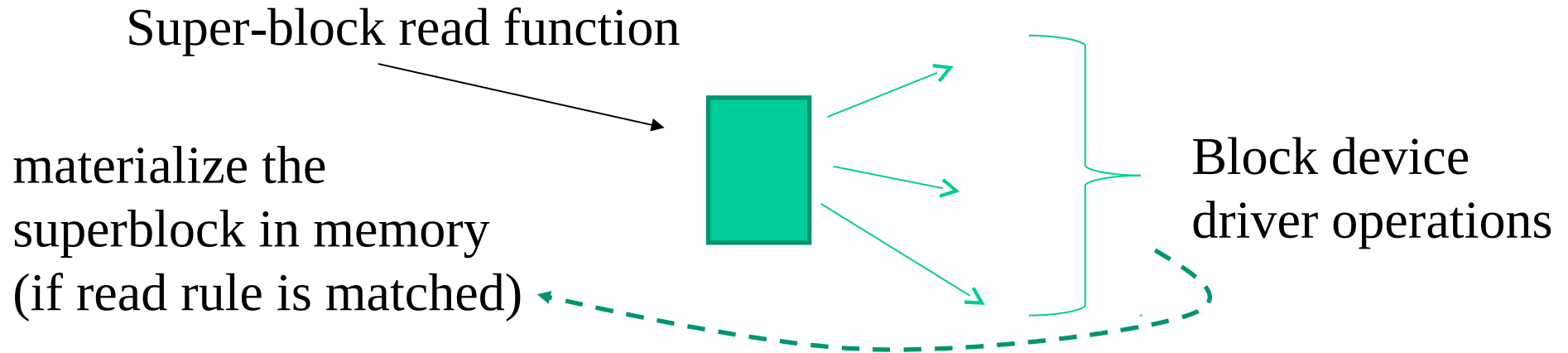
- Files are backed by data on a hard drive
- What **software modules do we need** for managing files on that hard drive in a well shaped OS-kernel??
 1. A function to read the device superblock for determining what files exist and where their data are
 2. A function to read device blocks for bringing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



Actual architecture (i)

- The super-block read function can exploit kernel level API in order to setup the VFS portion of the superblock, like:
 - `mount_bdev()`, which mounts a file system stored on a block device
 - `mount_single()`, which mounts a file system that shares an instance between all mount operations
 - `mount_nodev()`, which mounts a file system that is not on a physical device
 - `mount_pseudo()`, a helper function for pseudo-file systems (sockfs, pipefs, generally file systems that can not be mounted)

Actual architecture (ii)

- All the previously listed functions will take a call-back function as a parameter, which will be called in order to finalize the super-block materialization
- This will be done in file-system specific manner
- This function typically just **fills** the super-block content

Super-block read function

Fill callback-function



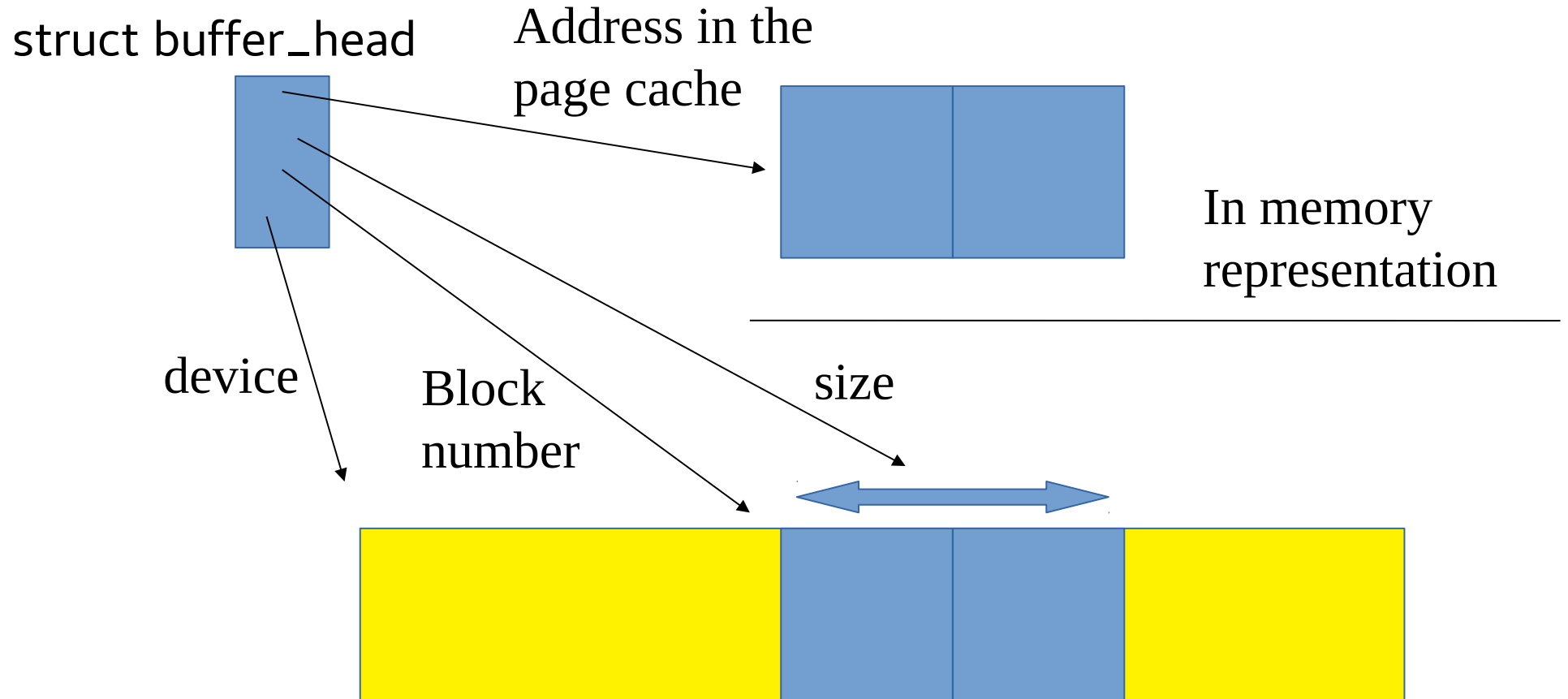
The “magic number”

- In the end a block device is anyhow a sequence of bytes
- We can read this sequence and check whether it contains (e.g. in the super block) some identifying code we are expecting
- If this is not true, then we can abort the instantiation of the superblock in memory
- For Posix the command “`file [-s] /dev/{device-name}`” allows to extract the magic number (the code) and reports the information on the actual file system type kept by a device

Buffer/page cache

- It is simply a memory area where we keep blocks of devices for managing operations (read/write)
- Linux offers the struct `buffer_head` data structure to manage these blocks, which is made by the following main data
 - `*b_data`, pointer to a memory area where the data was read from or where the data must be written to
 - `b_size`, buffer size
 - `*b_bdev`, the block device
 - `b_blocknr`, the number of the block on the device that has been loaded or needs to be saved on the device

A scheme



Getting/putting device blocks

`__bread()` → reads a block with the given number and given size in a `buffer_head` structure; returns a pointer to the `buffer_head` structure (NULL on error)

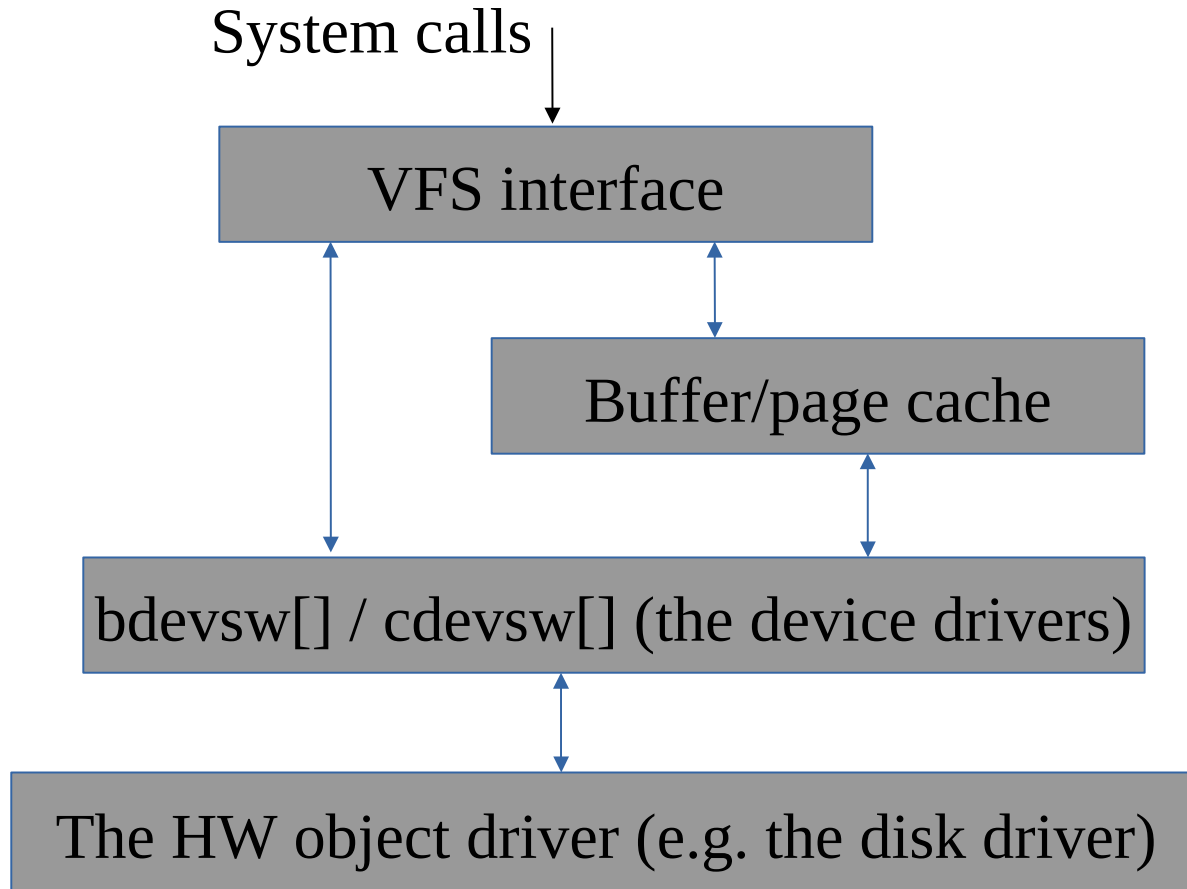
`sb_bread()` → the size of the read block is taken from the superblock;

`mark_buffer_dirty()` → marks the buffer as dirty (sets the `BH_Dirty` bit); the buffer will be written to the disk at a later time (from time to time the `bdflush` kernel thread wakes up and writes the buffers to disk);

`brlease()` → frees up the memory used by the buffer, after it has previously written the buffer on disk if needed;

`map_bh()` → associates the buffer-head with the corresponding sector

The overall layering

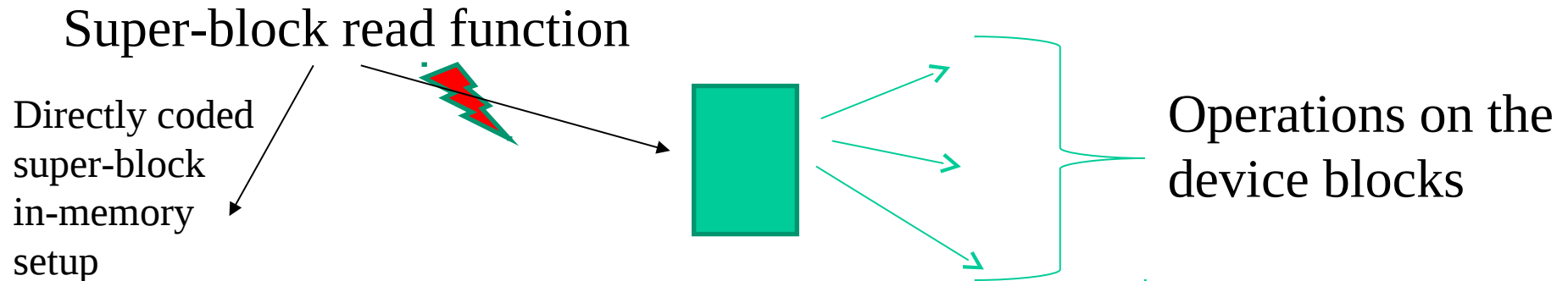


Regular files vs devices

- Any regular file can be seen as a block device hosting a file system
- To correctly associate this role to the file we will need to mount the corresponding file system using a specific block-device driver
- This is the -o loop driver
- This enables passing through the VFS architecture multiple times (in terms of actual actions excuted when system calls are called)
- We can therefore create a stack of file system devices

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 **do not have an on-device** 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



RAM file system fill example – from kernel 5

```
static int ramfs_fill_super(struct super_block *sb, struct fs_context *fc){  
  
    struct ramfs_fs_info *fsi = sb->s_fs_info;  
    struct inode *inode;  
  
    sb->s_maxbytes          = MAX_LFS_FILESIZE;  
    sb->s_blocksize         = PAGE_SIZE;  
    sb->s_blocksize_bits    = PAGE_SHIFT;  
    sb->s_magic              = RAMFS_MAGIC;  
    sb->s_op                = &ramfs_ops;  
    sb->s_time_gran         = 1;  
  
    inode = ramfs_get_inode(sb, NULL, S_IFDIR | fsi->mount_opts.mode, 0);  
  
    sb->s_root = d_make_root(inode);  
    if (!sb->s_root)  
        return -ENOMEM;  
  
    return 0;  
}
```

Here we are simply allocating other two data structures in memory, namely the inode and the dentry

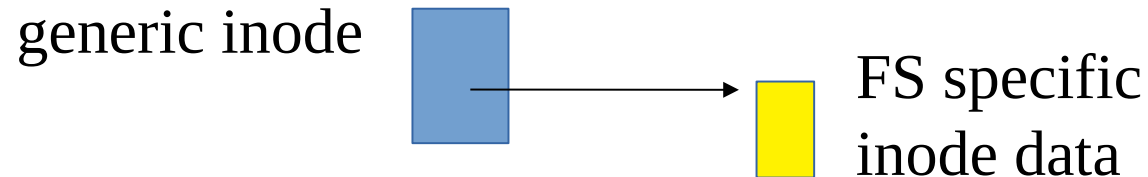
Baseline API for i-nodes and dentry

`struct inode *new_inode(struct super_block *sb)` → we simply allocate a generic i-node data structure making `id` refer to a generic super-block data structure

`struct dentry *d_make_root(strct inode *root_inode)` → we simply create a generic dentry data structure that will figure out as the root one, and we link it to the root-inode

The root-inode can be populated in a FS specific manner (e.g. upon file system mount) reading an actual inode from a device

It is typical that these data structures will keep generic fields used by the VFS plus some filed (e.g. a pointer) usable for linking FS specific data



Baseline structure of a superblock-fill function

```
int <FS_name>_fill_super(struct super_block *sb, ...){
```

```
.....
```

```
bh = sb_bread(); //read the FS specific superblock from device
```

```
... // populate the FS-specific structure in memory
```

```
brelse(bh); //release the page-cache kept data (not mandatory)
```

```
root_inode = <FS_name>_iget(sb,0) //get the root inode (generic + FS specific data)
```

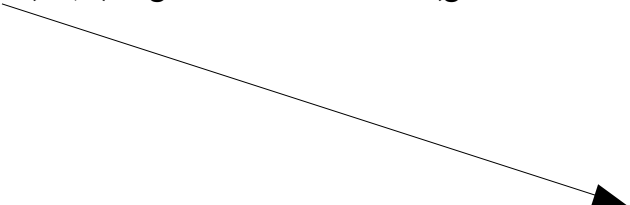
```
...
```

```
d_make_root(root_inode);
```

```
...
```

```
}
```

index 0 is typical of the
root-inode of any file system



```
int <FS_name>_iget(struct super_block *sb, int inode){
```

```
.....
```

```
inode = ... // allocate a generic inode
```

```
...
```

```
bh = sb_bread(); //read the FS-specific inode with given index from device
```

```
inode → <field> = bh → <something>;
```

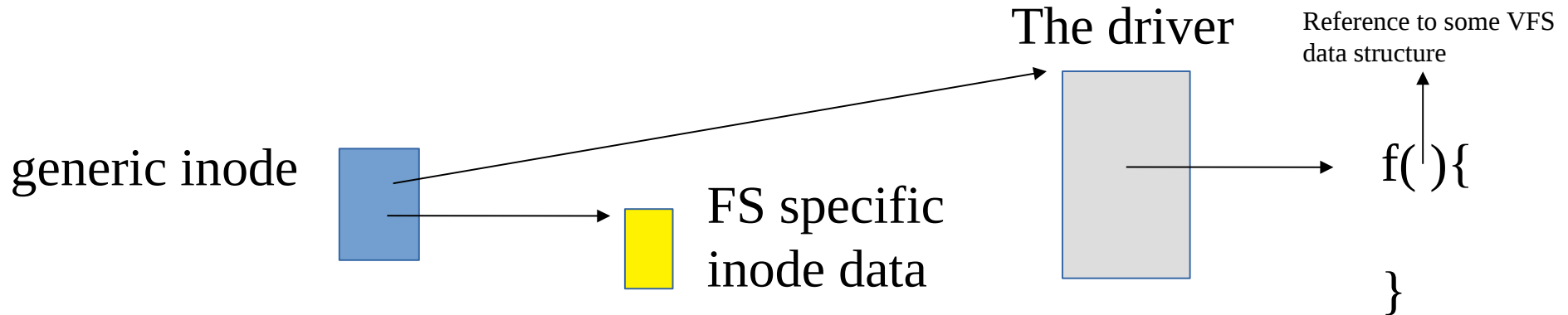
```
brelse(bh); //release the page-cache kept data (not mandatory)
```

```
...
```

```
}
```

Data structures vs drivers

- A driver for operations on a data structure in the VFS is a table of function pointers
- When one of the operations is invoked we can pass as parameter the address of the generic data structure
- From this address the driver can access (more or less directly) the FS specific data
- As mentioned before a data structure in the VFS keeps a reference to the actual driver for its operations



The VFS startup in Linux

- 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)

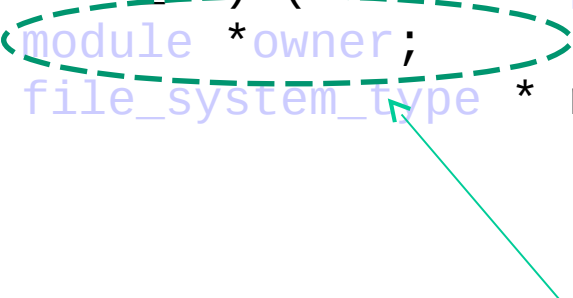
```
struct file_system_type {  
    const char *name;  
    int fs_flags;  
    .....  
    struct super_block *(*read_super) (struct super_block *, void *, int);  
    struct module *owner;  
    struct file_system_type * next;  
    struct list_head fs_supers;  
    .....  
};
```

Moved to the mount field
in newer kernel versions



... 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;  
    ...  
    ...  
}
```

A dashed green oval encircles the 'struct module *owner;' and 'struct file_system_type * next;' lines. A green arrow points from the text 'Beware this!!' to the 'struct file_system_type * next;' line.

Beware this!!

Rootfs and basic fs-type API (i)

- 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 link it to a proper list
- The linkage one is

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

Rootfs and basic fs-type API (ii)

- 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);  
    ...  
}
```

let's check with the details 

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;
}
```

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

User level checks on the managed file systems

- The file system currently manageable by the kernel can be listed by accessing the `/proc/filesystems` file
- The `nodev` field in the output tells that a specific file system is handled as a in-memory one, e.g.:

```
nodev    sysfs
nodev    rootfs
nodev    ramfs
.....
nodev    proc
.....
ext3
ext4
```

- Among the `nodev` file systems we typically find `sys` and `proc`

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 exists for any file/directory of the specific file system

More details on the data structures

`struct vfsmount`  Tells, e.g., what is the parent FS

`struct super_block`  Keeps basic FS metadata

`struct inode`  Keeps per I/O object metadata

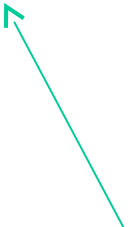
`struct dentry`  Tells what is a name for an I/O object along the FS hierarchy

The structure vfstmount (still in place in kernel 3.xx)

```
struct vfstmount {
    struct list_head mnt_hash;
    struct vfstmount *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

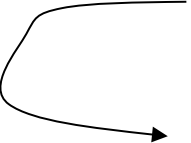
- Access to any field of a structure is based on compiler rules when relying on classical ‘.’ or ‘->’ operators
- Machine code is generated in such a way to correctly displace into the proper field
- `__randomize_layout` introduces a reshuffle of the fields, with the inclusion of padding
- This is done based on pseudo random values selected at compile time
- Hence an attacker who 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 – Kernel 5 example

```
struct super_block {
    struct list_head    s_list;           /* Keep this first */
    dev_t               s_dev;           /* search index; _not_ kdev_t */
    ...
    unsigned long       s_blocksize;
    loff_t              s_maxbytes;      /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations *s_op;
    ...
    unsigned long       s_magic;
    struct dentry        *s_root;
    ...
    struct list_head    s_mounts; /* list of mounts */
    struct block_device *s_bdev;
    ...
    void                *s_fs_info;      /* Filesystem private info */
    ...
    const struct dentry_operations *s_d_op; /* default d_op for dentries */
    ...
    struct user_namespace *s_user_ns;
    ...
} __randomize_layout;
```

The structure dentry – Kernel 5 example

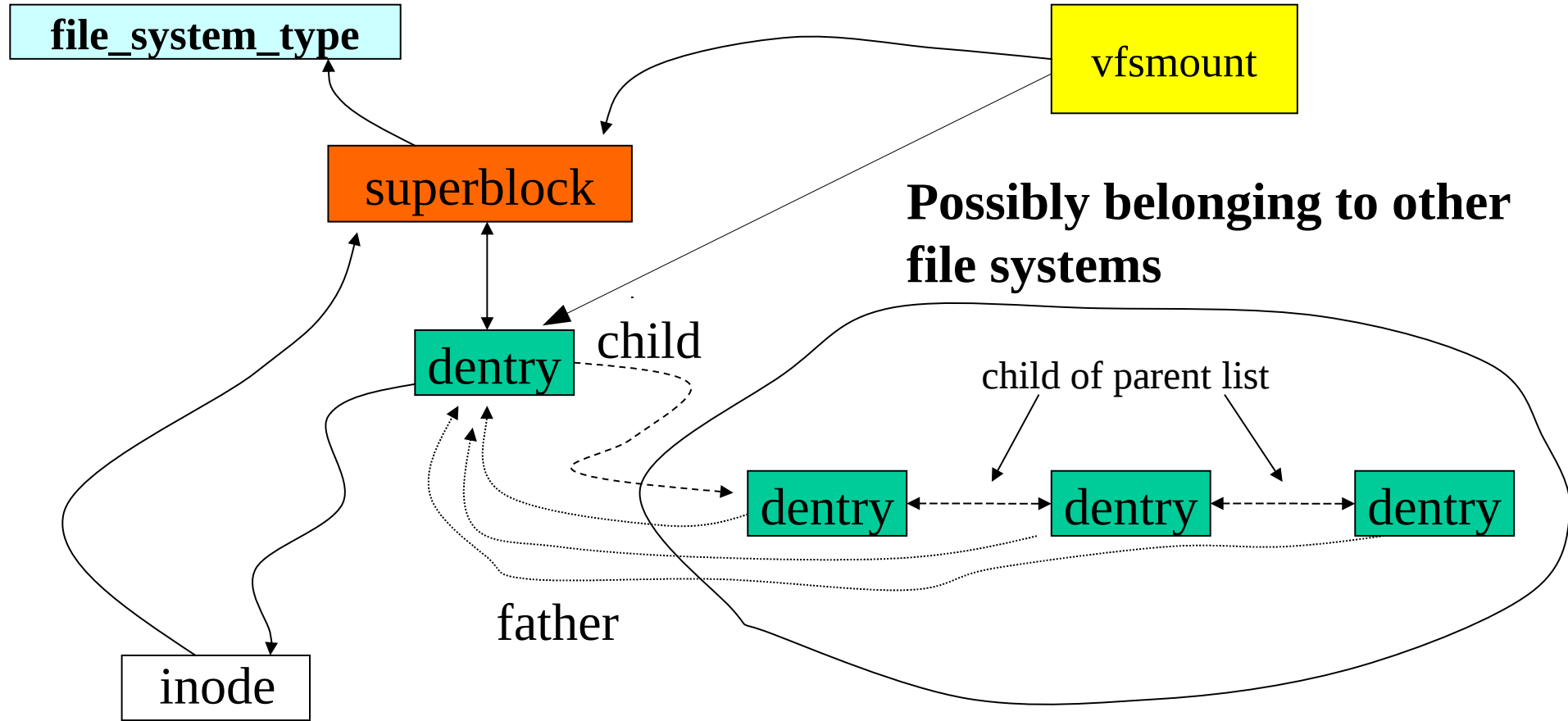
```
struct dentry {  
    ...  
    struct dentry *d_parent;    /* parent directory */  
    struct qstr d_name;  
    struct inode *d_inode; /* Where the name belongs to */  
    unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */  
    ...  
    const struct dentry_operations *d_op;  
    struct super_block *d_sb; /* The root of the dentry tree */  
    ...  
    void *d_fsdata;                /* fs-specific data */  
    ...  
    struct list_head d_child; /* child of parent list */  
    struct list_head d_subdirs; /* our children */  
    ...  
} __randomize_layout;
```



The structure inode – Kernel 5 example

```
struct inode {
    umode_t          i_mode;
    unsigned short   i_opflags;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int      i_flags;
    ...
    const struct inode_operations    *i_op;
    struct super_block *i_sb;
    ...
    loff_t              i_size;
    ...
    spinlock_t          i_lock;  /* i_blocks, i_bytes, maybe i_size */
    ...
    union {
        const struct file_operations    *i_fop;  /* former ->i_op->default_file_ops */
        void (*free_inode)(struct inode *);
    };
    ...
    void                *i_private; /* fs or device private pointer */
} __randomize_layout;
```

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()`

Mount tree setup – kernel 3

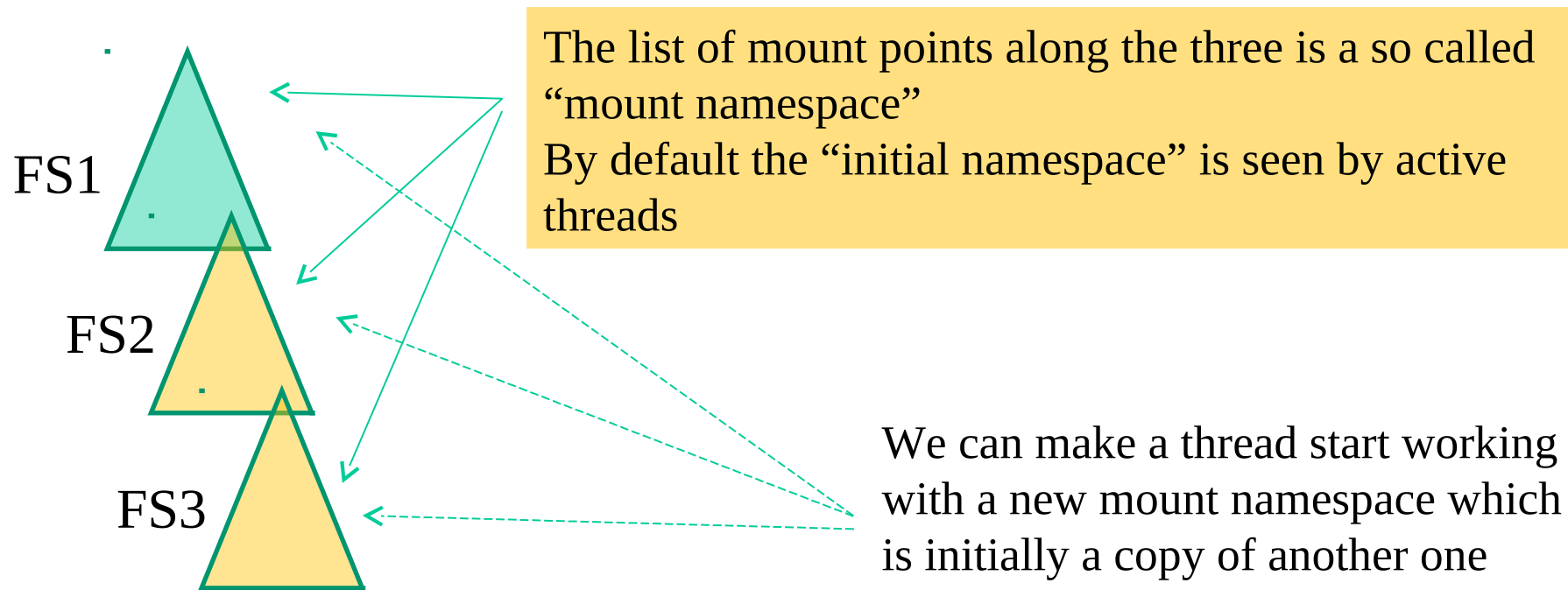
```
static void __init init_mount_tree(void){
    struct vfsmount *mnt;
    struct namespace *namespace;
    struct task_struct *p;

    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



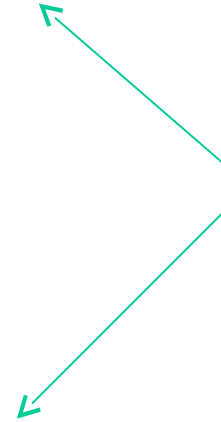
Moving to another mount namespace makes `mount/unmount` operations only acting on the current namespace (except if the mount operation is tagged with `SHARED`)

Actual system calls for mount namespaces

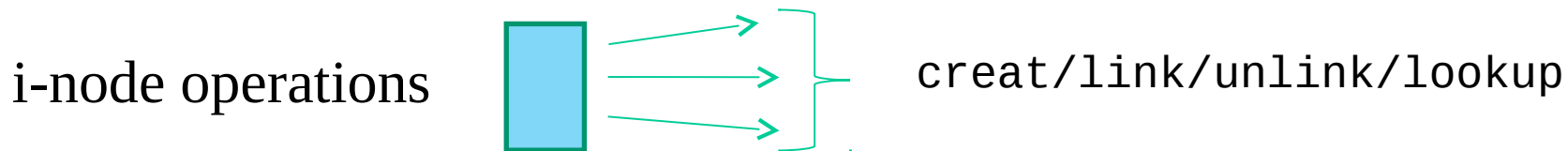
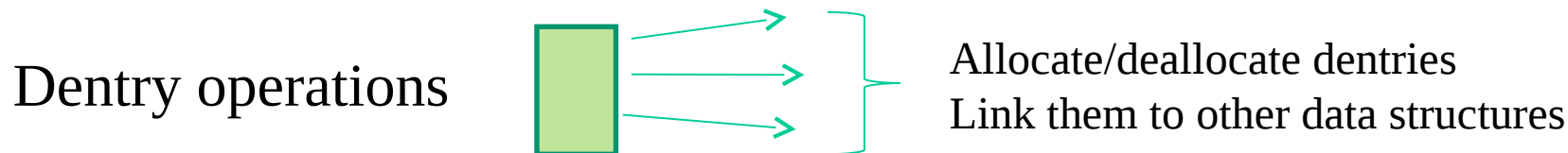
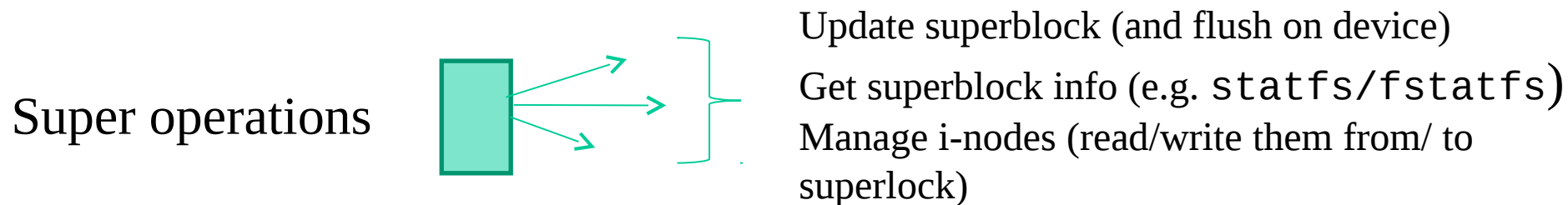
`clone(... int flags ...)`

`CLONE_NEWNS`

`unshare(int flags)`



An overall view



TCB vs VFS

- 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` was defined as follows in kernel 2.4

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

3.xx/4.7 kernel style

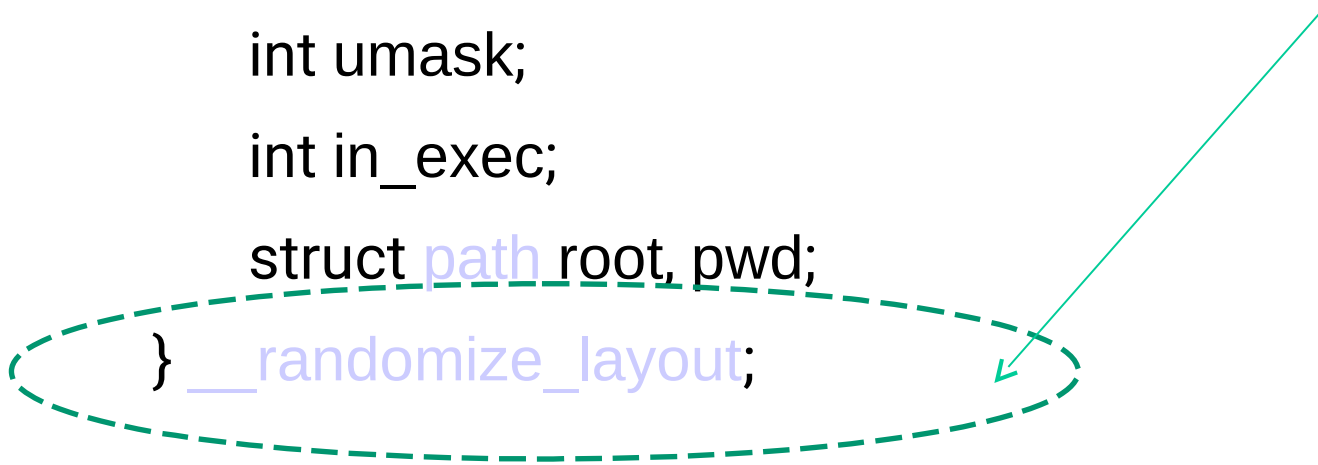
See [include/linux/fs_struct.h](#)

```
8 struct fs_struct {  
9     int users;  
10    spinlock_t lock;  
11    seqcount_t seq;  
12    int umask;  
13    int in_exec;  
14    struct path root, pwd;  
15 };
```

... and then 4.8 or later style

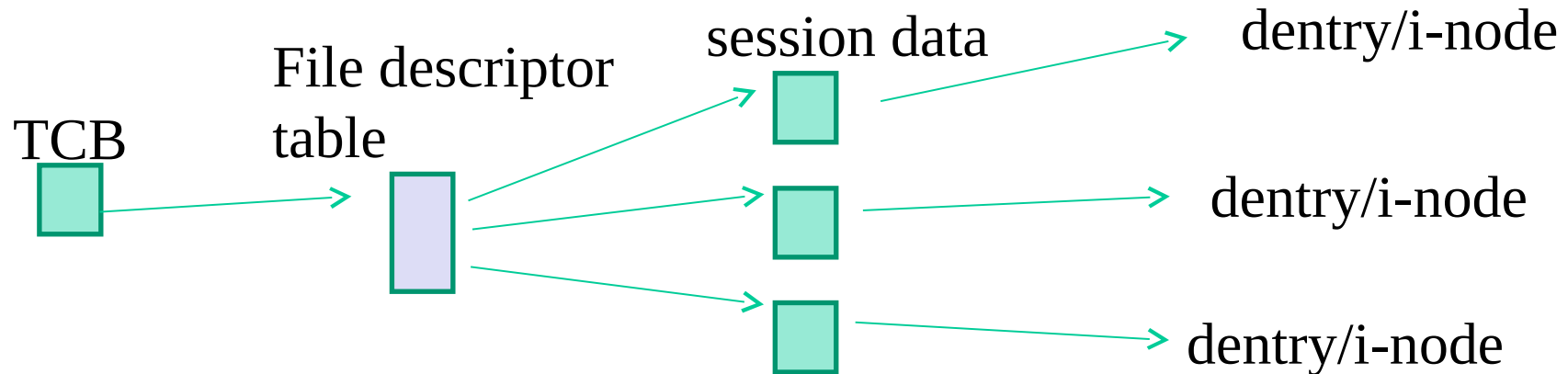
```
struct fs_struct {  
    int users;  
    spinlock_t lock;  
    seqcount_t seq;  
    int umask;  
    int in_exec;  
    struct path root, pwd;  
} __randomize_layout;
```

Towards more security



File descriptor table

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



Classical 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;  ← bitmap for close on exec flags  
    fd_set *open_fds;       ← bitmap identifying open fds  
    fd_set close_on_exec_init;  
    fd_set open_fds_init;  
    struct file * fd_array[NR_OPEN_DEFAULT];  
};
```

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)

```
775 struct file {
776     union {
777         struct llist_node    fu_llist;
778         struct rcu_head      fu_rcuhead;
779     } f_u;
780     struct path              f_path;
781 #define f_dentry             f_path.dentry
782     struct inode             *f_inode;    /* cached value */
783     const struct file_operations *f_op;
784
785     /*
786      * Protects f_ep_links, f_flags.
787      * Must not be taken from IRQ context.
788      */
789     spinlock_t               f_lock;
790     atomic_long_t            f_count;
791     unsigned int              f_flags;
792     fmode_t                   f_mode;
793     struct mutex              f_pos_lock;
794     loff_t                    f_pos;
795     struct fown_struct        f_owner;
796     const struct cred         *f_cred;
797     struct file_ra_state      f_ra;
798
799     .....
800     ..... __randomize_layout;;
```

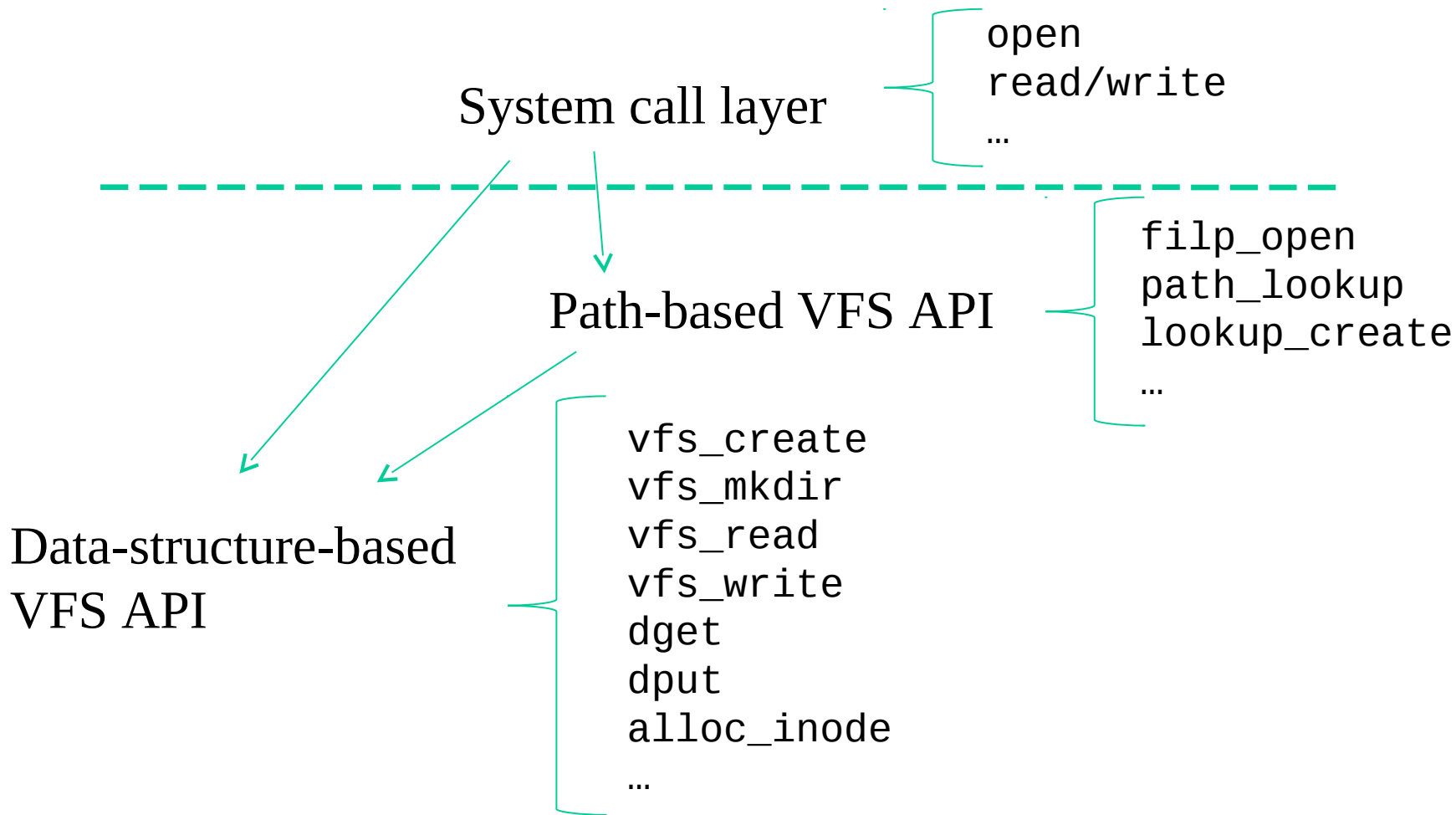
Now we have randomized layout and a few fields are moved to other pointed tables

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



Path-based API examples

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

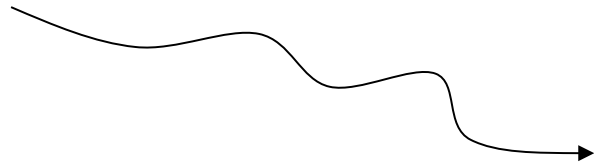
returns the address of the **struct file** associated with the opened file

open() system-call



kernel-level

filp_open()



i-node operation lookup()

In the end we pass through 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)
```

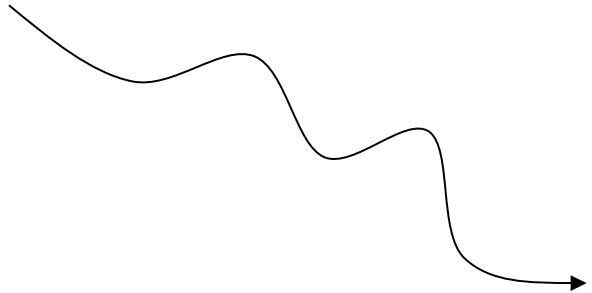
Acquires a dentry (by incrementing the reference counter)

```
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(.....)

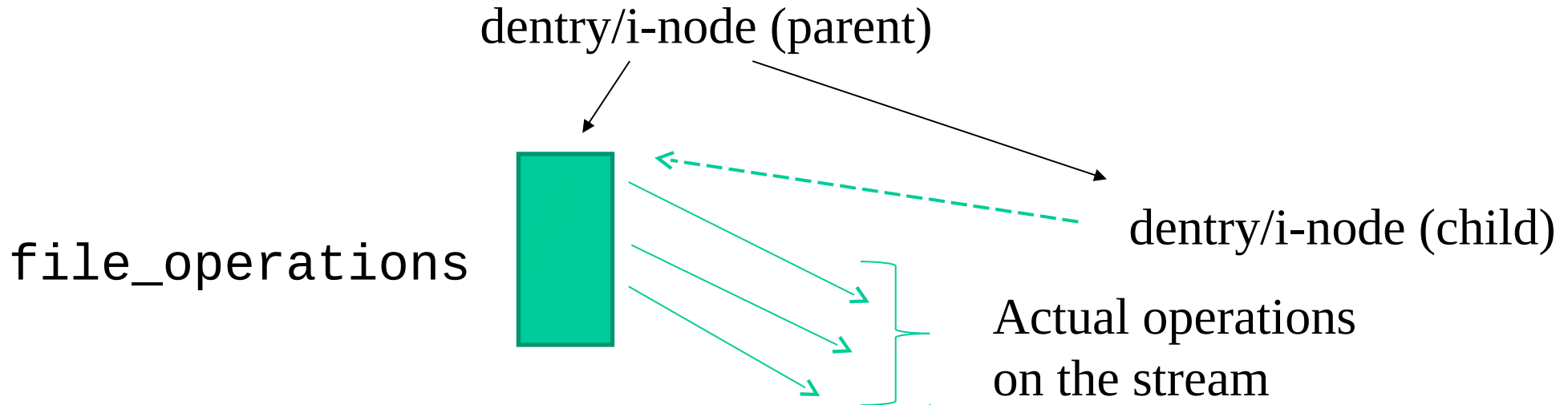
In the end we traverse dentry/i-node structures to retrieve the file operations table associated with that dentry

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, then we need to:
 - ✓ Identify the char-dev driver for operating on the file (this will depend on where we registered the driver for that device 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` keeps an information indicating the type of the i-node, e.g.:
 - directory
 - file
 - char device
 - block device
 - (named) pipe
- `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)

Char devices table

```
struct device_struct {  
    const char * name;  
    struct file_operations * fops;  
};  
  
static struct device_struct chrdevs[MAX_CHRDEV];
```

Device name

Device operations

- 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)

```
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 *);
    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);
};
```

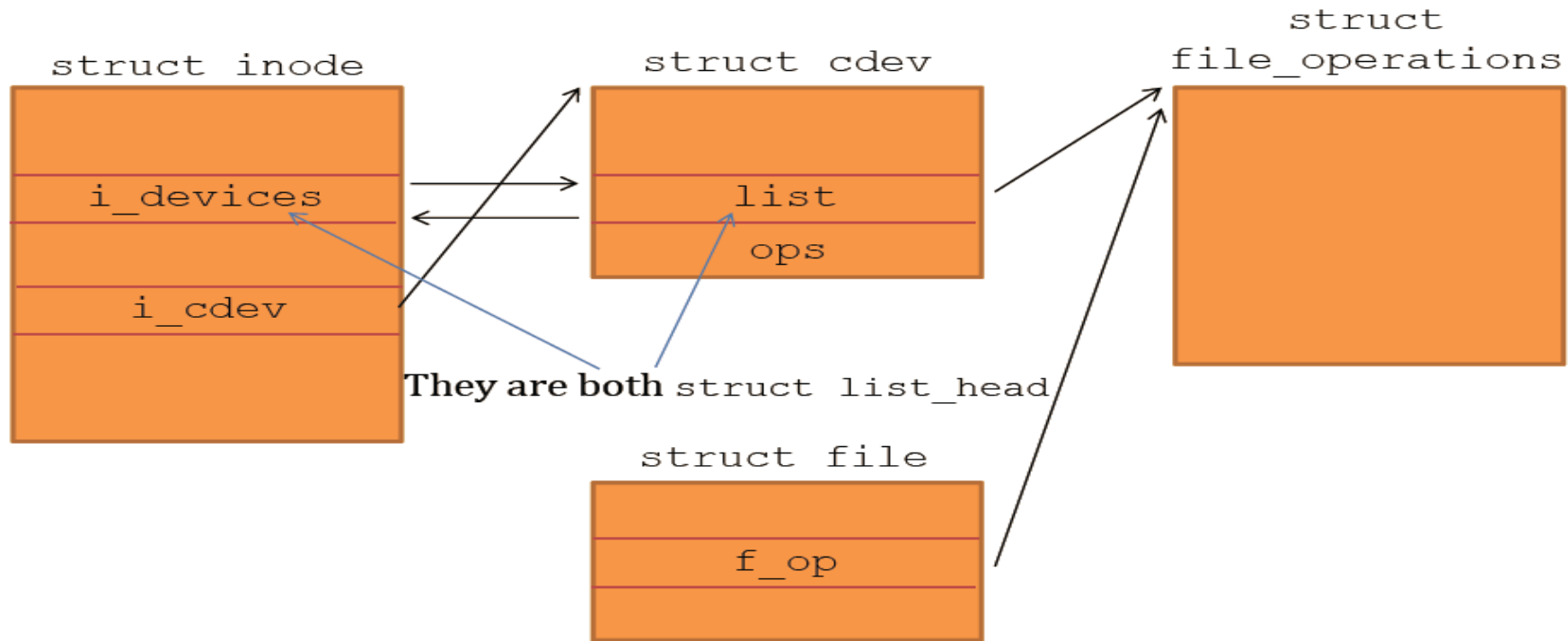
Kernel 3 or later - augmenting flexibility and structuring

```
#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

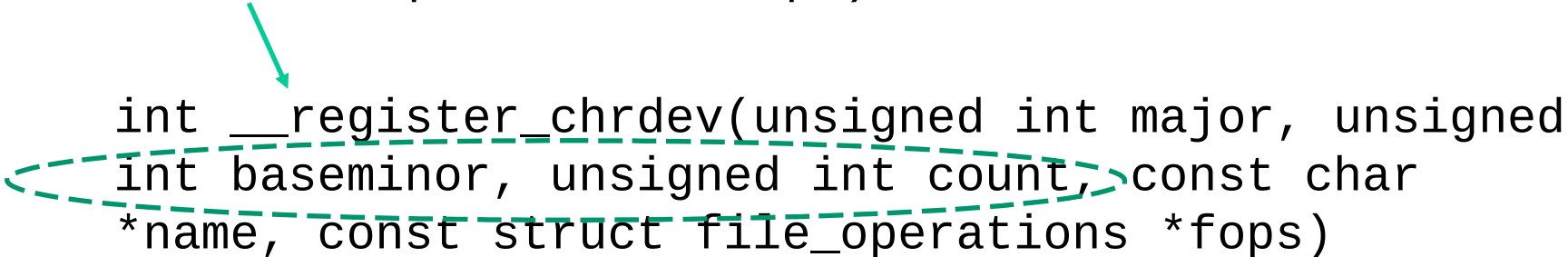


Operations remapping

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

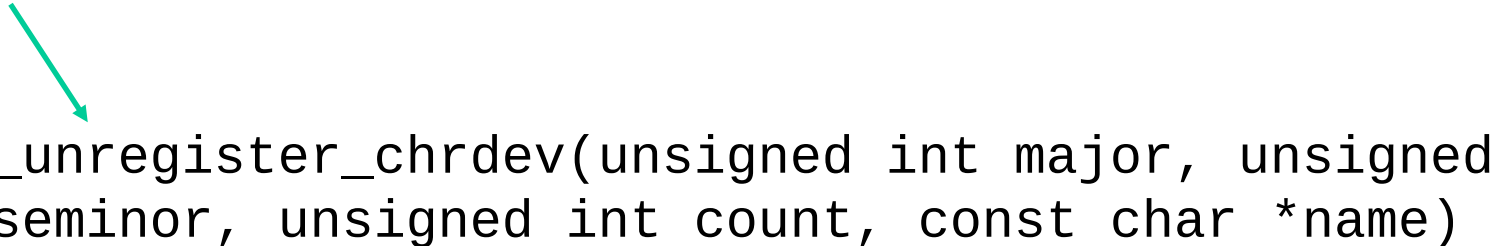
New
features →

```
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)
395 {
396     int pid;
397
398     rcu_scheduler_starting();
399     /*
400      * We need to spawn init first so that it obtains pid 1, however
401      * the init task will end up wanting to create kthreads, which, if
402      * we schedule it before we create kthreadd, will OOPS.
403 */
404     kernel_thread(kernel_init, NULL, CLONE_FS);
405     .....
406     numa_default_policy();
407     .....
408     ... ..
```

Switch off round-robin to first-touch

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(); ← registering drivers
    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.");
}
```

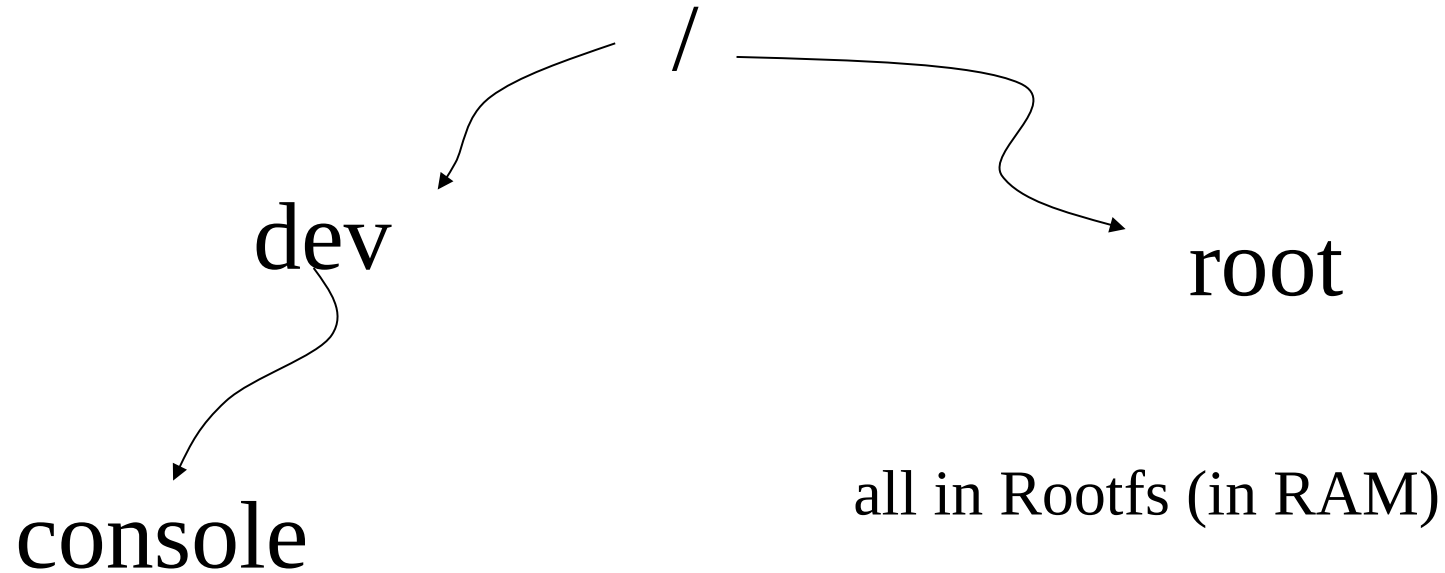
The prepare_namespace() function (2.4 style - minor variations are in kernels 3/4/5)

```
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()`



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 executing programs from this file system.

MS_RDONLY Mount file system read-only.

MS_REMOUNT Remount an existing mount. This allows you to change the mountflags and data of an existing mount without having to unmount and remount the file system. source and target should be the same value specified in the initial mount() call; filesystemtype is ignored.

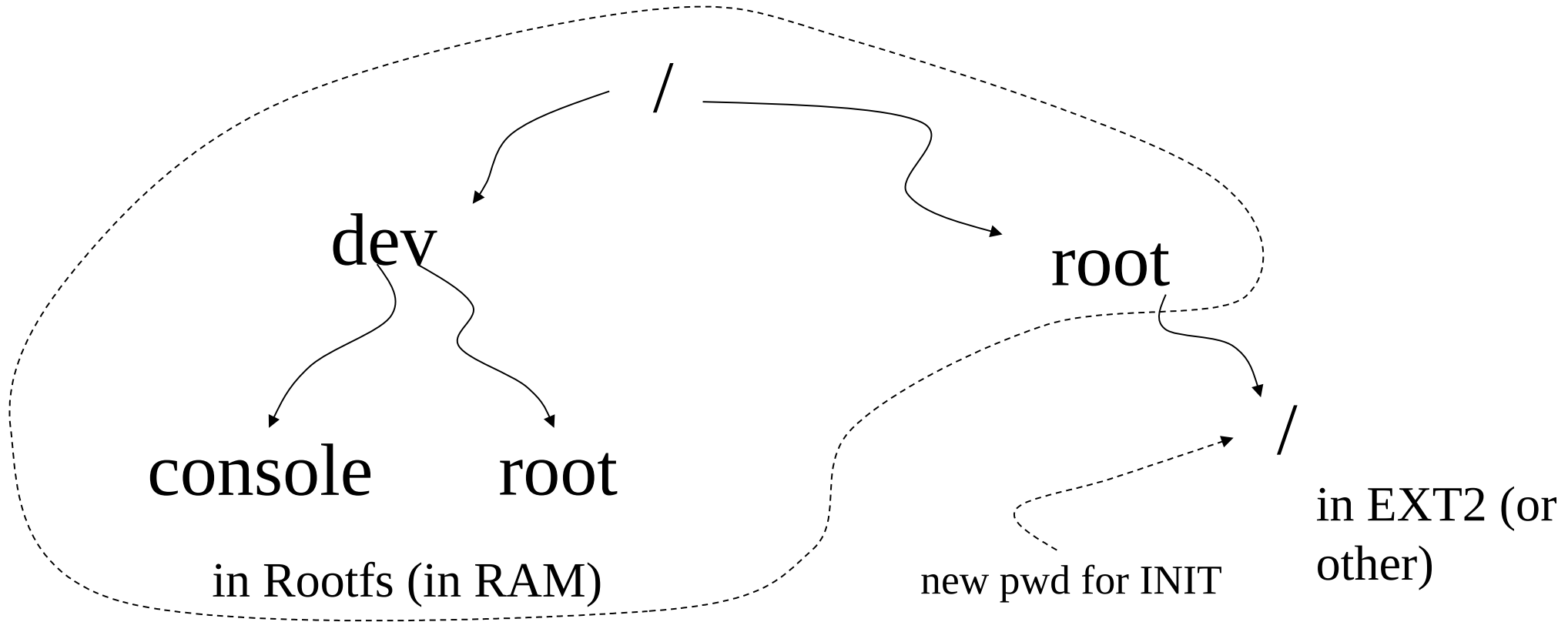
MS_SYNCHRONOUS Make writes on this file system synchronous (as though the O_SYNC flag to open(2) was specified for all file opens to this file system).

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

- 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
- This approach allows building views of the file system that can in general be articulated in a complex manner with respect to the mounted file system instances
- One of the advantages has been the introduction of “bind mounts” (more different paths towards the same mounted file system)

Description of open() – kernel side

The steps

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

Description of close() – kernel side

The steps

1. Release the dentry (by file descriptor) via `filp_close()` (internally calls `file_operation close`)
2. Release the file descriptor (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`