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
  - <u>FS independent part</u>: interfacing-layer towards other subsystems within the kernel
  - <u>FS dependent part</u>: data access/manipulation modules targeted at a specific file system type

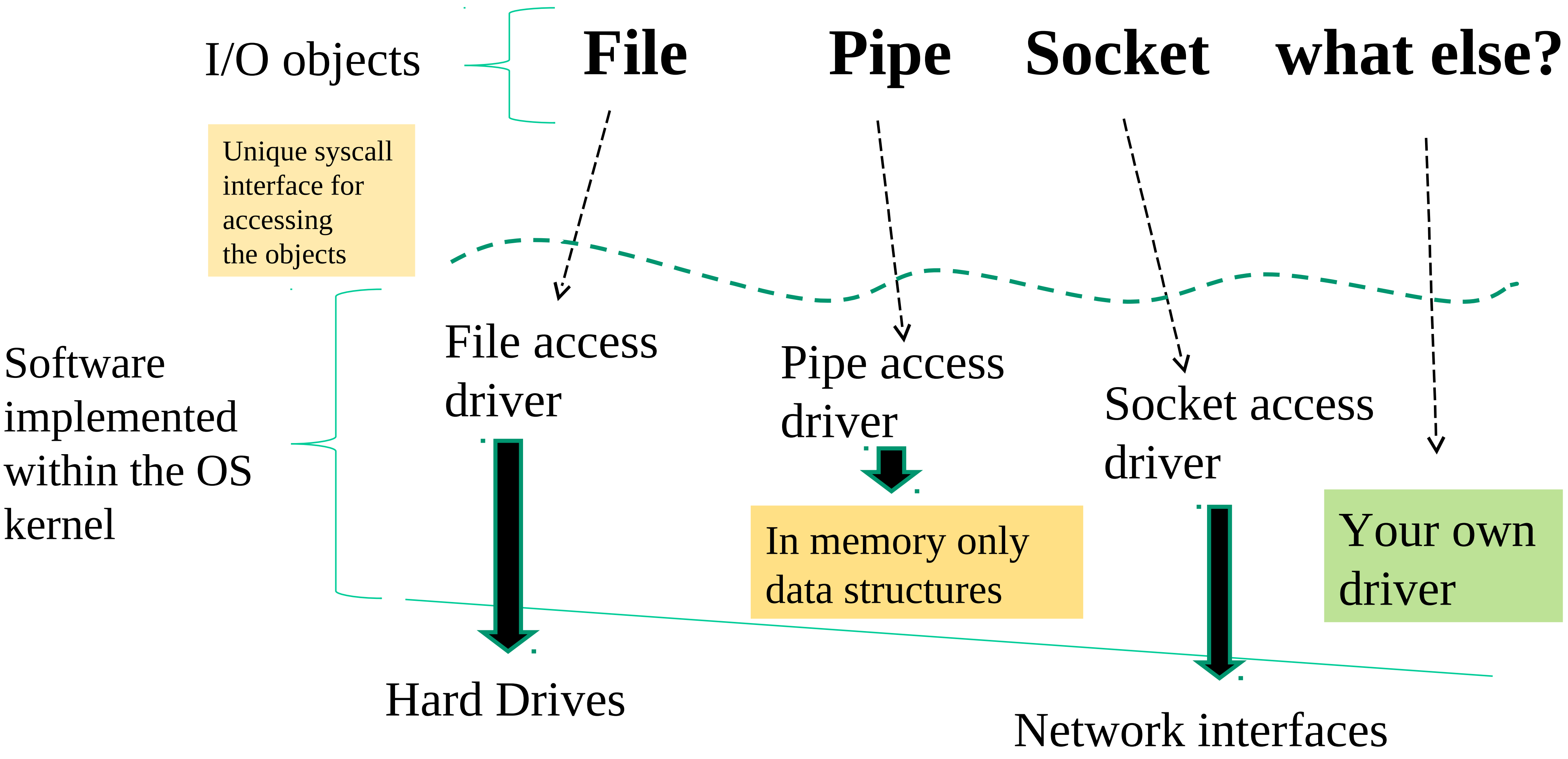
Connections

- 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

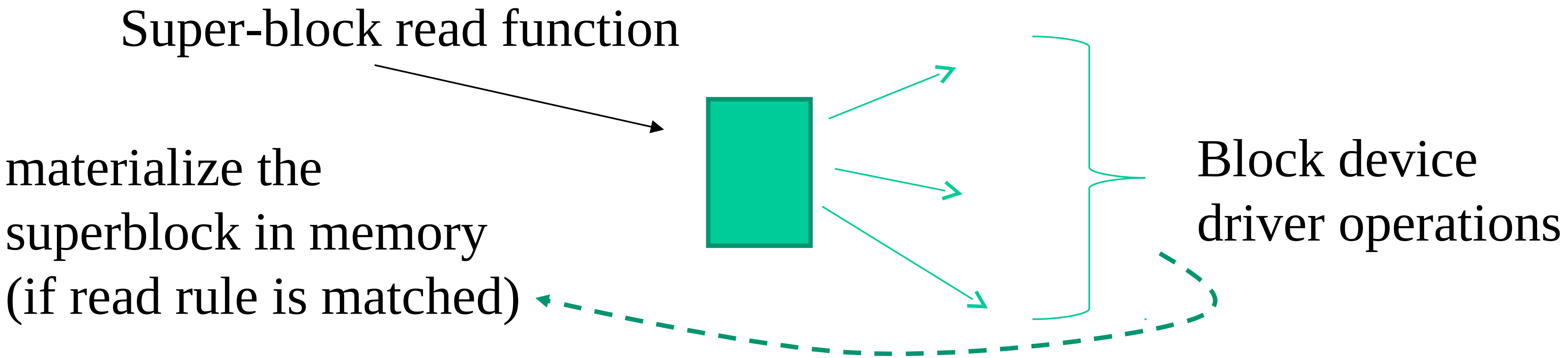
- Files are backed by data on a hard drive
- What **<u>software modules do we need</u>** for managing files on that hard drive in a well shaped OS-kernel??
  1. A function to <u>read the device superblock</u> for determining what files exist and where their data are
  2. A function to <u>read device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



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 **<u>fills</u>** the super-block content

Super-block read function

Fill callback-function

mount_bdev()

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

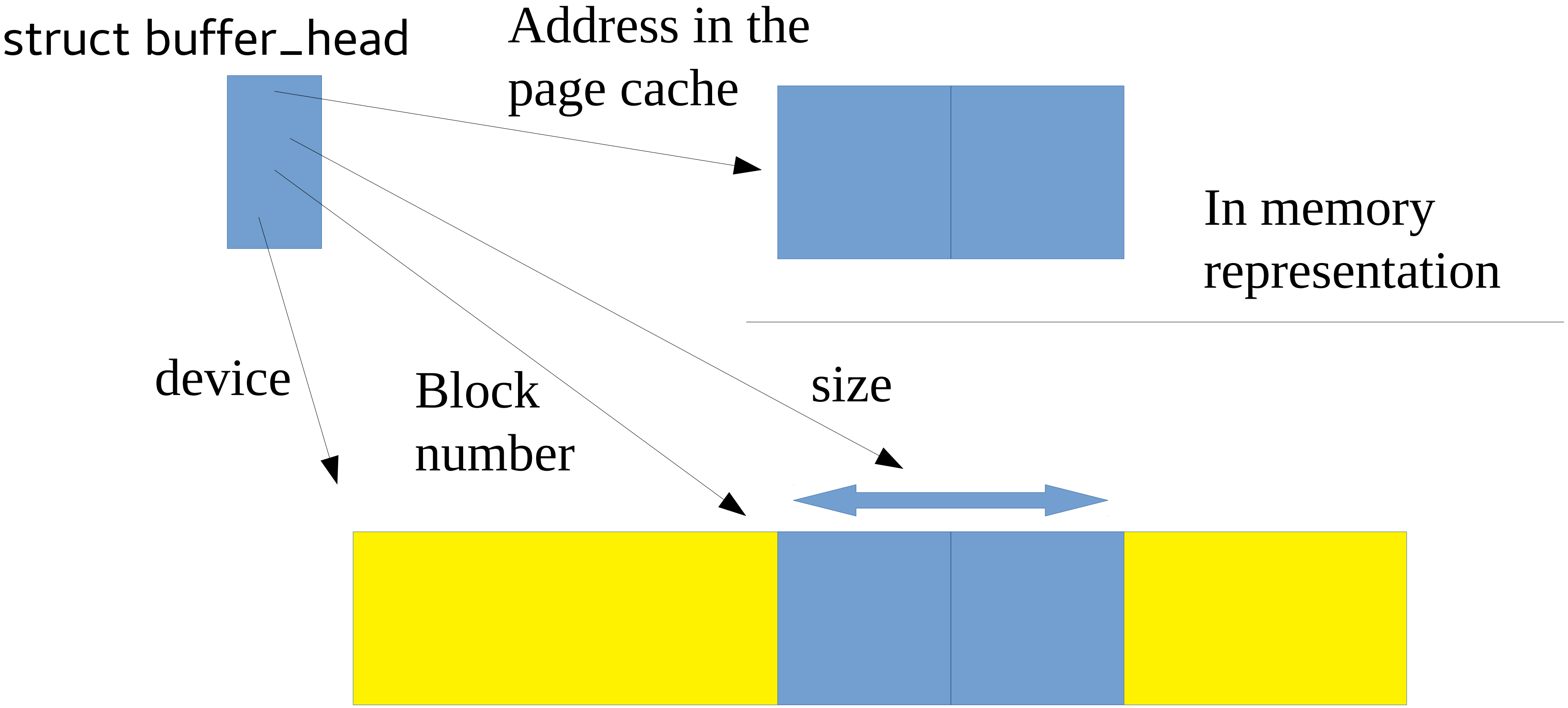
  - `*b_data`, pointer to a memory area where the data was read from or where the data must be written to

  - `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


struct buffer_head
Address in the page cache
In memory representation
device
Block number
size

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)

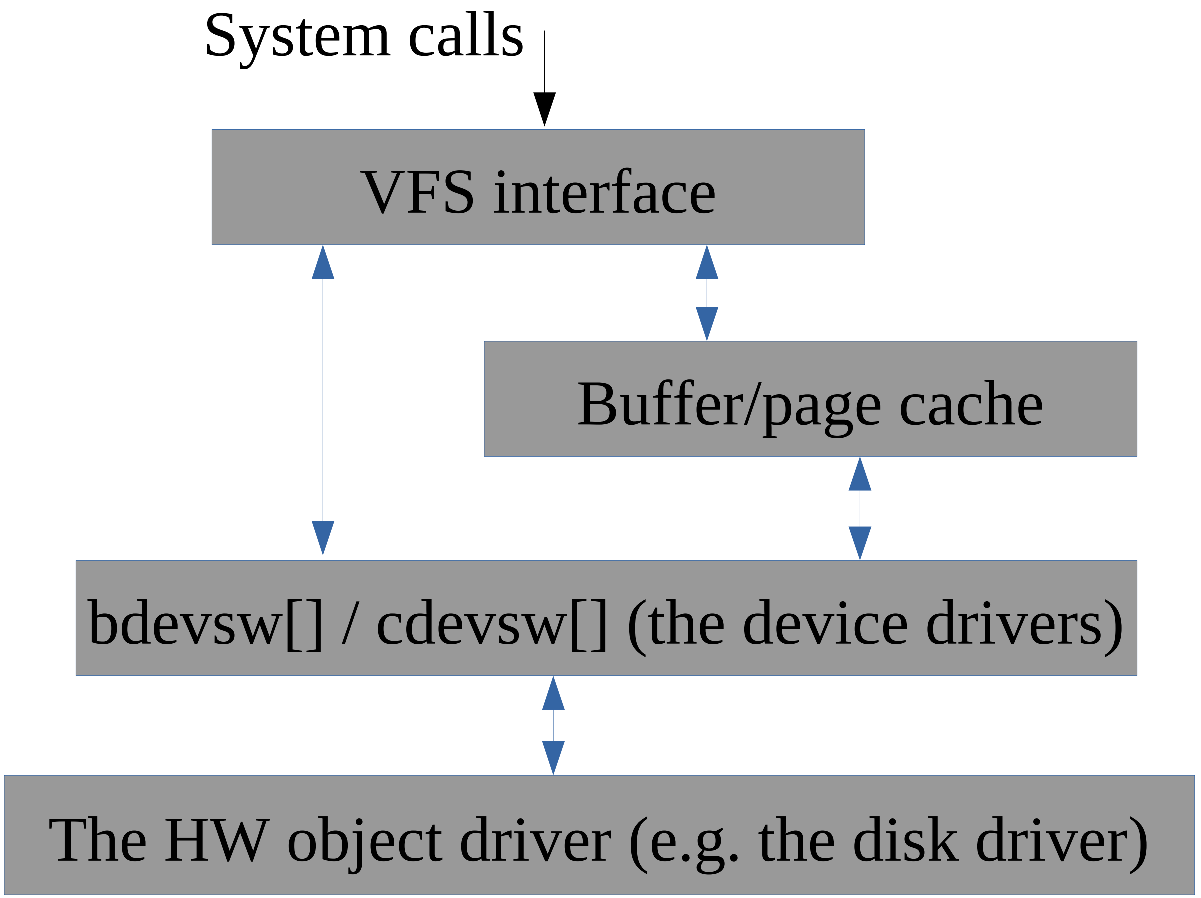
`sb_bread()` → the size of the read block is taken from the superblock;

`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);

`brelse()` → 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

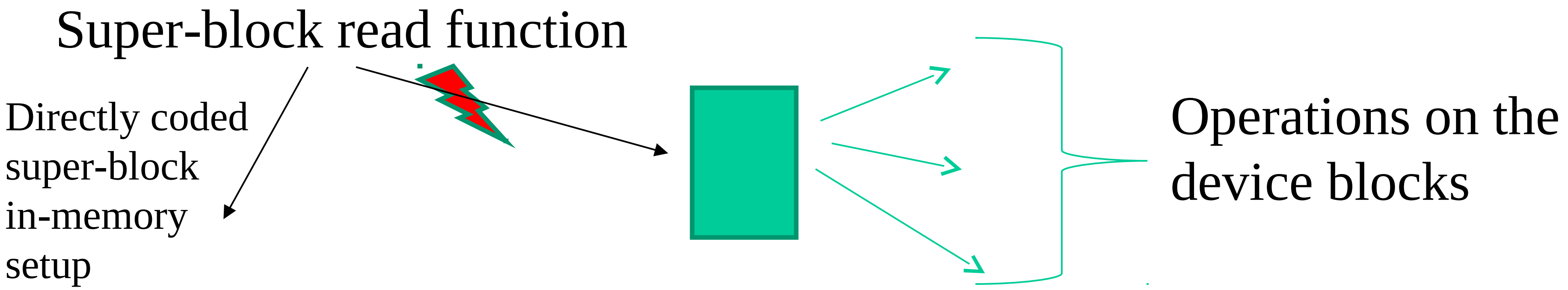


Regular files vs devices

- 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)
- <u>We can therefore create a stack of file system devices</u>

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



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 <u>inode</u> and the <u>dentry</u>

Baseline API for i-nodes and dentry

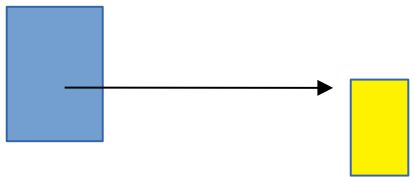
struct inode *new_inode(struct super_block *sb) → we simply allocate a generic i-node data structure making id refer to a generic super-bloc data structure

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

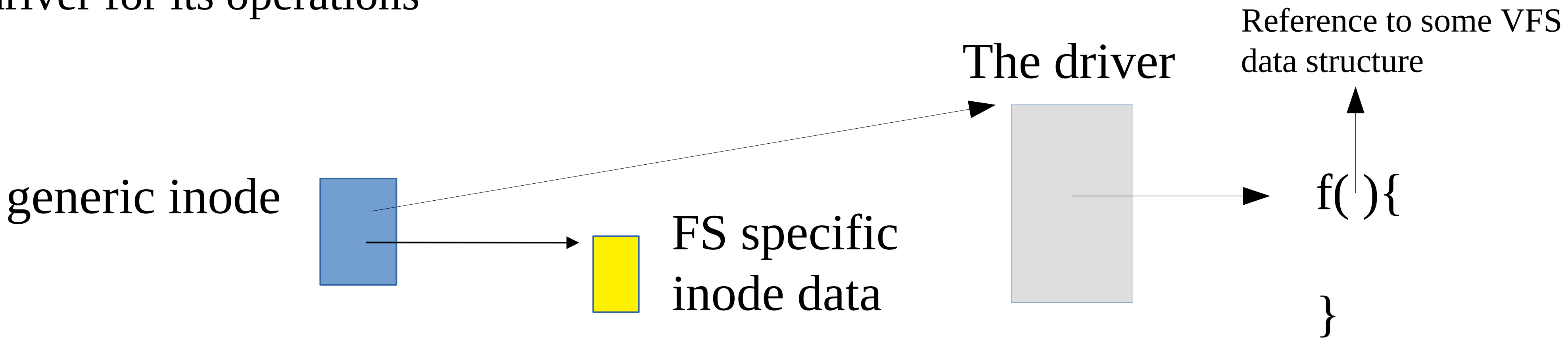
generic inode



FS specific inode data

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)

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 (i)

- Upon booting, a compile time defined instance of the structure `file_system_type` keeps meta-data for the **Rootfs**

- <u>This file system only lives in main memory</u> (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 exits 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 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 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 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 – 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; _not_ for fs use */
     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;
```

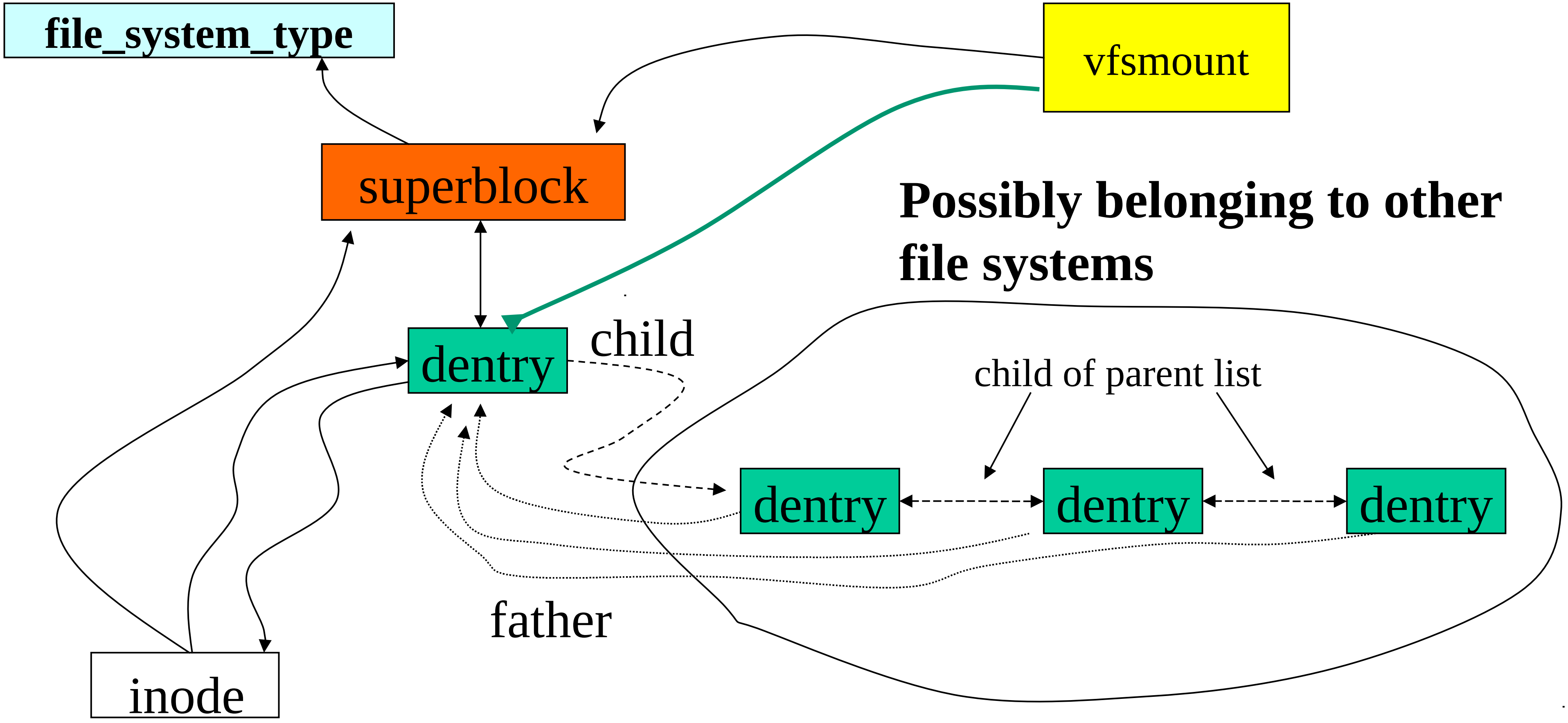
The structure dentry – Kernel 5 example

```
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

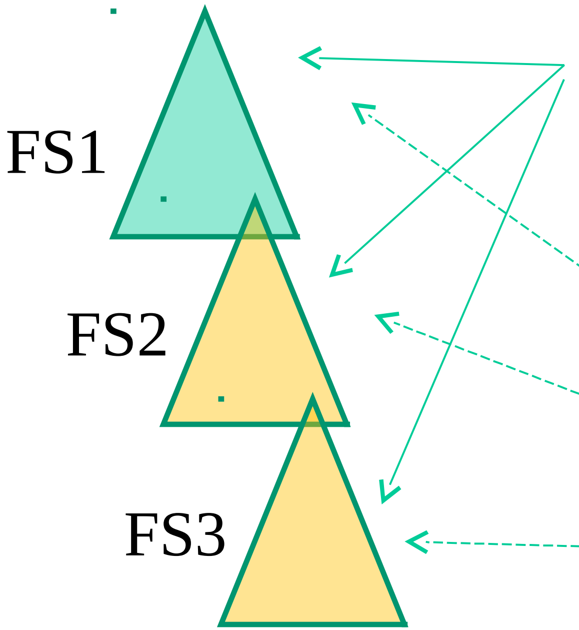
```
static void __init init_mount_tree(void){
    struct vfsmount *mnt;
    struct namespace *namespace;
    struct task_struct *p;

    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



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

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

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

Actual system calls for mount namespaces

```
clone(… int flags …)

                              CLONE_NEWNS

unshare(int flags)
```

An overall view

Super operations

Update superblock (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

TCB vs VFS

- The TCB keeps the field `struct fs_struct *fs` pointing to I nformation 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, * 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;
};
```

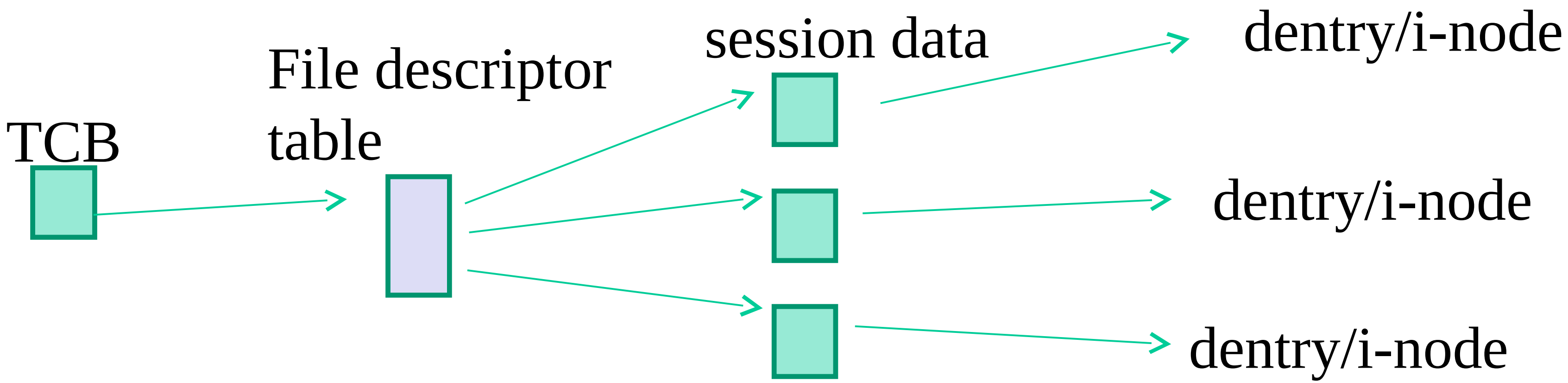
… and then 4.8 or later style

```
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
- 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;
  fd_set *open_fds;
  fd_set close_on_exec_init;
  fd_set open_fds_init;
  struct file * fd_array[NR_OPEN_DEFAULT];
};
```

bitmap for close on exec flags

bitmap identifying 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

`open()` system-call

kernel-level

`filp_open()`

i-node operation `lookup()`

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(......)`

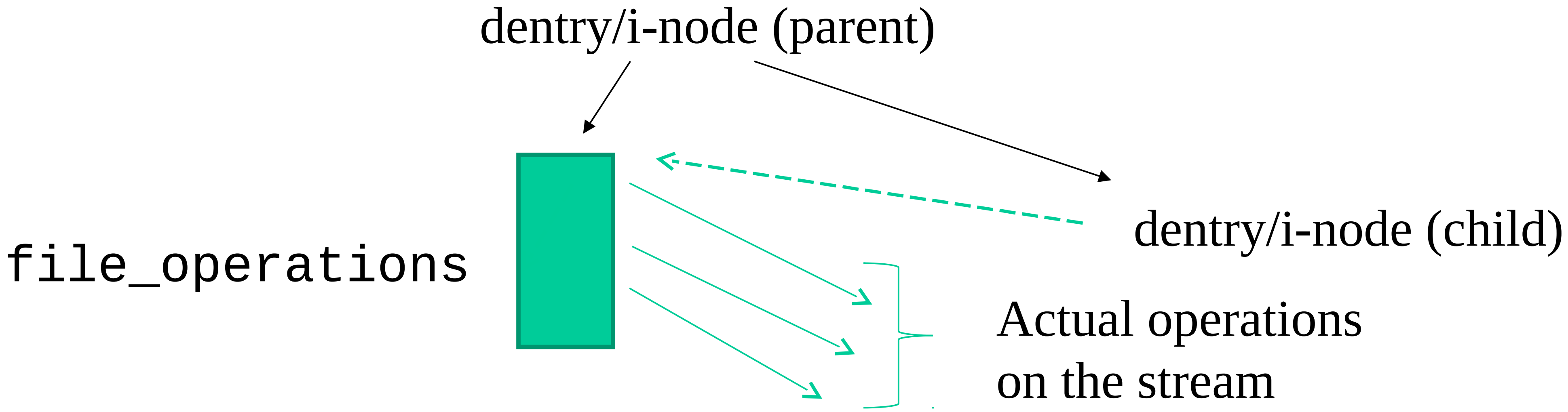
In the end we traverse dentry/i-node structures to retrieve the file operations table associated with that dentry

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

- `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 **<u>MAJOR and MINOR</u>** 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