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

Topics:

1. (Virtual) File Systems: design approach and architecture
2. 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/dev) 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

Linux 2.4 up to 4.17: main functions

- initialization of the file system takes place via an execution path complying with:
  - ➢`vfs_caches_init()` (in `fs/dcache.c`)
    - ✓ `mnt_init()` (in `fs/namespace.c`)
      - ✓`init_rootfs()` (in `fs/ramfs/inode.c`)
      - ✓`init_mount_tree()` (in `fs/namespace.c`)
- While setting up the VFS, the different types of file systems that are supported are defined, and the associated data structures are initialized
- Typically, at least two different FS types are supported
  - ➢Rootfs (file system in RAM)
  - ➢Ext
- 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

- The description of a specific FS type is done via the structure `file_system_type` defined in `include/linux/fs.h`
- This structure keeps infornation 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;
  ……
};
```

Rootfs (RAM file system)

- Upon booting the kernel, an instance of the structure `file_system_type` is allocated to keep meta-data for the **Rootfs**
- This file system only lives in main memory (hence it is re-initialized each time the kernel boots)
- The associated data act as initial "inspection" point for the identification of additional reachable 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
- They are
  - ➢ `DECLARE_FSTYPE(var,type,read,flags)` (in `include/linux/fs.h`)
  - ➢ `int register_filesystem(struct file_system_type *)` (in `fs/super.c`)

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

```
......
static DECLARE_FSTYPE(rootfs_fs_type, "rootfs",
    ramfs_read_super, FS_NOMOUNT|FS_LITTER);

......

int __init init_rootfs(void)
{
    return register_filesystem(&rootfs_fs_type);
}
```

2.4 kernel instance …

.. and then kernel 4.17 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` (in `include/linux/mount.h`)
  - `struct super_block` (in `include/linux/fs.h`)
  - `struct inode` (in `include/linux/fs.h`)
  - `struct dentry` (in `include/linux/dcache.h`)

- The instances of `struct vfsmount` and `struct super_block` keep file system proper information (e.g. in terms of relation with other file systems)

- The instances of `struct inode` and `struct dentry` are such that one copy exits for any file/directory of the specific file system

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

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

…. now structured this way in 4.xx

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

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

`randstruct`

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

The structure super_block

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

The structure dentry

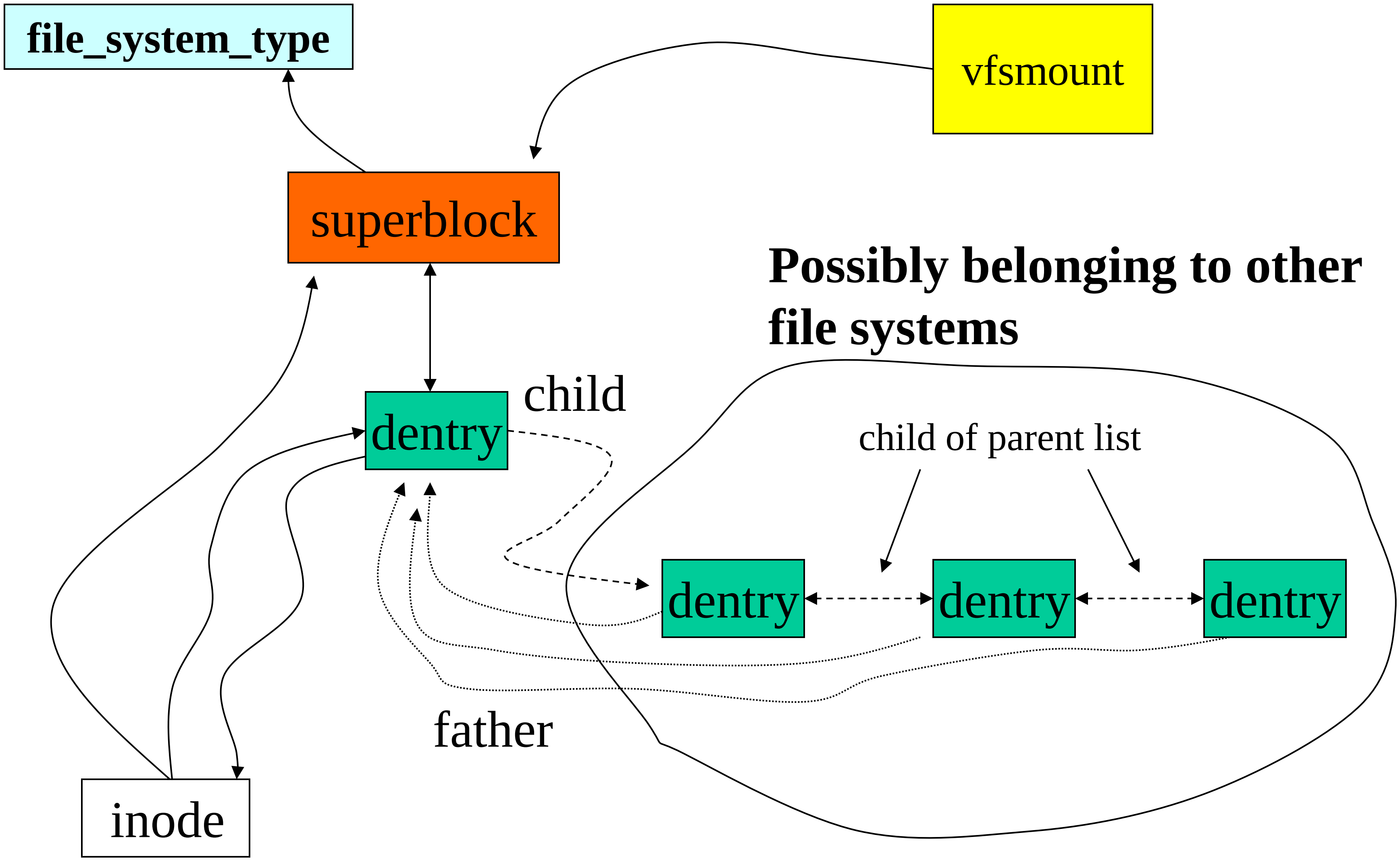
```
struct dentry {
        atomic_t d_count;
        ……
        struct inode  * d_inode;   /* Where the name belongs to */
        struct dentry * d_parent;  /* parent directory */
        struct list_head d_hash;   /* lookup hash list */
        ……
        struct list_head d_child;  /* child of parent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

This is for “short” names

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

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

Overall scheme



Initializing the Rootfs instance

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

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

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

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

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

VFS vs PCBs (2.4 style)

- The PCB 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;
};
```

- For the IDLE PROCESS we will get that both `root` and `pwd` point the only existing dentry (at this point of the boot)

3.xx kernel style

See **linux**/**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 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

Reading the Rootfs super-block (2.4 style)

- As said, the super-block read-function for **Rootfs** is `ramfs_read_super`, which is defined in `fs/ramfs/inode.c`

```
static struct super_block *ramfs_read_super(struct super_block
                      * sb, void * data, int silent)
{
  struct inode * inode;
  struct dentry * root;
  sb->s_blocksize = PAGE_CACHE_SIZE;
  sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
  sb->s_magic = RAMFS_MAGIC;
  sb->s_op = &ramfs_ops;
  inode = ramfs_get_inode(sb, S_IFDIR | 0755, 0);
  if (!inode)
        return NULL;
  root = d_alloc_root(inode);
  if (!root) {
        iput(inode);
        return NULL;
  }
  sb->s_root = root;
  return sb;
}
```

Super-block operations

- Generally speaking, super-block operations are in charge of
  - Managing statistic on the file system
  - Creating and managing i-nodes
  - Flushing onto the device any updated information on the state of the file system
- In some case a few of these functionalities are not actually used (depending on the particular type of file system, e.g. the file system in RAM)
- Triggering the kernel functions for accessing statistics takes place via the system calls `statfs` and `fstatfs`

`struct super_operations` (2.4 style)

- It is defined in `include/linux/fs.h`

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*read_inode) (struct inode *);
    void (*read_inode2) (struct inode *, void *) ;
    void (*dirty_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs) (struct super_block *);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    struct dentry * (*fh_to_dentry)(struct super_block *sb,
                          __u32 *fh, int len, int fhtype, int parent);
    int (*dentry_to_fh)(struct dentry *, __u32 *fh, int *lenp,
                          int need_parent);
    int (*show_options)(struct seq_file *, struct vfsmount *);
};
```

An example for the file system in RAM

- the modules are defined in `fs/ramfs/inode.c`

```
static int ramfs_statfs(struct super_block *sb,
                                struct statfs *buf)
{
   buf->f_type = RAMFS_MAGIC;
   buf->f_bsize = PAGE_CACHE_SIZE;
   buf->f_namelen = NAME_MAX;
   return 0;
}


static struct super_operations ramfs_ops = {
   statfs:            ramfs_statfs,
   put_inode:         force_delete,
}
```

Dentry operations (2.4 style)

- They specify non-default operations for manipulating d-entries
- The table maintaining the associated function pointers is defined in `include/linux/dcache.h`
- For the file system in RAM this structure is not used

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, int);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *,
                      struct qstr *, struct qstr *);
    void (*d_delete)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
};
```

Removes the pointed i-node (to be done when releasing the dentry)

Removes the dentry, which is done when the reference counter is set to zero

3.xx kernel style

See linux/include/linux/dcache.h

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, unsigned int);
        int (*d_weak_revalidate)(struct dentry *, unsigned int);
        int (*d_hash)(const struct dentry *, struct qstr *);
        int (*d_compare)(const struct dentry *, const struct dentry *,
                        unsigned int, const char *, const struct qstr *);
        int (*d_delete)(const struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_prune)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
        char *(*d_dname)(struct dentry *, char *, int);
        struct vfsmount *(*d_automount)(struct path *);
        int (*d_manage)(struct dentry *, bool);
}
```

i-node operations (2.4 style)

- They specify i-node related operations
- The table maintaining the corresponding function pointers is defined in `include/linux/fs.h`

```
struct inode_operations {
 int (*create) (struct inode *,struct dentry *,int);
 struct dentry * (*lookup) (struct inode *,struct dentry *);
 int (*link) (struct dentry *,struct inode *,struct dentry *);
 int (*unlink) (struct inode *,struct dentry *);
 int (*symlink) (struct inode *,struct dentry *,const char *);
 int (*mkdir) (struct inode *,struct dentry *,int);
 int (*rmdir) (struct inode *,struct dentry *);
 int (*mknod) (struct inode *,struct dentry *,int,int);
 int (*rename) (struct inode *, struct dentry *,
               struct inode *, struct dentry *);
 int (*readlink) (struct dentry *, char *,int);
 int (*follow_link) (struct dentry *, struct nameidata *);
 void (*truncate) (struct inode *);
 int (*permission) (struct inode *, int);
 ………
};
```

An example for the file system in RAM

- i-node operations for the file system in RAM are defined in `fs/ramfs/inode.c` as follows

```
static struct inode_operations
ramfs_dir_inode_operations = {
   create:          ramfs_create,
   lookup:          ramfs_lookup,
   link:            ramfs_link,
   unlink:          ramfs_unlink,
   symlink:         ramfs_symlink,
   mkdir:           ramfs_mkdir,
   rmdir:           ramfs_rmdir,
   mknod:           ramfs_mknod,
   rename:          ramfs_rename,
};
```

`struct nameidata`

- `struct nameidata` is exploited (especially in terms of parameter values) in several VFS operations
- It is defined in `include/linux/fs.h` as follows

```
struct nameidata {
    struct dentry *dentry;
    struct vfsmount *mnt;
    struct qstr last;
    unsigned int flags;
    int last_type;
};
```

management of strings

VFS intermediate functions (2.4/2.6 style)

- These functions are invoked as a result of system call execution
- They ultimately rely on per-file-system internal management functions, such as dentry, inode and super block operations
- Here is a list of some important intermediate functions

```
int path_lookup(const char *path, unsigned flags,
        struct nameidata *nd)
```
(in `fs/namei.c`)

provides within `struct nameidata` 3 field values: (parent) dentry in the path, vfsmount, reference to the string related to the last element in path. The actual behavior depends on `flags` (LOOKUP_PARENT)

```
static struct dentry *lookup_create(struct nameidata
        *nd, int is_dir)
```
(in `fs/namei.c`)

Returns the address of the dentry whose name in path is defined by the string `nd`. If this dentry does not exist (no dir/file has that name) it gets created `int vfs_mkdir(struct inode *dir, struct dentry  *dentry, int mode)` (in `fs/namei.c`)

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

`static __inline__ struct dentry * dget(struct   dentry *dentry)` (in `include/linux/dcache.h`)

Acquires a dentry (by incrementing the reference counter)

`void dput(struct dentry *dentry)` (in `include/linux/dcache.h`)

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

`long do_mount(char * dev_name, char * dir_name, char *type_page, unsigned long flags, void *data_page)` (in `fs/namespace.c`)

Mounts a device (hence the corresponding file system) onto a target directory

`static struct inode *alloc_inode(struct super_block *sb)`
(in `fs/inode.c`)

allocates an i-node and initializes it according to the specific file system rules

```
struct dentry * lookup_hash(struct qstr *name,
                            struct dentry * base)
```

takes a pointer to a dentry directory (namely `base`) and the name of a child (namely `name`), and returns a pointer to the child dentry, if any

```
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`

`int do_truncate(struct dentry *dentry, loff_t length)`

Reduces the length of the file associated with `dentry` to the value `length`. Returns 0 upon success

Example relations with FS dependent functions

`path_lookup()`

When the exploration in the path requires accessing the specific file system

i-node operation `lookup()`

when using flags LOOKUP_FOLLOW (symbolic links)

i-node operation `do_follow_link()`

`alloc_inode()`

super operation `alloc_inode()`

`dput()`

dentry operation `d_delete()`

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 which 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/4.xx 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 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

Device numbers

- LINUX (just like any UNIX-like system) associates with any device a couple of numbers called: MAJOR and MINOR
- MAJOR is used as the key for accessing the device driver as registered within a proper table
- MINOR identifies the actual instance of the device driven by that driver (this can be selected/imposed by the driver programmer)
- There exist different tables for registering devices, depending on whether the device is a char or a block one
- These are defined in
  - `fs/devices.c` for char devices
  - `fs/block_dev.c` for block devices
- All the drives that have ben selected while configuring the kernel are registered upon booting
- In the above source files we can also find device independent functions for accessing the actual driver

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 o 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

- It is defined in `include/linux/fs.h`

```
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);
};
```

Block devices table

```
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

- It is defined in `include/linux/fs.h`

```
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;
};
```

Device numbers

- for i386 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

VFS “i-node”

- The field `umode_t i_mode` within `struct inode` keps an information indicating the type of the i-node
- Classical types are
  - 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 **<u>MAJOR and MINOR numbers for the device file that gets created</u>**, otherwise this parameter is a don’t care

Kernels 3/4: 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

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)
```

Dicothomy

- For char devices the management of read/write operations is in charge of the device driver
- This is not the same for block devices
- read/write operations on block devices are handled via a single API related to buffer cache operations
- The actual implementation of the buffer cache policy will determine the real execution activities for block device read/write operations

Common interface

- It is called “requests” in LINUX – or “strategy” in UNIX systems
- It encapsulated the optimizations for managing each specific device (e.g. via the elevator algorithm)
- The request interface is associated with a queue of pending requests towards the block device
- The association request-queue/major-number is done via the array `blk_dev[]`

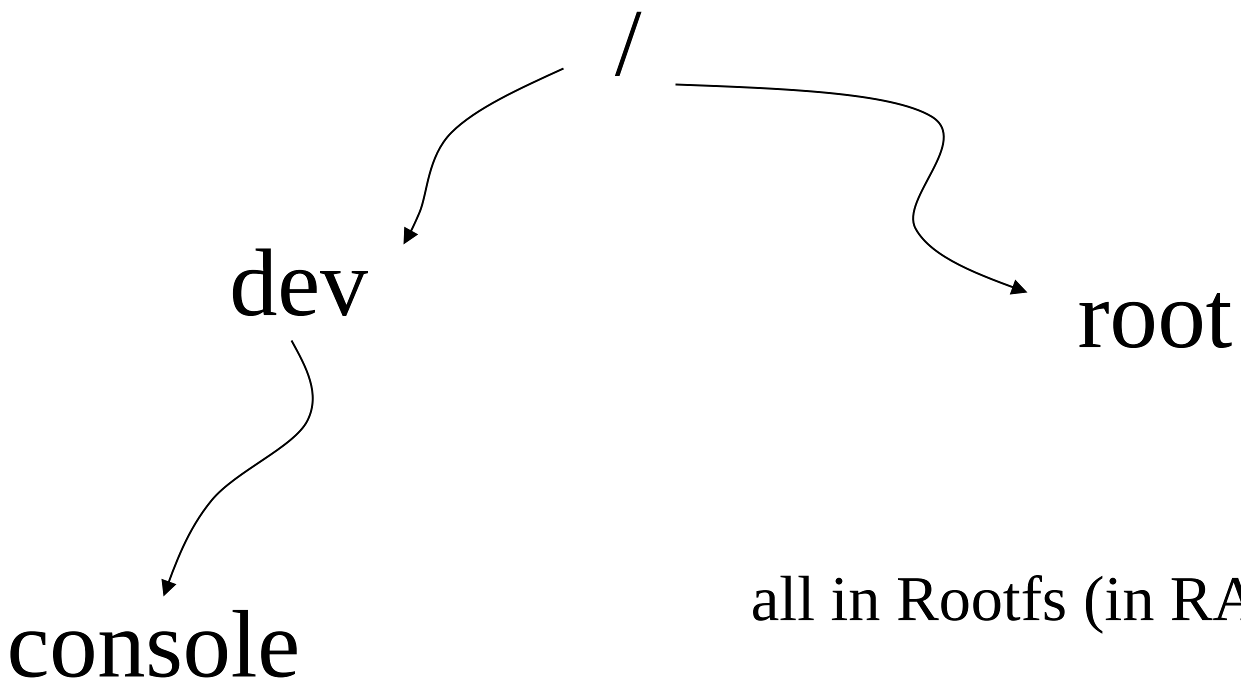
The `prepare_namespace()` function (2.4 style - minor variations are in kernels 3/4)

- it is defined in `init/do_mounts.c`

```
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()`



all in Rootfs (in RAM)

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 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);
```

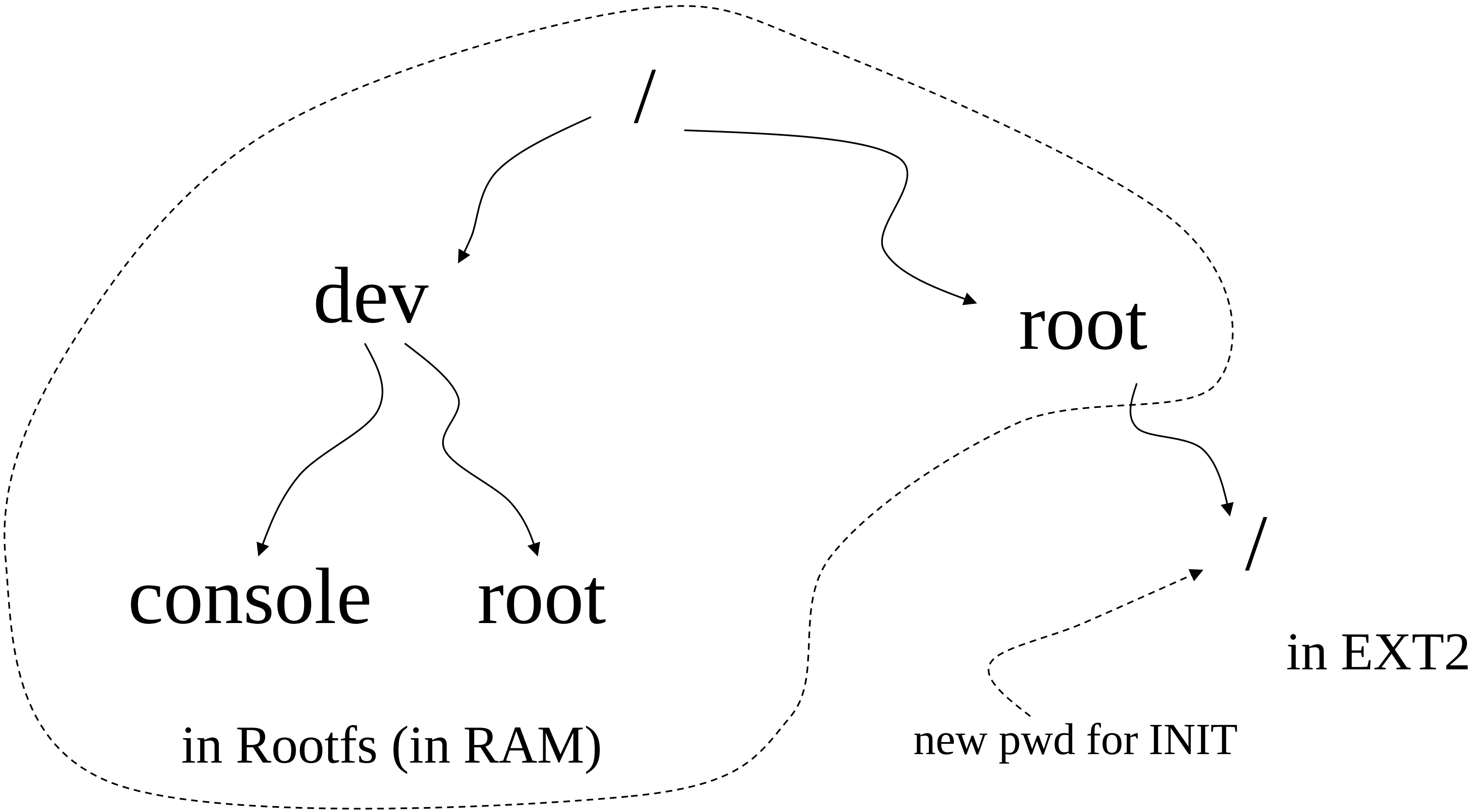
```
MS_NOEXEC      Do not allow programs to be executed from this file
               system.

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 onto “/” of Rootfs

File descriptor table

- PCB keeps the field `struct files_struct *files` which points to the descriptor table
- This table is defined in `include/linux/sched.h` 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];
};
```

`struct file` (2.4 style)

- This is defined in `include/linux/fs.h`

```
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 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;

.........
..........
```

Opening a file (2.4/2.6 case)

- The function `sys_open()` defined in `fs/open.c` is formed by two parts
  - The first one is coded within the function and allocates a file descriptor (hence it reserves a pointer to `struct file`)
  - The second relies on an invocation the intermediate function `struct file *filp_open(const char * filename, int flags, int mode)` which returns the address of the `struct file` associated with the opened file

The function `filp_open()`

- It executes the following two tasks
  1. Creation/opening of the dentry and of the i-node associated with the file
  2. Creation of the `struct file` for the working session on the file

- The first task is carried out via the function `open_namei()` defined in `fs/namei.c` which relies on
  - `path_lookup()` (in `fs/namei.c`)
  - `lookup_hash()` (in `fs/namei.c`)
  - `vfs_create()` (in `fs/namei.c`)
  - `do_truncate()` (in `fs/open.c`)

- The second task exploits the `dentry_open()` function

`int open_namei(const char * pathname, int flag,`
`                  int mode, struct nameidata *nd)`

Creates/opens i-node and dentry associated with `pathname` and outputs the pointer to that dentry via `nd`. Returns 0 upon success.

`flag` is the same as the one passed to the open system call wih the only difference that the two least significant bits have the following semantic :

```
* Note that the low bits of "flag" aren't the same as in the open
* system call - they are 00 - no permissions needed
*                         01 - read permission needed
*                         10 - write permission needed
*                         11 - read/write permissions needed
* which is a lot more logical, and also allows the "no perm" needed
* for symlinks (where the permissions are checked later).
```

"mode" represents the standard mode of the open system call.

`struct file *dentry_open(struct dentry *dentry,`
`                         struct vfsmount *mnt, int flags)`

Creates `struct file` associated with the dentry pointed by `dentry` and related to the file system `mnt`. Returns the pointer to `struct file`. `flags` corresponds to the flags in input to the open system call

The `sys_open()` code

```
asmlinkage long sys_open(const char * filename, int flags, int mode){
        char * tmp;
        int fd, error;
        …
        tmp = getname(filename);
        fd = PTR_ERR(tmp);
        if (!IS_ERR(tmp)) {
                fd = get_unused_fd();
                if (fd >= 0) {
                        struct file *f = filp_open(tmp, flags, mode);
                        error = PTR_ERR(f);
                        if (IS_ERR(f))
                                goto out_error;
                        fd_install(fd, f);
                }
out:
                putname(tmp);
        }
        return fd;
out_error:
        put_unused_fd(fd);
        fd = error;
        goto out;
}
```

Pointing to `f`

```
int get_unused_fd(void){
        struct files_struct * files = current->files;
        int fd, error;

        error = -EMFILE;
        write_lock(&files->file_lock);

repeat: fd = find_next_zero_bit(files->open_fds,
                                files->max_fdset,
                                files->next_fd);

        /*
         * N.B. For clone tasks sharing a files structure, this test
         * will limit the total number of files that can be opened.
         */
        if (fd >= current->rlim[RLIMIT_NOFILE].rlim_cur)
                goto out;

        /* Do we need to expand the fdset array? */
        if (fd >= files->max_fdset) {
                error = expand_fdset(files, fd);
                if (!error) {
                        error = -EMFILE;
                        goto repeat;
                }
                goto out;
        }
```

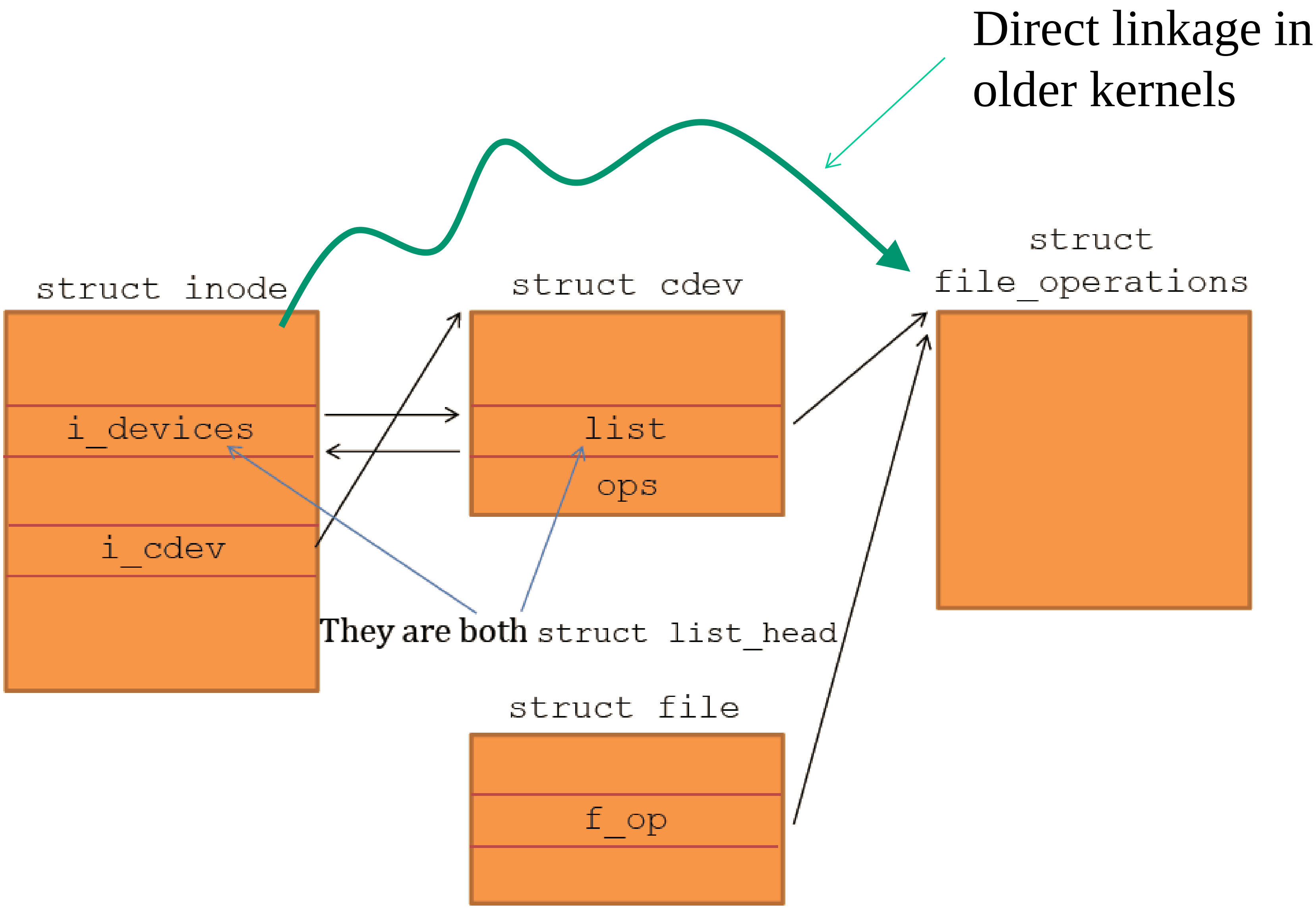
```
        /*
         * Check whether we need to expand the fd array.
         */
        if (fd >= files->max_fds) {
                error = expand_fd_array(files, fd);
                if (!error) {
                        error = -EMFILE;
                        goto repeat;
                }
                goto out;
        }
        FD_SET(fd, files->open_fds);
        FD_CLR(fd, files->close_on_exec);
        files->next_fd = fd + 1;
#if 1
        /* Sanity check */
        if (files->fd[fd] != NULL) {
             printk(KERN_WARNING "get_unused_fd: slot %d not NULL!\n", fd)
             files->fd[fd] = NULL;
        }
#endif
        error = fd;

out:
        write_unlock(&files->file_lock);
        return error;
}
```

new standard

Helps

- in `include/linux/time/h` we can find macros for manipulating bitmaps associated with open files
- They are
  - `FD_SET(fd,fdsetp)`
  - `FD_CLR(fd,fdsetp)`
- `fdsetup` identifies the bitmap where the bit at offset `fd` needs to be set/unset

A scheme on i-node to file operations mapping for kernels 3/4



Closing a file

- The function `sys_close()` defined in `fs/open.c` is formed by two parts
  - The first one consists of a set of statements for releasing he file descriptor associated with the open file
  - The second one relies on the intermediate function `int filp_close(struct file *filp, fl_owner_t id),` defined in `fs/open.c`, which is in charge of flushing the data structures associated with the file (`struct file`, `dentry` and `i-node`)

The function `filp_close()`

- This relies on the following internal functions

```
void dnotify_flush(struct file *filp, fl_owner_t id)
```
(in `fs/dnotify.c`)

Notifies that file slushing has been finalized dso that dentry and i-node operations can be carried out

```
void locks_remove_posix(struct file *filp,
                        fl_owner_t owner)
```
(in `fs/locks.c`)

Removes the lock on `struct file`

```
void fput(struct file * file)
```
(in `fs/file_table.c`)

deallocates `struct file`

The sys_close() code

```
asmlinkage long sys_close(unsigned int fd)
{
        struct file * filp;
        struct files_struct *files = current->files;

        write_lock(&files->file_lock);
        if (fd >= files->max_fds)
                goto out_unlock;
        filp = files->fd[fd];
        if (!filp)
                goto out_unlock;
        files->fd[fd] = NULL;
        FD_CLR(fd, files->close_on_exec);
        __put_unused_fd(files, fd);
        write_unlock(&files->file_lock);
        return filp_close(filp, files);

out_unlock:
        write_unlock(&files->file_lock);
        return -EBADF;
}
```

```
static inline void __put_unused_fd(struct files_struct
                        *files, unsigned int fd)
{
  FD_CLR(fd, files->open_fds);
  if (fd < files->next_fd)
        files->next_fd = fd;
}
```

- This is defined in `include/linux/file.h`

The write system call

- The function `sys_write()` is defined in `fs/read_write.c` as shown below

```
asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t
  count) {
  ssize_t ret; struct file * file;

  ret = -EBADF; file = fget(fd);
  if (file) {
  if (file->f_mode & FMODE_WRITE) {
       struct inode *inode = file->f_dentry->d_inode;
       ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file,
                       file->f_pos, count);
    if (!ret) {
       ssize_t (*write)(struct file *, const char *, size_t, loff_t *);
       ret = -EINVAL;
       if (file->f_op && (write = file->f_op->write) != NULL)
             ret = write(file, buf, count, &file->f_pos);
          }
     }
    if (ret > 0)       dnotify_parent(file->f_dentry, DN_MODIFY);
    fput(file);
  }
  return ret;
}
```

The read system call

- The function `sys_read()` is defined in `fs/read_write.c` as shown below

```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf,    size_t
 count){
 ssize_t ret;  struct file * file;

 ret = -EBADF;
 file = fget(fd);
 if (file) {
 if (file->f_mode & FMODE_READ) {
   ret = locks_verify_area(FLOCK_VERIFY_READ, file->f_dentry->d_inode,
                               file, file->f_pos, count);
   if (!ret) {
       ssize_t (*read)(struct file *, char *, size_t, loff_t *);
       ret = -EINVAL;
       if (file->f_op && (read = file->f_op->read) != NULL)
       ret = read(file, buf, count, &file->f_pos);
       }
 }
 if (ret > 0)  dnotify_parent(file->f_dentry, DN_ACCESS);
 fput(file);
 }
 return ret;
}
```

`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 poc 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

- proc exploits the following data structure defined in `include/linux/proc_fs.h`

```
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;
};
```

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
- There exists a root `proc_dir_entry`, which is linked to the i-node and to the root dentry of proc

Mounting proc

- The proc file system is not 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
- This is done in relation to information provided by `/etc/fstab`
- Typically, the root of EXT2 keeps the directory `/proc` that is exploited as the mount point for proc
- **NOTE:** given that proc 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

Specific identifiers

```
struct vfsmount *proc_mnt;
         (in fs/proc/inode.c)

struct proc_dir_entry *proc_net,
*proc_bus, *proc_root_fs,
*proc_root_driver;
         (in fs/proc/root.c)
```

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, 4096 on arm and i386.)
`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 2.6 kernels)

- 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

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 objects 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

The specification of the actual read/write operations occurs via the kobject parameter

struct kobject->kobj_type->sysfs_ops

```
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);
};
```