ARCOS Group
Computer Science and Engineering Department
Universidad Carlos III de Madrid

Lesson 5
File Systems

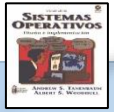
Operating System Design
Degree in Computer Science and Engineering, Double Degree CS&E + BA

Recommended readings

Base

1. Carretero 2007:
    1. Chapter 9

Recommended



1. Tanenbaum 2006(en):
    1. Chap.5
2. Stallings 2005:
    1. Three part
3. Silberschatz 2014:
    1. Chap. 10, 11 & 12

To remember…

1. To study the associated theory.
   - To study the bibliography material: slides only are not enough.
2. To review the class explanations.
   - To perform the guided laboratory progressively.
3. To exercise competitions.
   - To build the laboratory progressively.
   - To build as much exercise as possible.

Overview

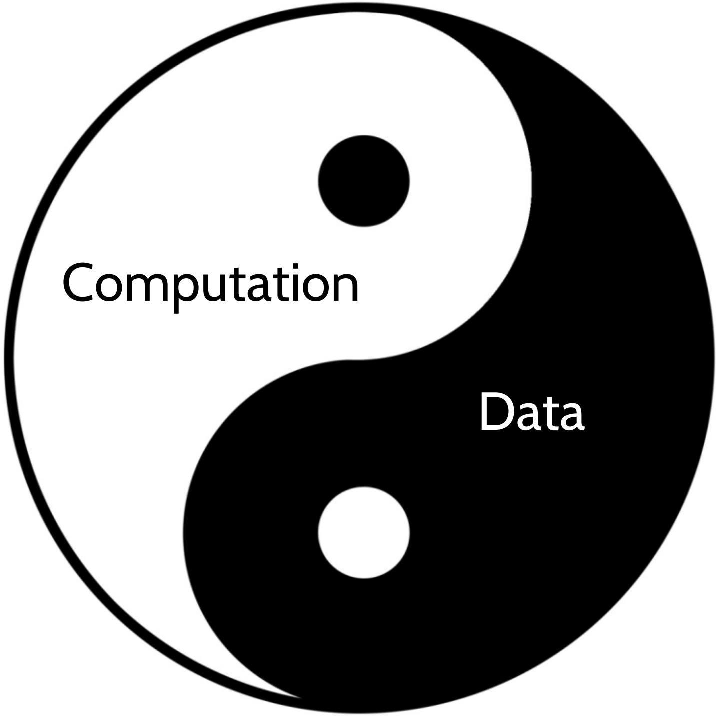
1. Introduction
2. File system internals and framework
3. Design and development of a file system
4. Complementary aspects

Overview

1. **Introduction**
2. File system internals and framework
3. Design and development of a file system
4. Complementary aspects

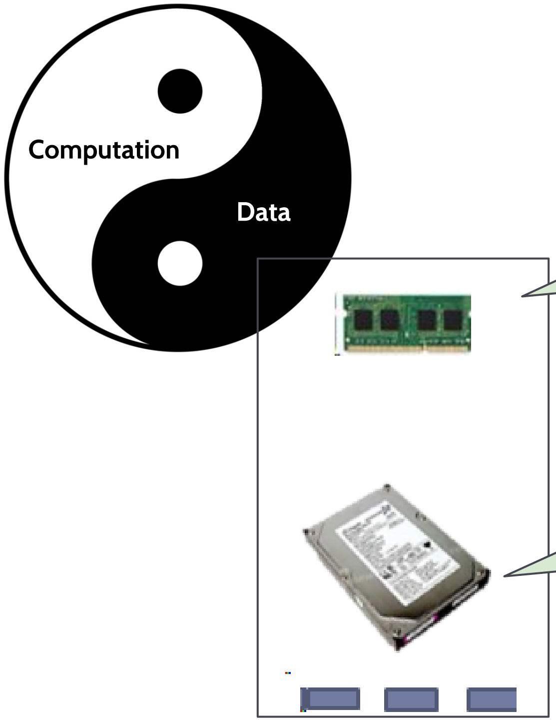
Storage System Scope

~2020



Storage System Scope

~2020

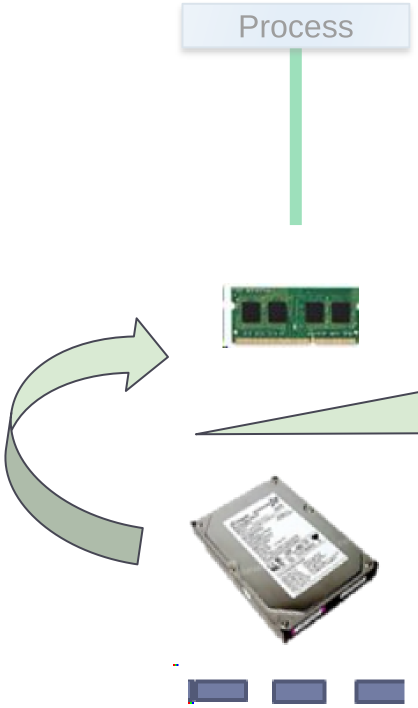


1. Main memory:
   - **NO persistent** data
   - Work with bytes or words
   - Few capacity: only the working set for short period

1. Secondary memory:
   - **Persistent** data
   - Work with **data blocks**
   - Great capacity: all possible data needed in time

Storage System Scope

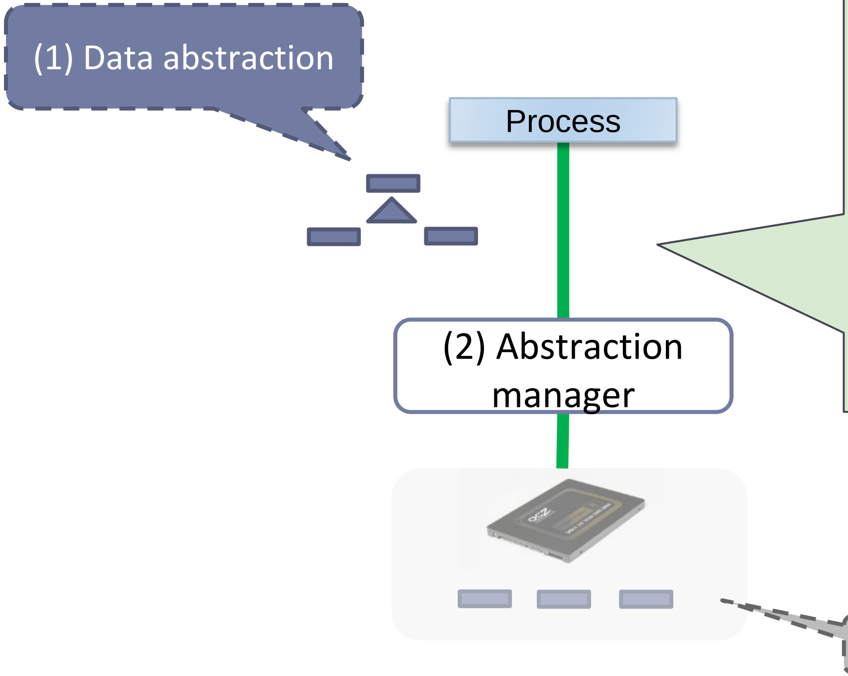
~2020



- It is necessary to locate in which block the data are, load/save those blocks, etc.
- Goal: avoid (if possible) programmers have to deal with block management for search, save, etc.

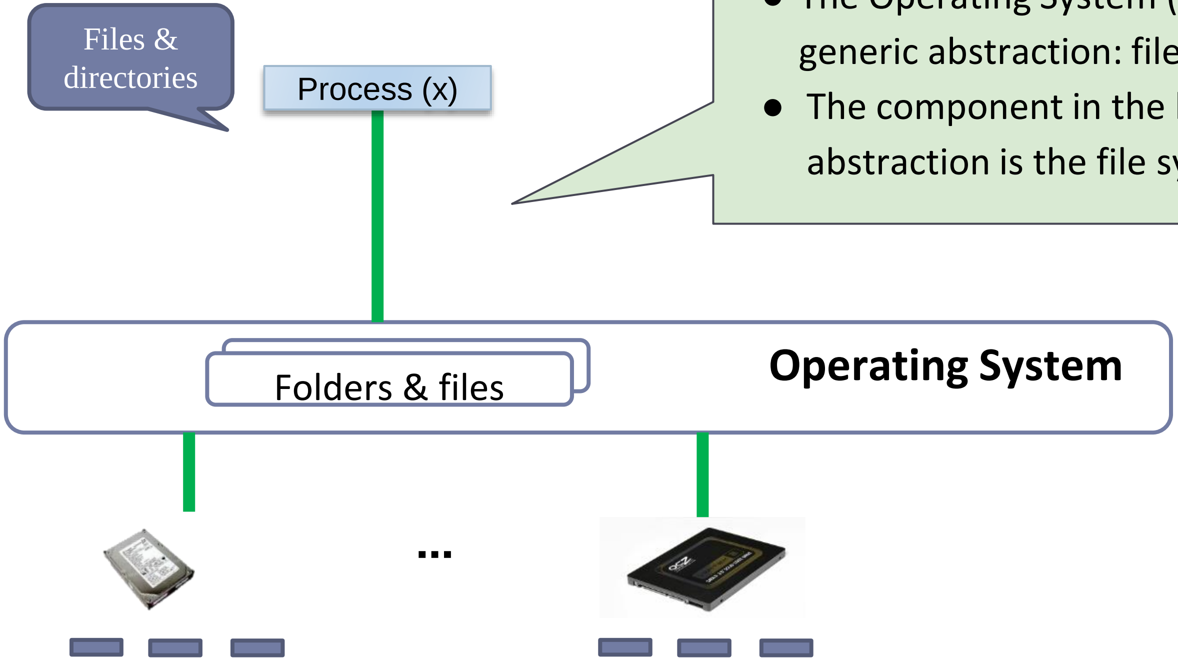
Storage System Scope

~2020



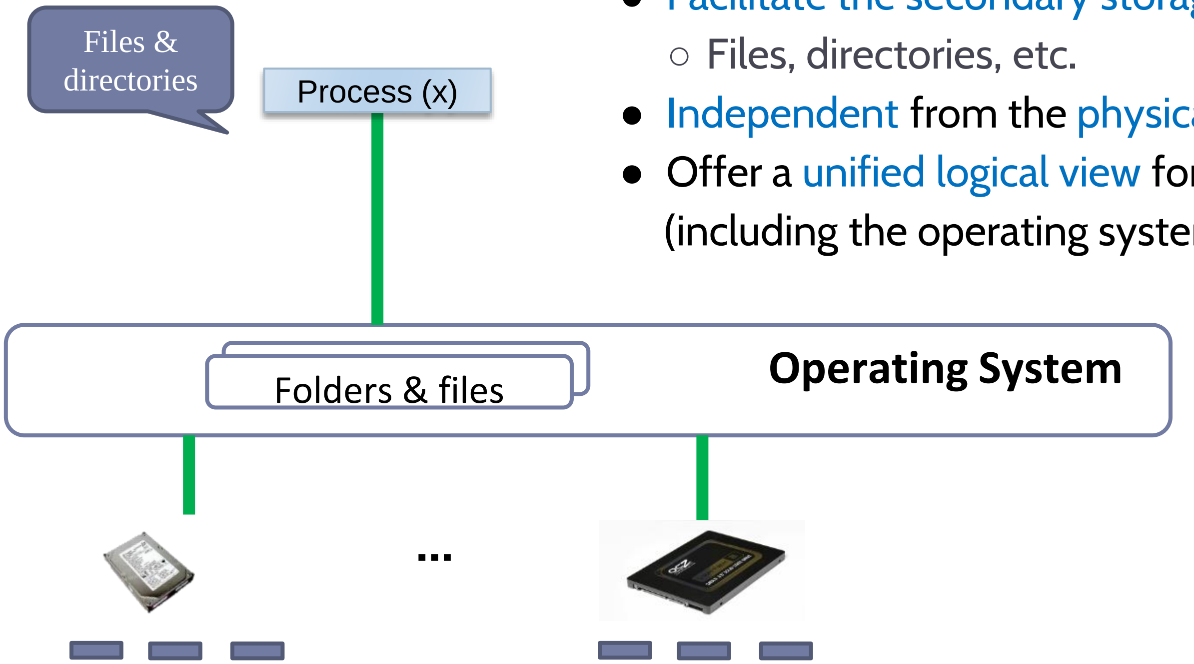
- To avoid programmers deal with block details, it is proposed an intermediated high-level abstraction.
- Main goals:
  - To be device-independent.
  - To offer an unified logic vision.
  - To Balance simplicity and capacity.

(1/2) The O.S. includes a basic and generic abstraction: file system



- The Operating System (O.S.) already includes a basic generic abstraction: files and directories.
- The component in the kernel that manages this abstraction is the file system

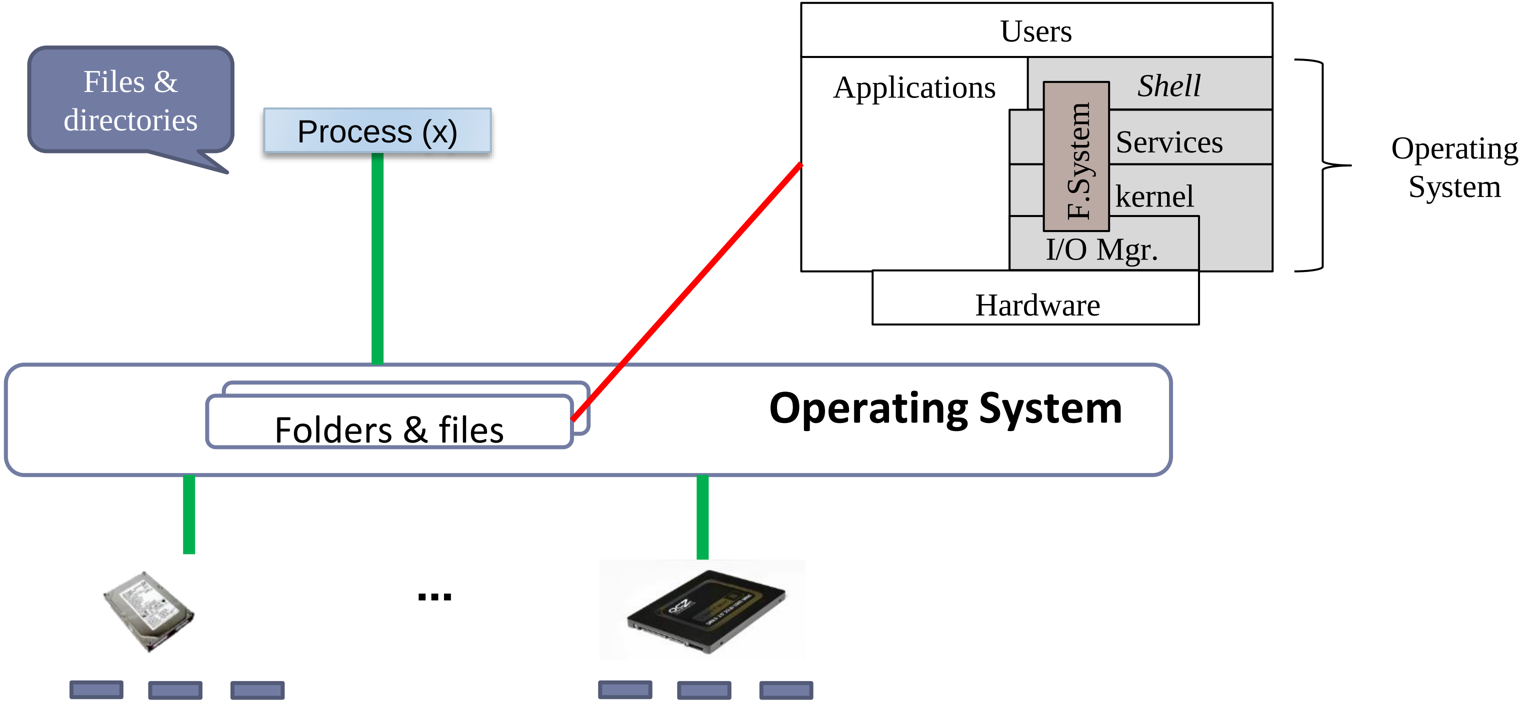
File system Characteristics



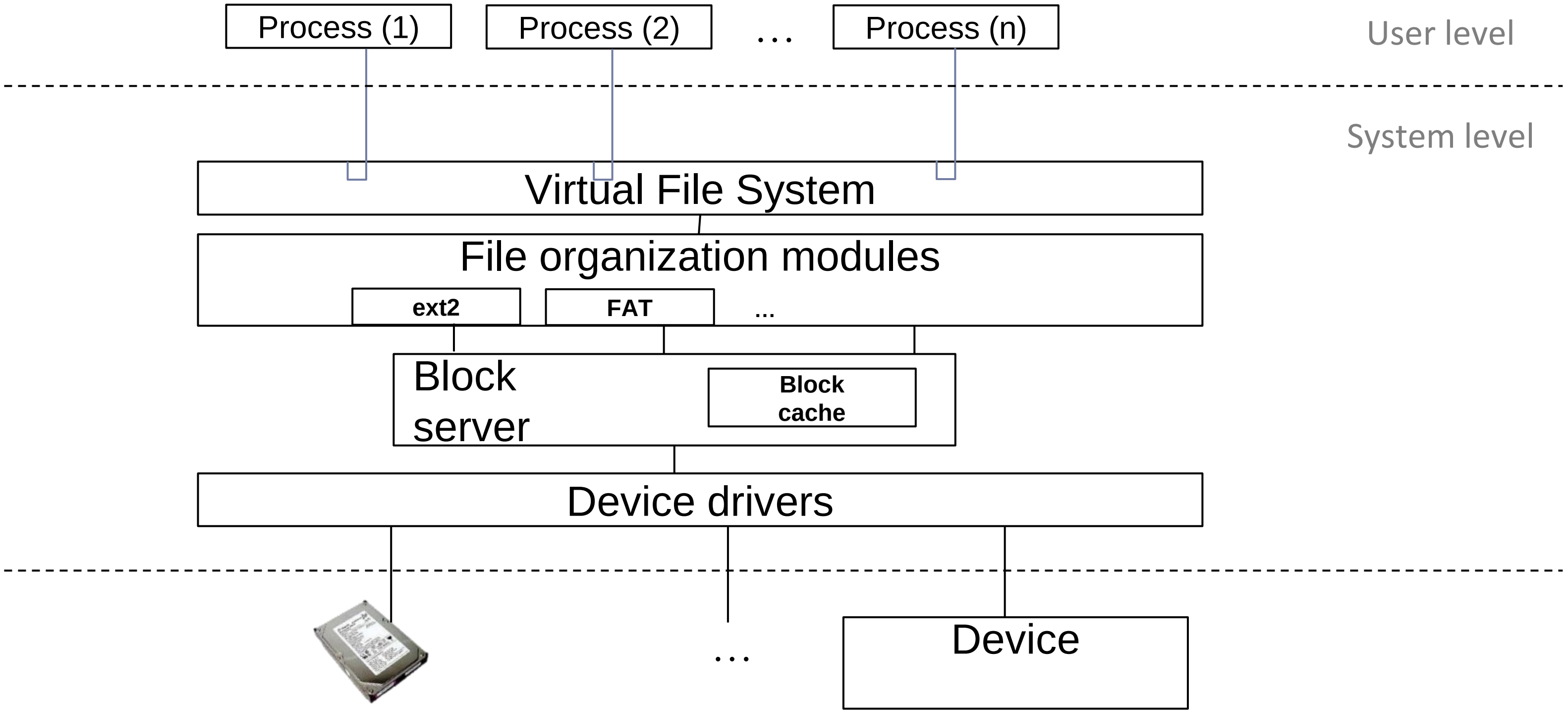
- Facilitate the secondary storage management.
  - Files, directories, etc.
- Independent from the physical device.
- Offer a unified logical view for users and applications (including the operating system itself).

File system Characteristics

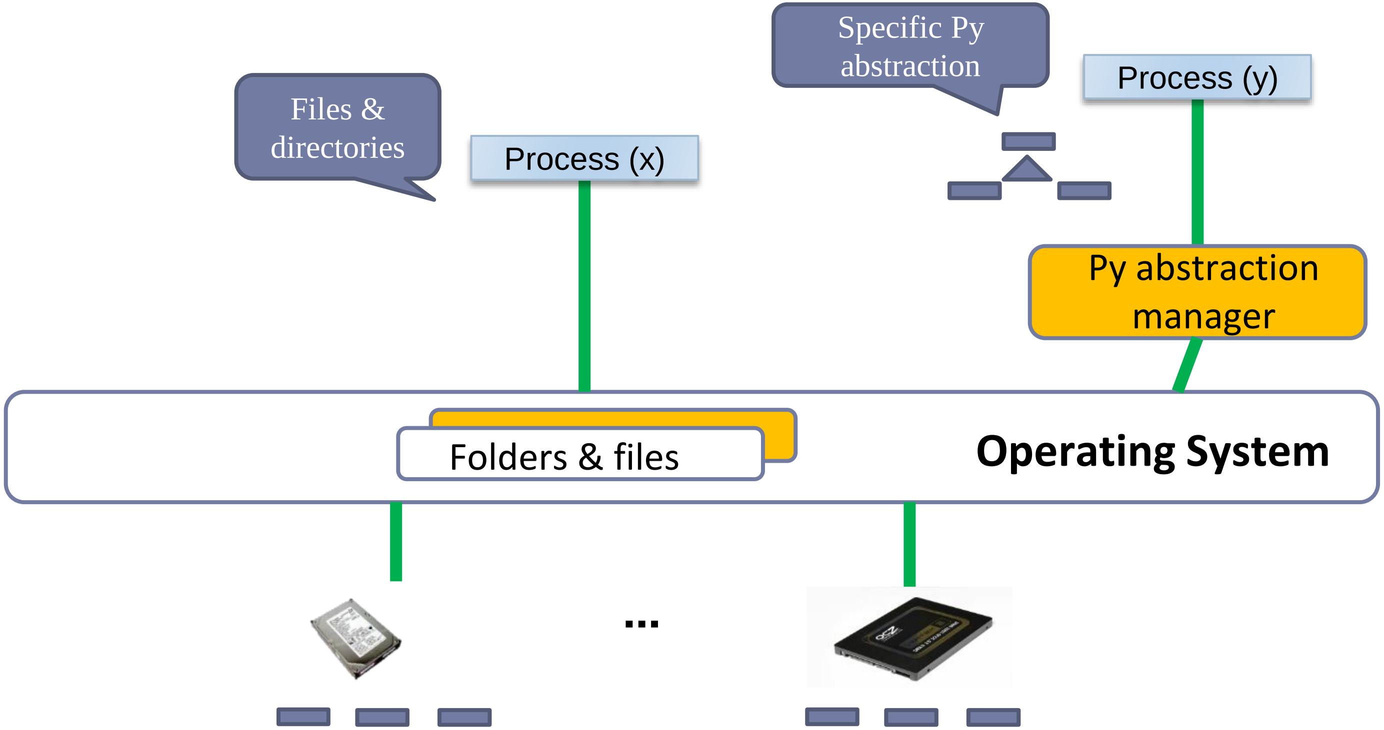
It is transversal to the components of the operating system



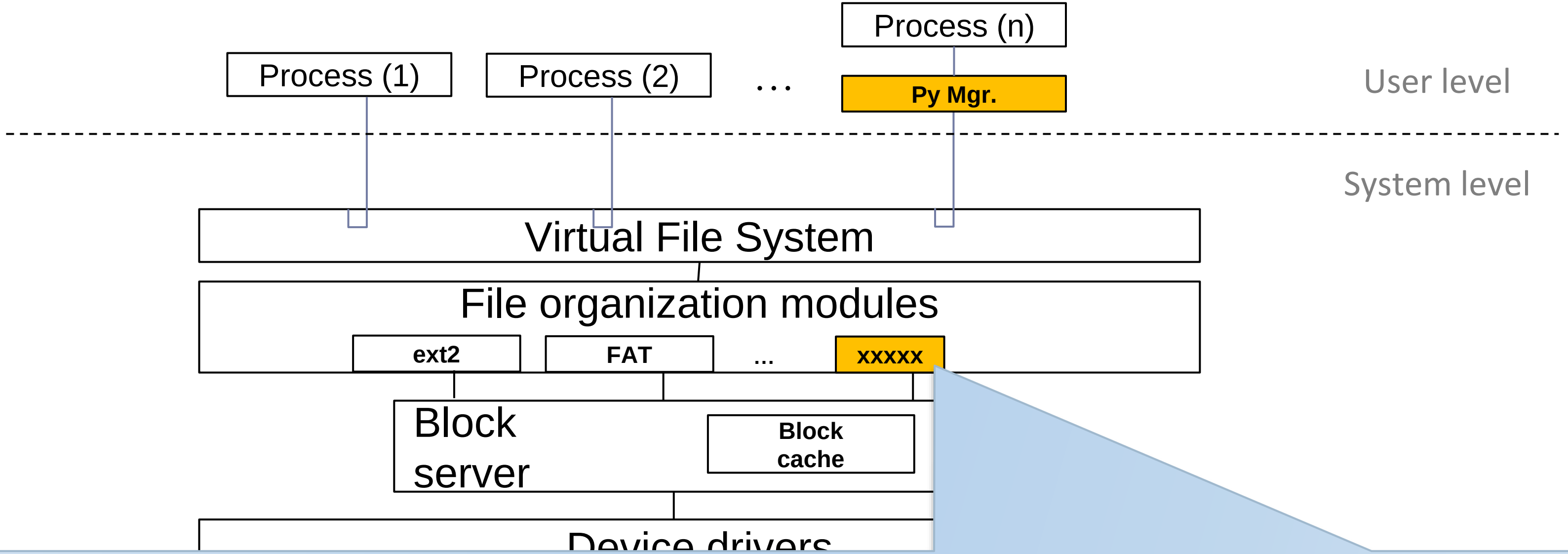
File system Architecture



(2/2) The operating system supports the addition of other abstractions (& mgr.)

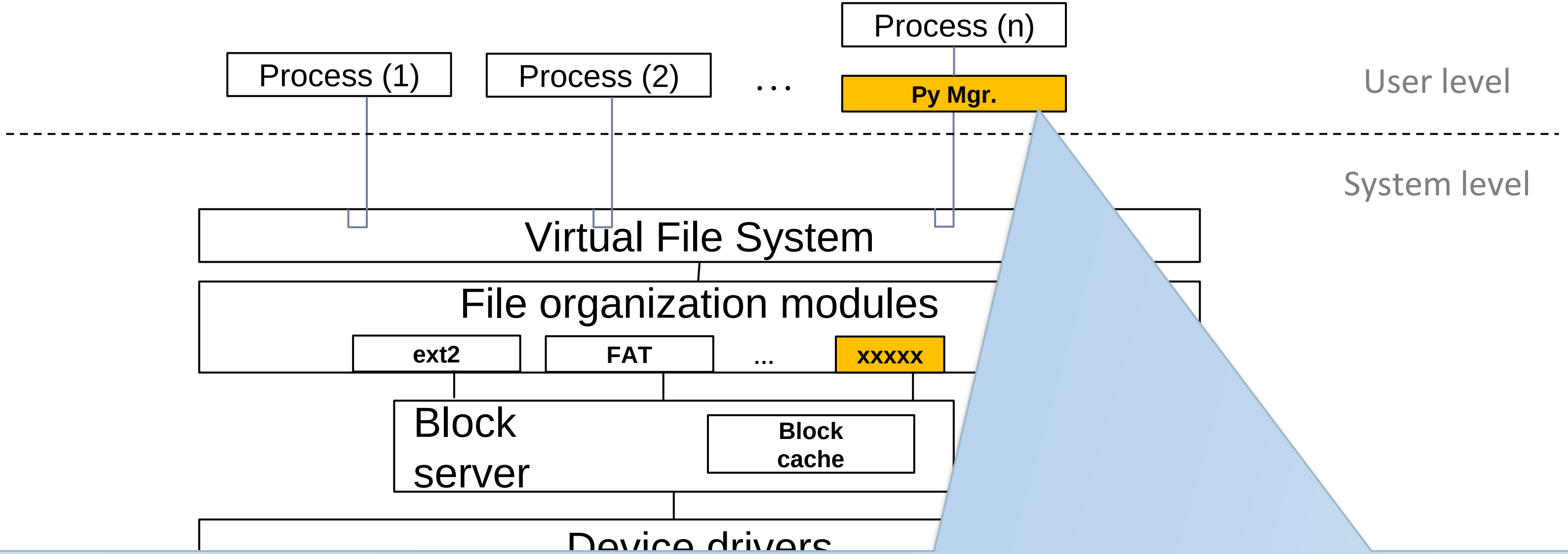


File system Architecture



- A new file system implementation could be added.

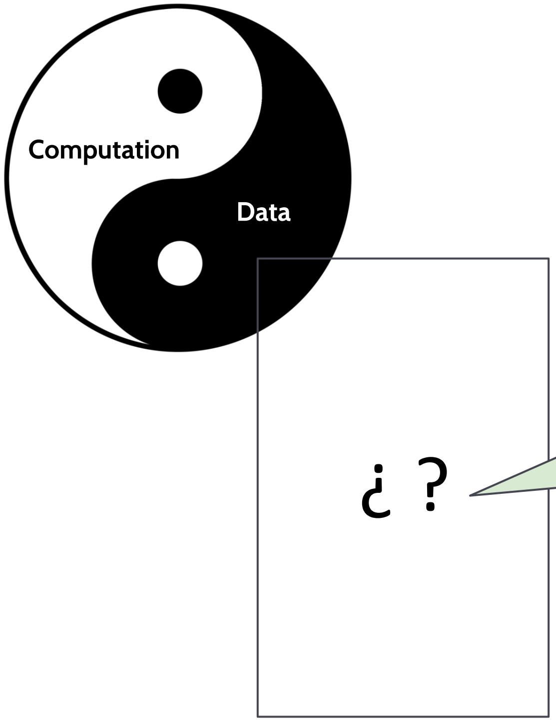
File system Architecture



- Other abstract representations could be implemented using the existing services on the Operating Systems (e.g.: database, nosql database, etc.)

Storage System Scope

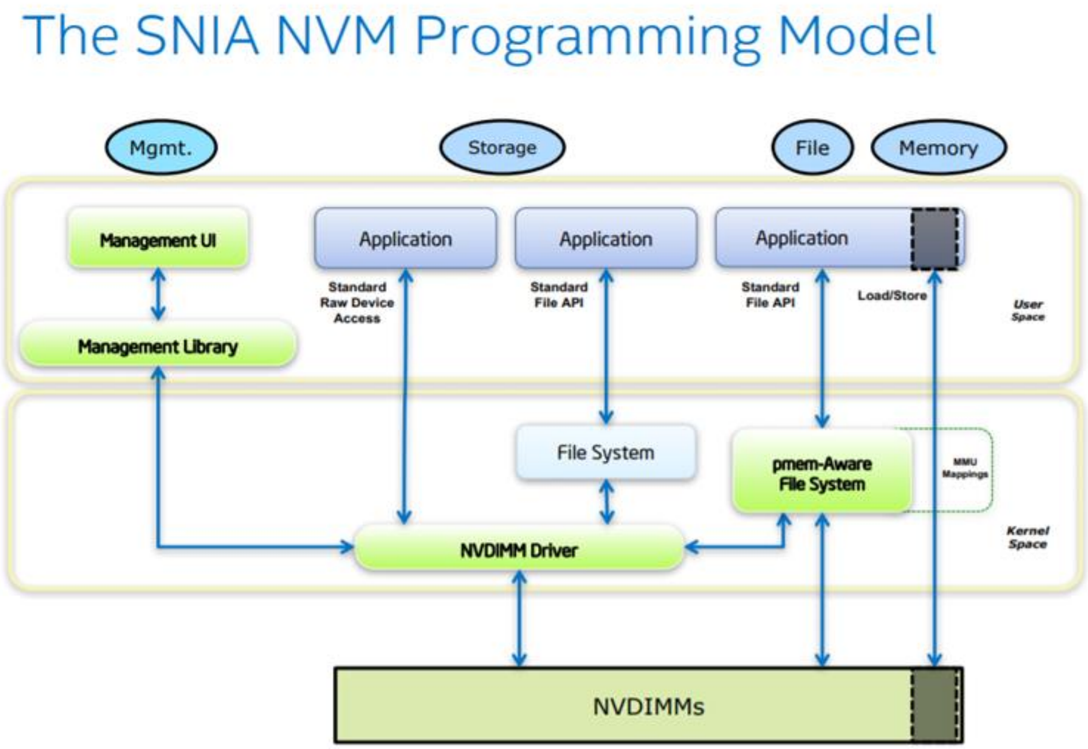
>> 2020



1. New memory (misc of main and secondary):
   - **Persistent** data
   - Work with bytes or words, and with data blocks
   - Great capacity and speed.

Storage System Scope

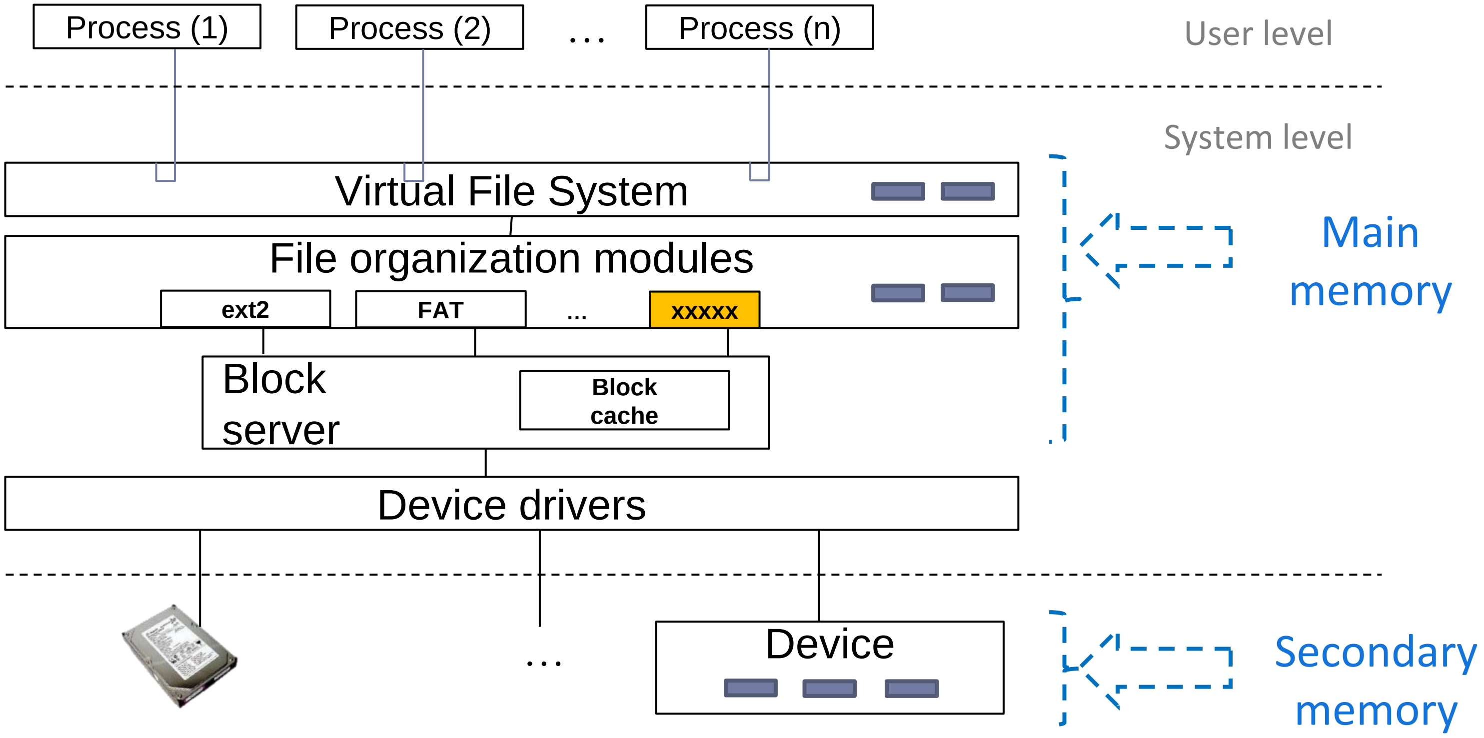
>> 2020



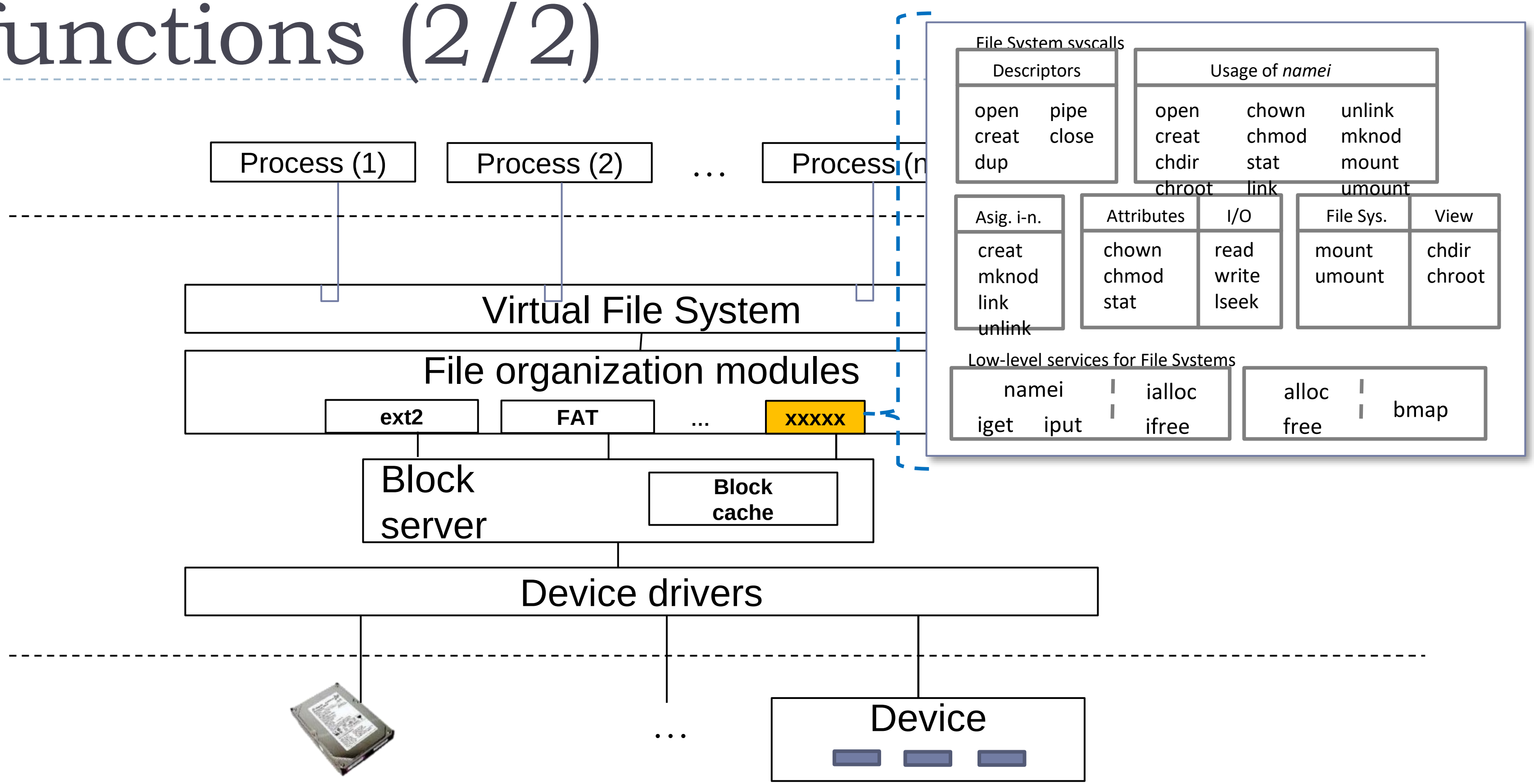
Overview

1. Introduction
2. **File system internals and framework**
3. Design and development of a file system
4. Complementary aspects

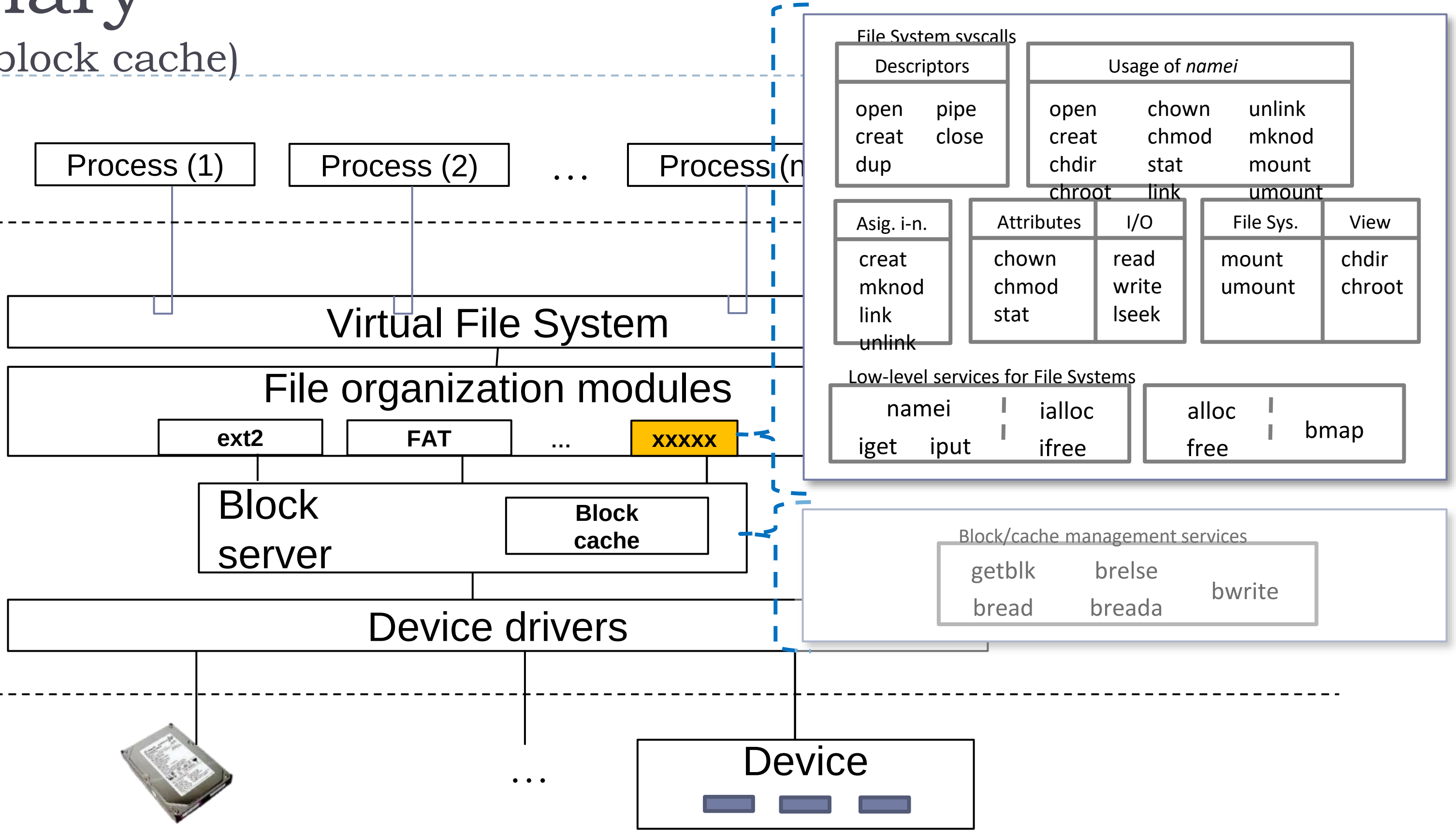
Management data structures (1/2)



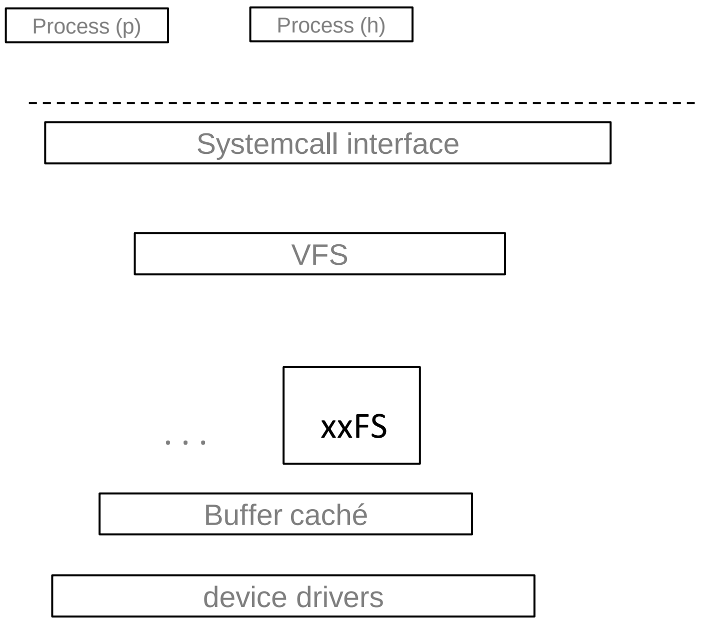
Management functions (2/2)



Summary
(including block cache)



File system organization
main aspects: Linux

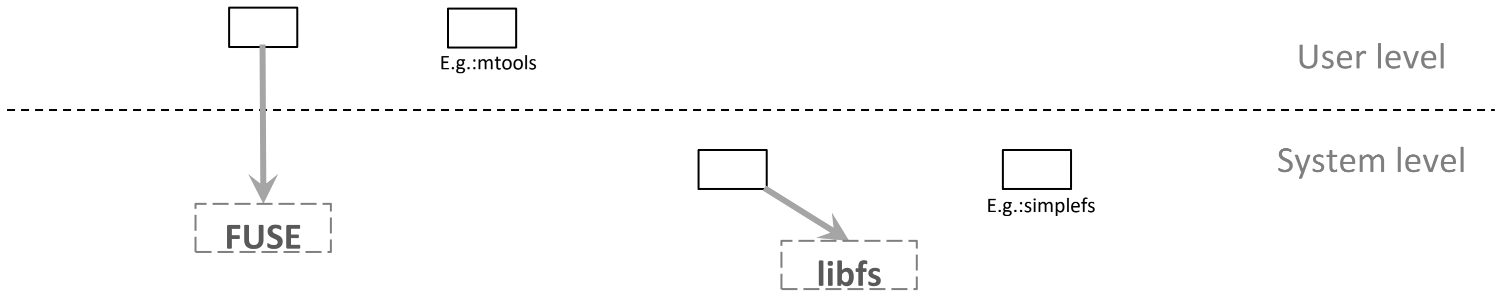


- Layered structure like UNIX.
- Main components:
  - System call interface
  - VFS: *Virtual File System*
  - xxFS: specific file system
  - Buffer caché: block cache
  - device drivers: *drivers*

http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/kernextc/virtual_fsys_over.htm#HDRA29C010F11

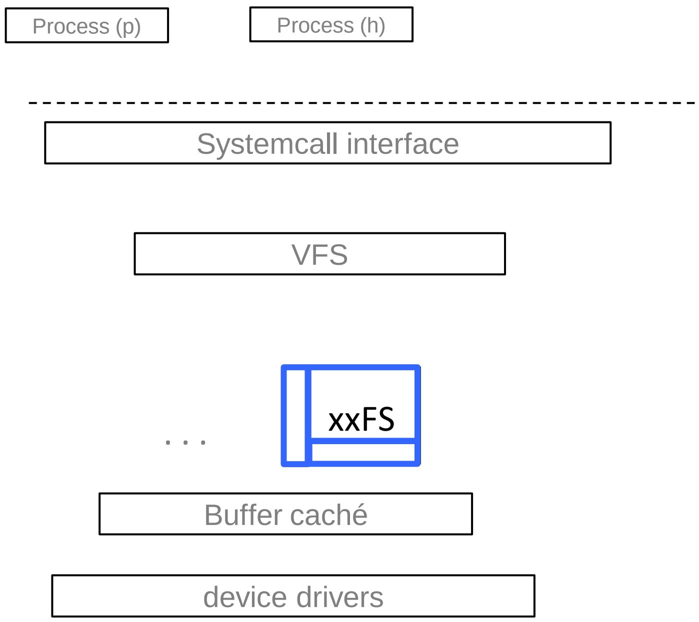
Main options (in Linux) for working in a new the file system

	User space	Kernel space
With *Framework*	FUSE	libfs
Without *Framework*	E.g.: mtools	E.g.: simplefs



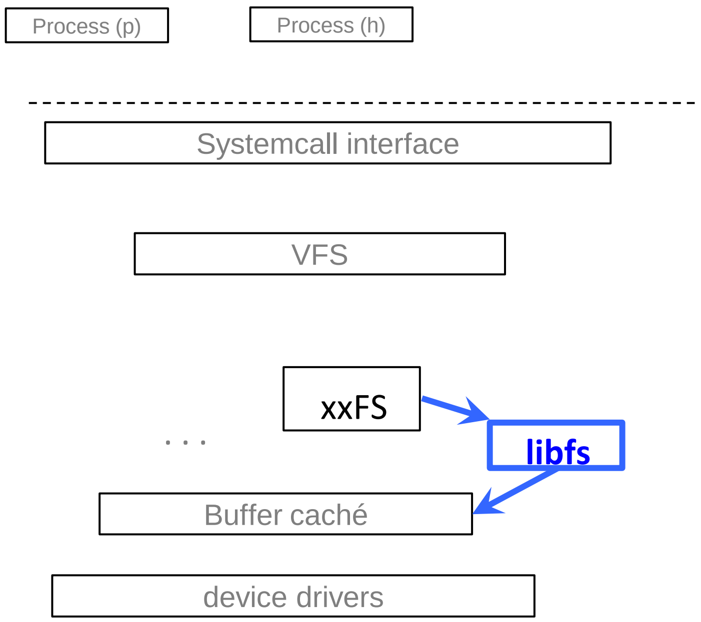
File system organization

without *framework*, within kernel. E.g.: simplefs



- Interface:
  - **register**: to register the file system
  - **...**
  - **open**: to open a work session
  - **read**: read data
  - **...**
  - **namei**: convert from path to i-node
  - **iget**: read a i-node
  - **bmap**: compute an associated offset block
  - **...**

File system organization
with *framework*, within kernel: libfs

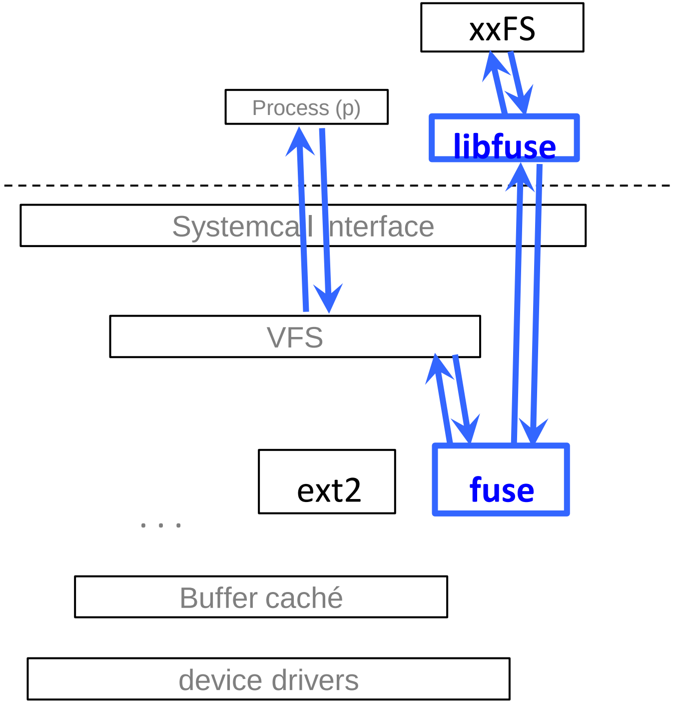


- Interface: libfs
  - **lfs_fill_super**: superblock
  - **lfs_create_file**: file creation
  - **lfs_make_inode**: default i-node
  - **lfs_open**: open a work session
  - **lfs_read_file**: read from file
  - **lfs_write_file**: write to file
  - ...

http://lwn.net/Articles/13325/

File system organization

with *framework*, user space: fuse



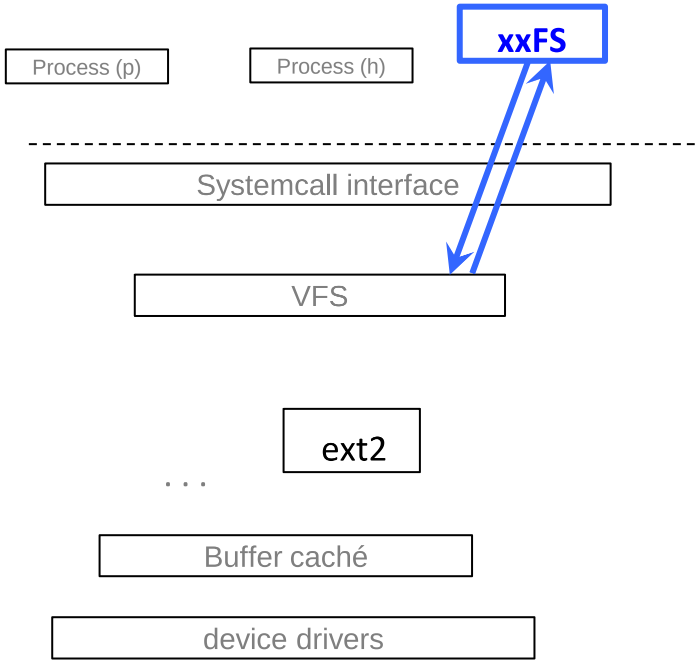
- Interface:
  File system in USer spacE

```
struct fuse_operations {
   ...
   int (*open) (const char *, struct fuse_file_info *);
   int (*read) (const char *, char *, size_t, off_t, struct
fuse_file_info *);
   int (*write) (const char *, const char *, size_t,
off_t,struct fuse_file_info *);
   int (*statfs) (const char *, struct statfs *);
   int (*flush) (const char *, struct fuse_file_info *);
   ...
};
```


http://www.ibm.com/developerworks/linux/library/l-fuse/

File system organization

without *framework*, user space. E.g.: mtools

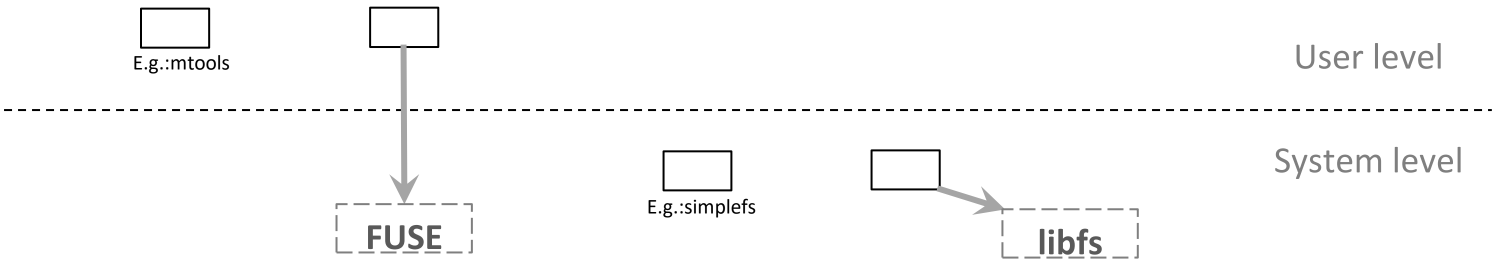


- To implement the file system interface in user space, and as library for other applications:
  - **open**: to open a work session
  - **read**: to read data
  - …
  - **namei**: to convert path into i-node
  - **iget**: read i-node
  - **bmap**: compute the associate block for a given offset
  - …

http://www.gnu.org/software/mtools/manual/mtools.html

summary:
Main options for the file system organization

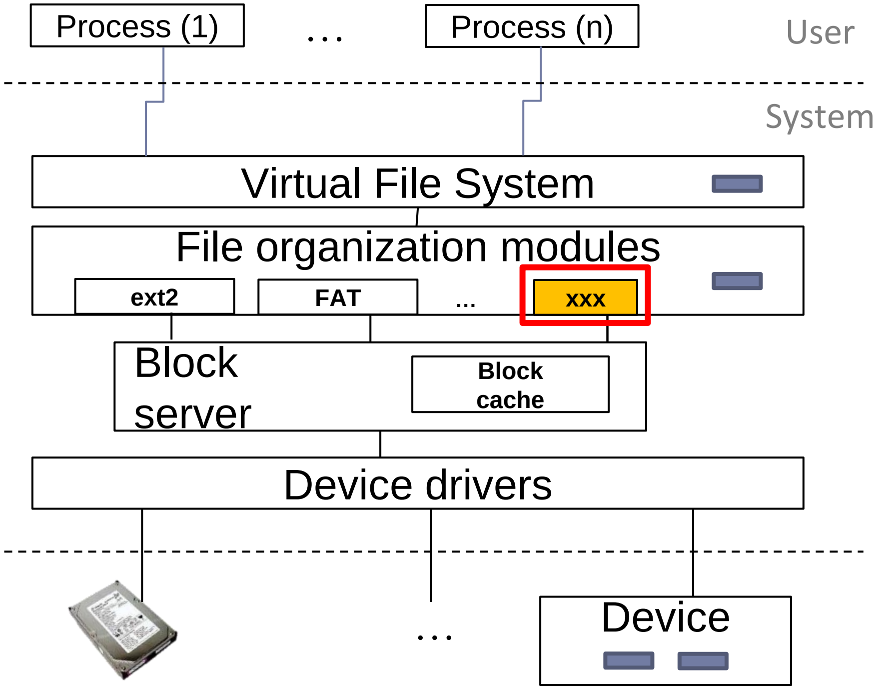
	User space	Kernel space
With *Framework*	FUSE	libfs
Without *Framework*	E.g.: mtools	E.g.: simplefs



Overview

1. Introduction
2. File system internals and framework
3. **Design and development of a file system**
4. Complementary aspects

Design and development of a file system



- **File system requirements**
- Main data structures in the secondary memory
- Main data structures in the main memory
- Block management
- Internal (and service) functions

Main requirements
e.g.: Unix-like file system

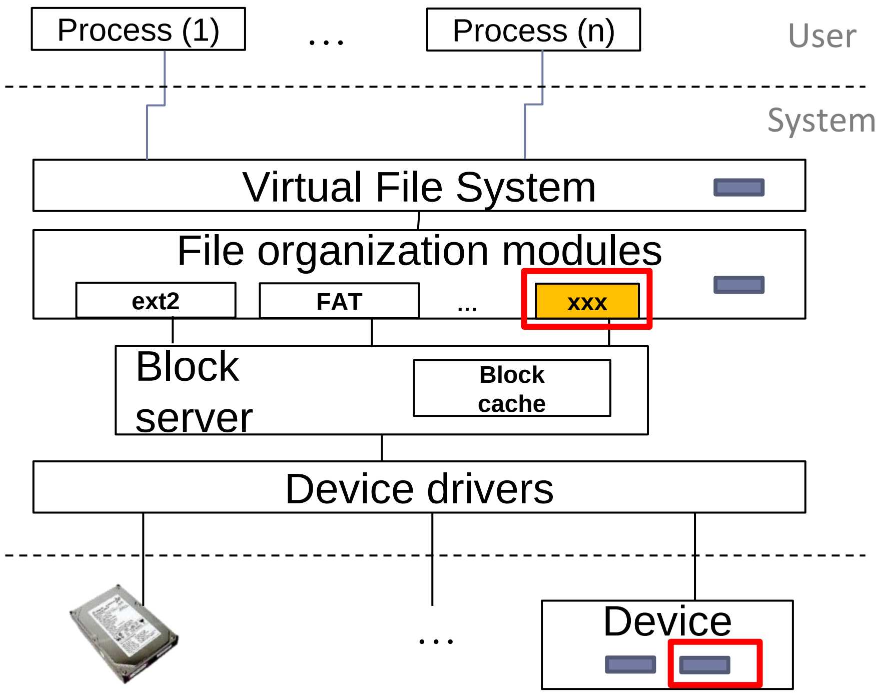
- Processes have to use a secure interface, without direct access to the kernel data structures.
- Share the file offset position among processes from the same parent that open the file.
- Offer functionality for working with a file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file systems registered in the kernel, and keep track of the mount point of these file systems.

Getting the proper storage system for the requirements…

http://en.wikipedia.org/wiki/List_of_file_systems

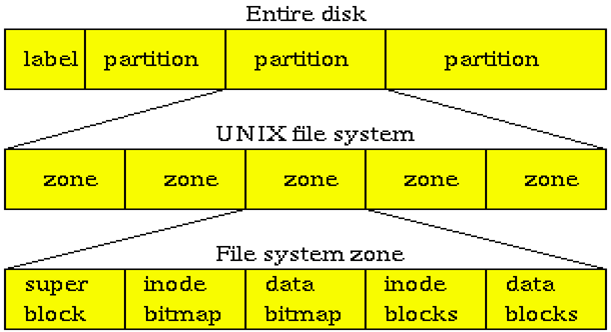
1. To search a file system that satisfies the requirements.
2. To adapt an existing file system in order to satisfy the requirements.
3. To build a file system that satisfies the requirements.

Design and development of a file system



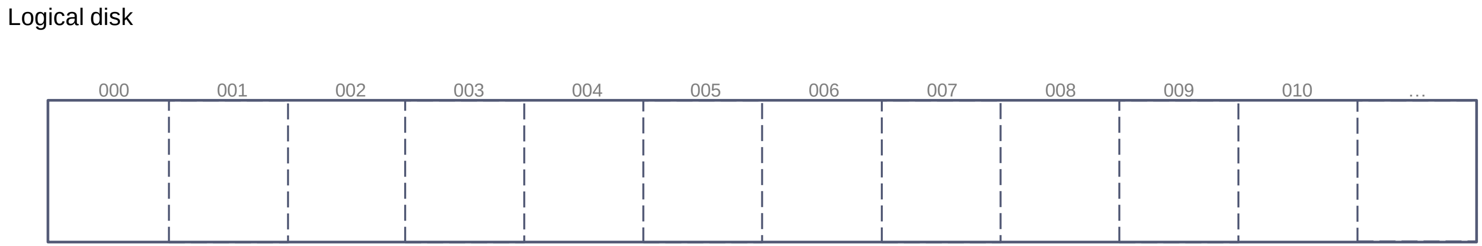
- File system requirements
- **Main data structures in the secondary memory**
- Main data structures in the main memory
- Block management
- Internal (and service) functions

File system Structures

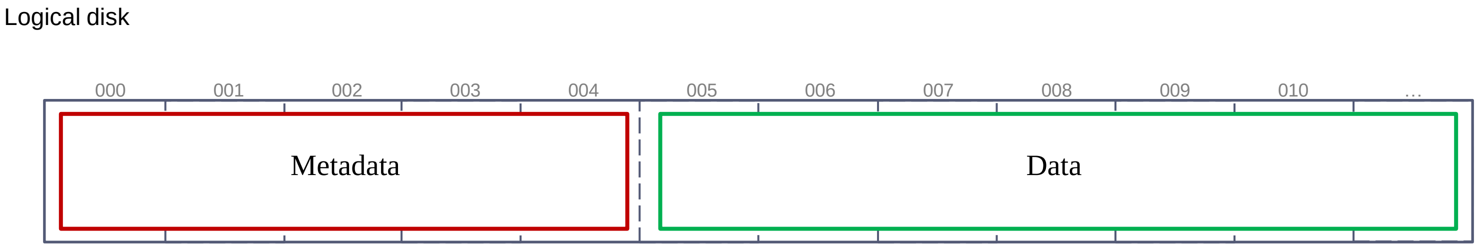


- UNIX/Linux
- FAT

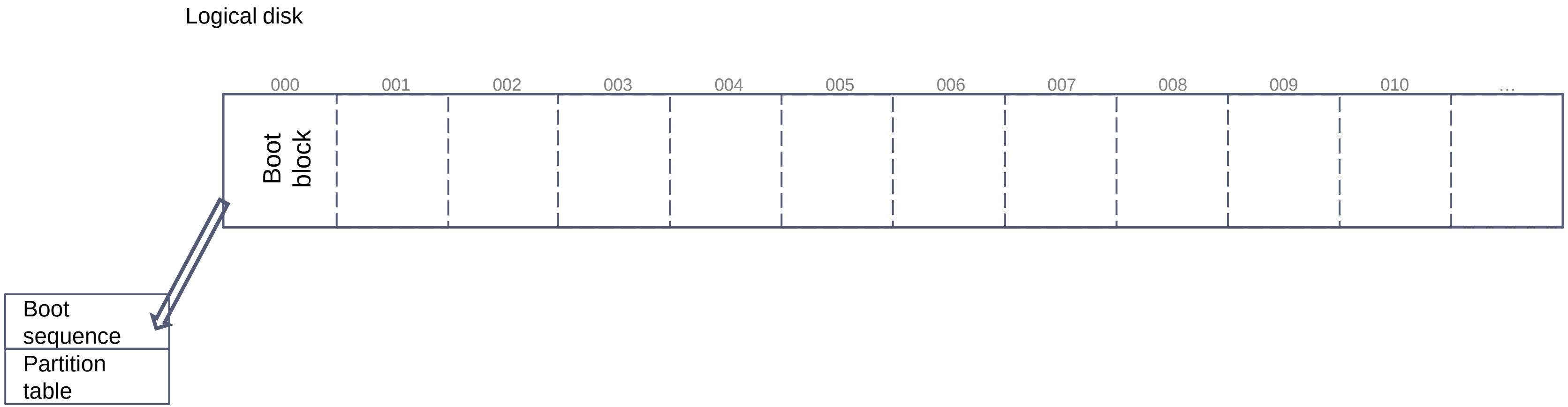
File system:
Unix-like representation



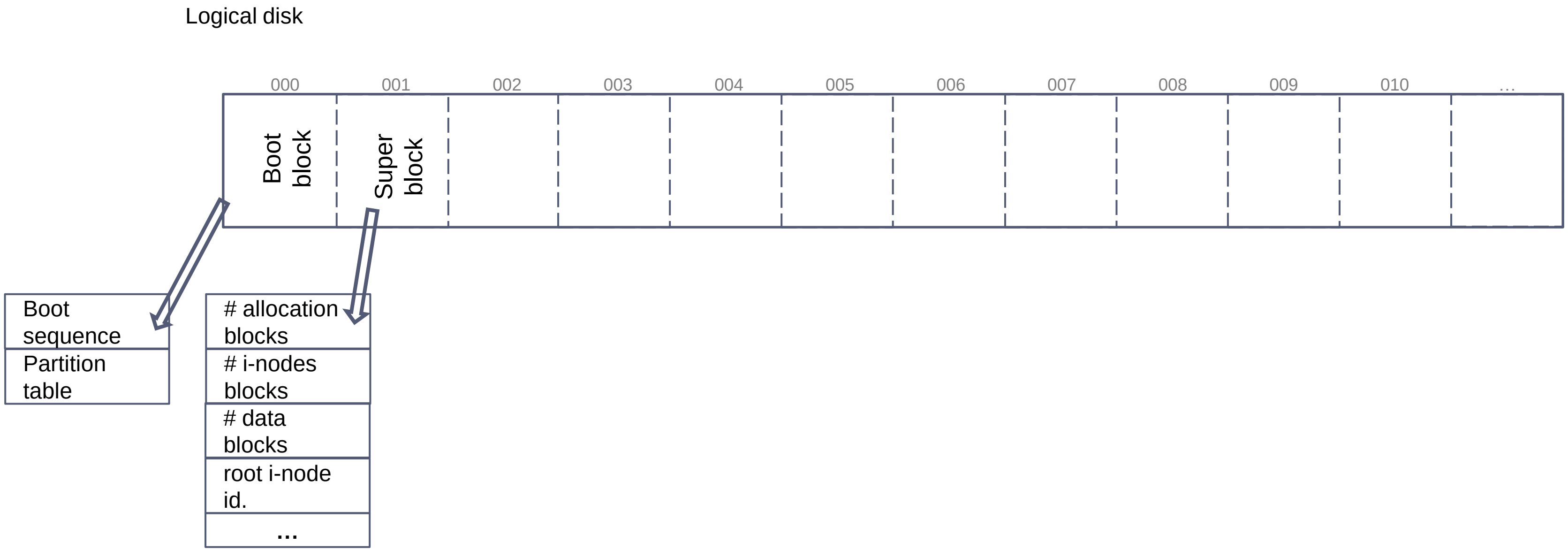
File system:
Unix-like representation



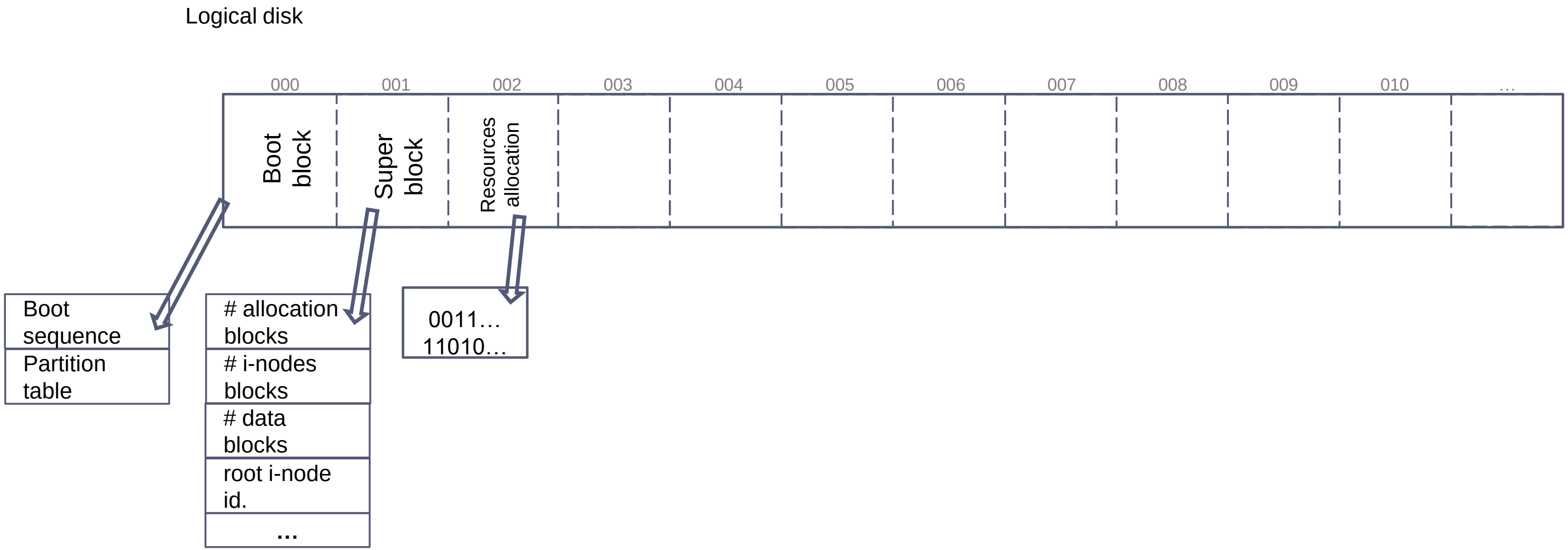
File system:
Unix-like representation



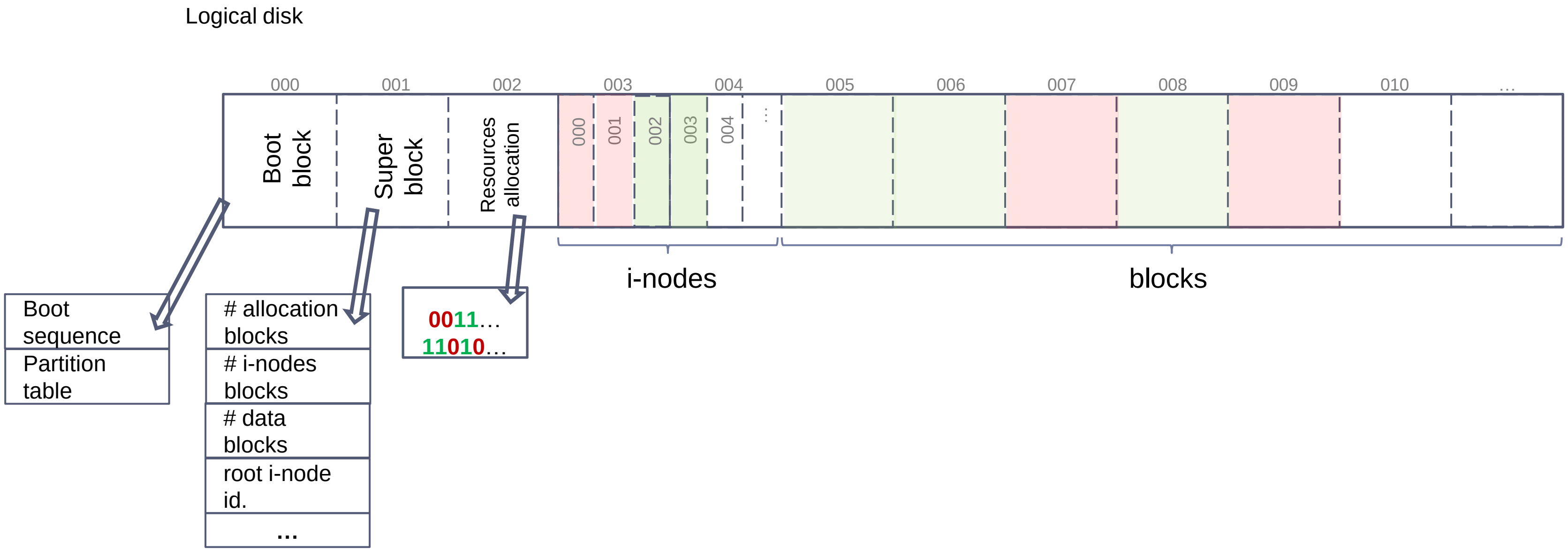
File system:
Unix-like representation



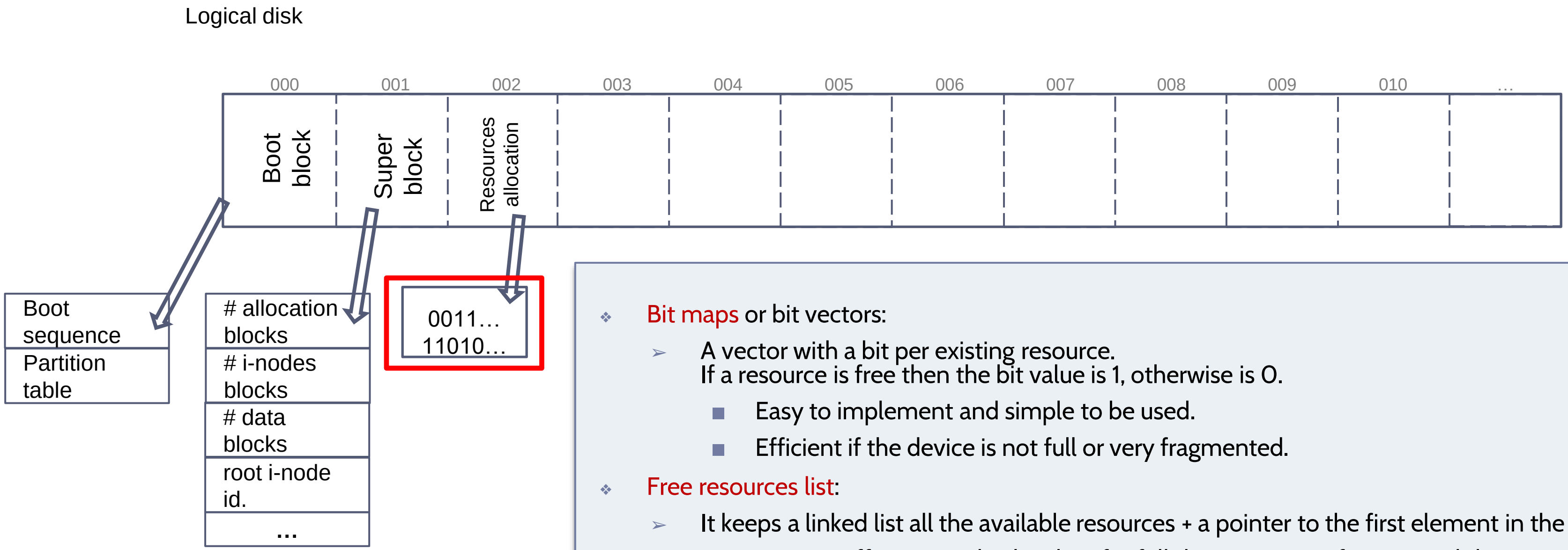
File system:
Unix-like representation



File system:
Unix-like representation

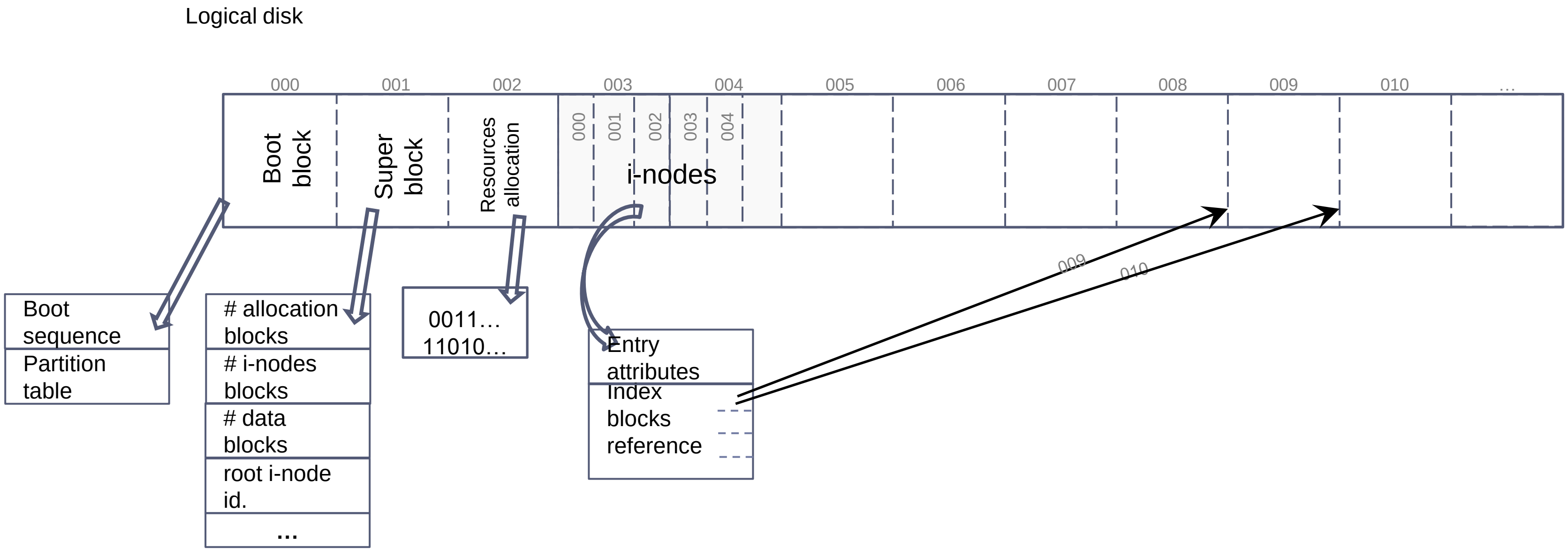


File system:
Unix-like representation

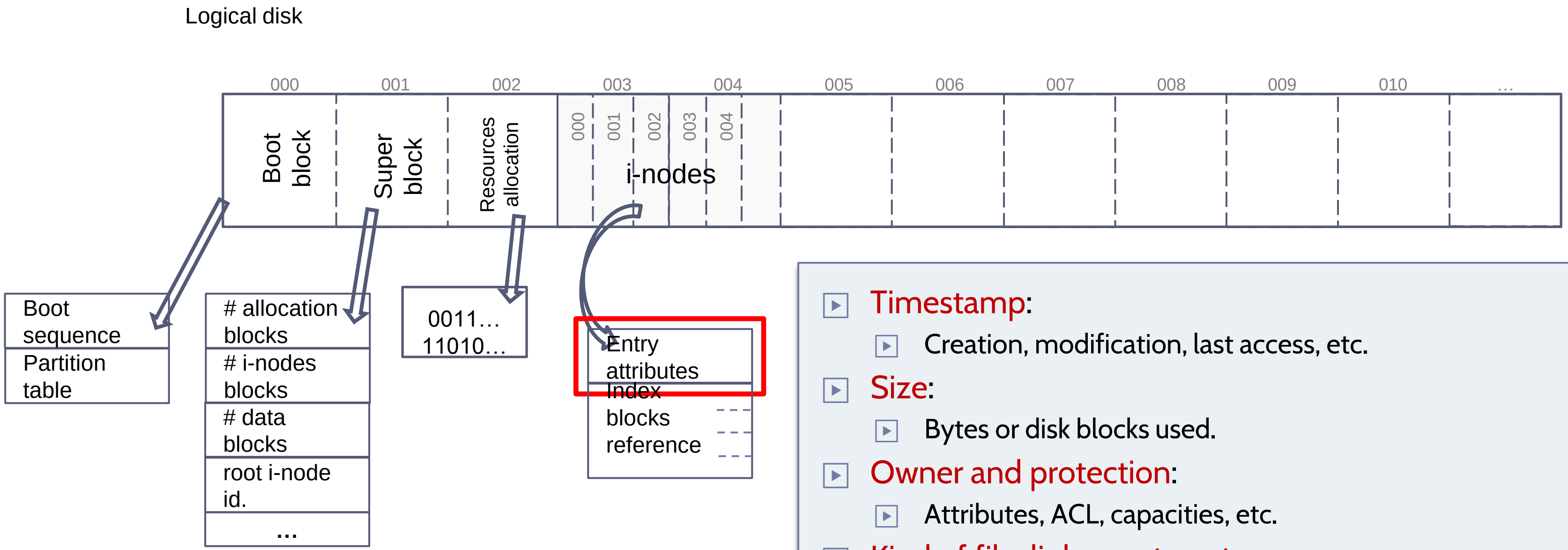


- Bit maps or bit vectors:
  - A vector with a bit per existing resource.
    If a resource is free then the bit value is 1, otherwise is 0.
    - Easy to implement and simple to be used.
    - Efficient if the device is not full or very fragmented.
- Free resources list:
  - It keeps a linked list all the available resources + a pointer to the first element in the list.
    - No very efficient method, unless for full devices or very fragmented devices.
- Indexing:
  - Index table with the identification of the free resources.

File system:
Unix-like representation

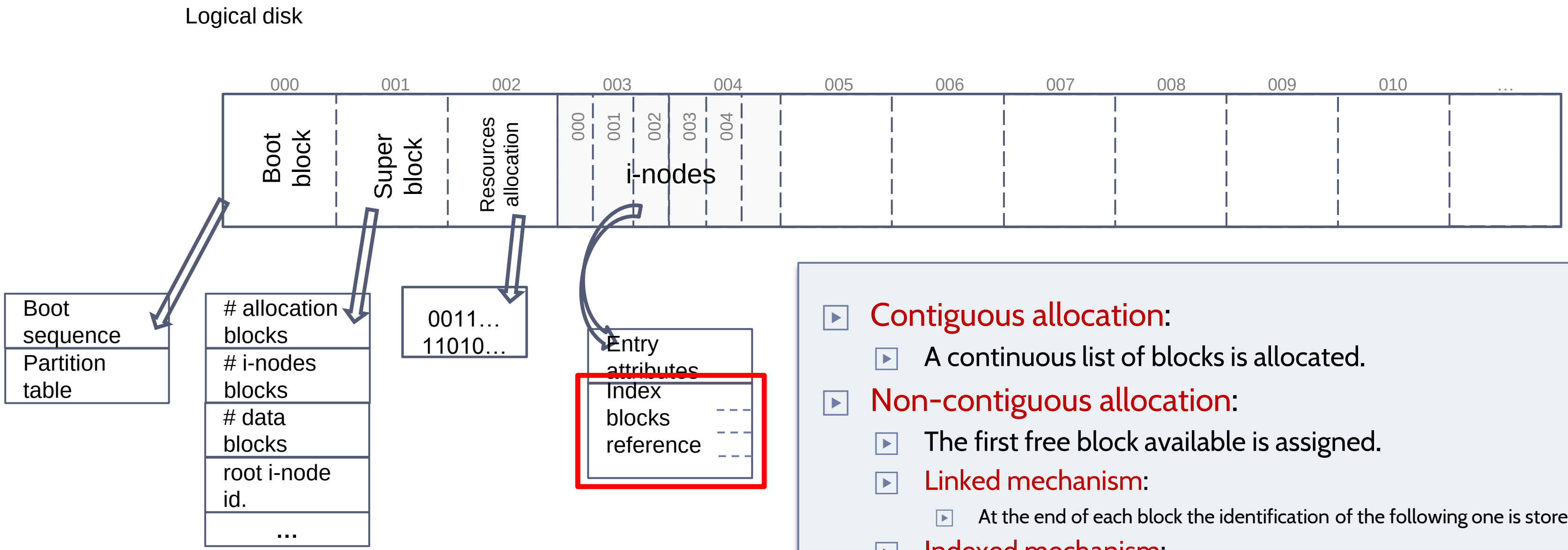


File system:
Unix-like representation



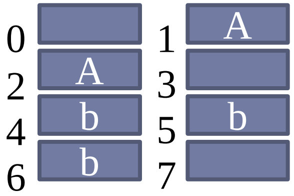
- Timestamp:
  - Creation, modification, last access, etc.
- Size:
  - Bytes or disk blocks used.
- Owner and protection:
  - Attributes, ACL, capacities, etc.
- Kind of file, link counter, etc.

File system:
Unix-like representation



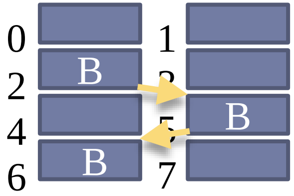
- **Contiguous allocation**:
  - A continuous list of blocks is allocated.
- **Non-contiguous allocation**:
  - The first free block available is assigned.
  - **Linked mechanism**:
    - At the end of each block the identification of the following one is stored.
  - **Indexed mechanism**:
    - Blocks with the references of all file blocks.

File systems:
resources allocation alternatives



F	I	L
A	1	2
B	4	3

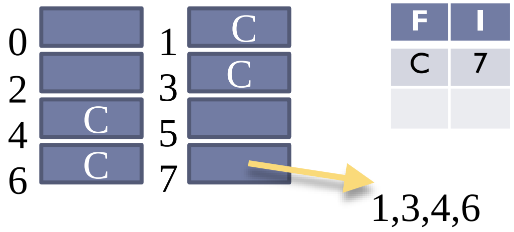
- **Contiguous allocation:**
  - The blocks of the files are contiguous
  - It needs: first (I) and # of blocks (L)
  - (A) Ideal for immutable files



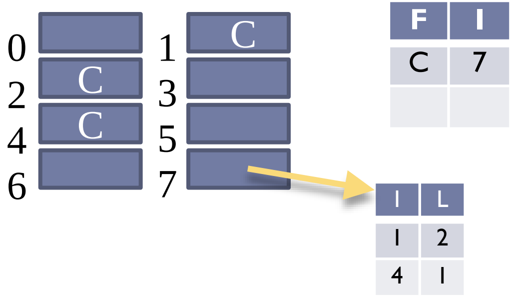
F	I	L
B	2	3

- **Non-contiguous allocation :**
  - Each block has the reference of the following one
  - It needs: first (I) and # of blocks (L)
  - (D) Random access is a little bit hard.

File systems:
resources allocation alternatives

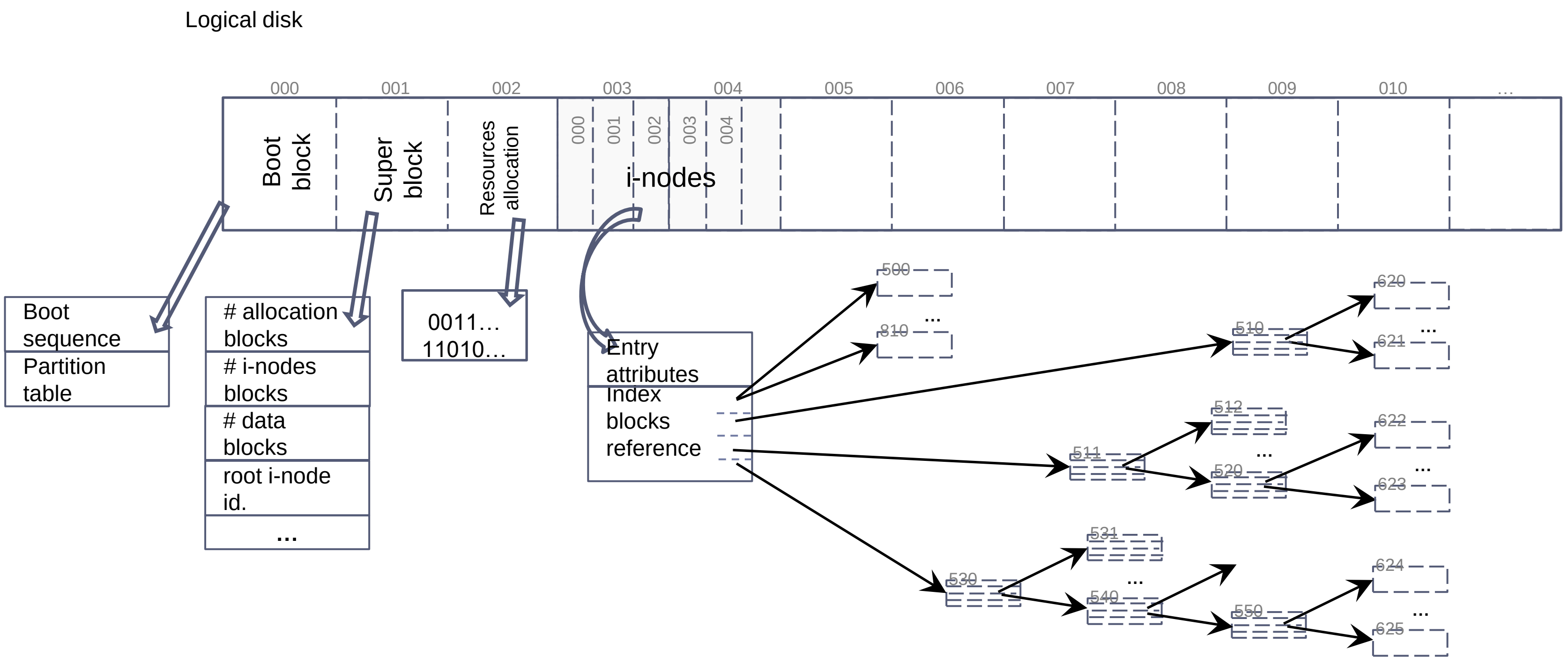


- Indexed allocation (blocks):
  - Some blocks are used to store the reference list of file data blocks.
  - Metadata needed: id. Of the first index block.
  - (D) Fragmentation: need to defrag.



- Indexed allocation (extends):
  - Some blocks are used to store the reference list of continuous file data blocks sequences.
  - Metadata needed: id. of the first index block.
  - (D) Fragmentation: need to defrag.

File system:
Unix-like representation



How elements are represented

- Files

- Directories

- Links

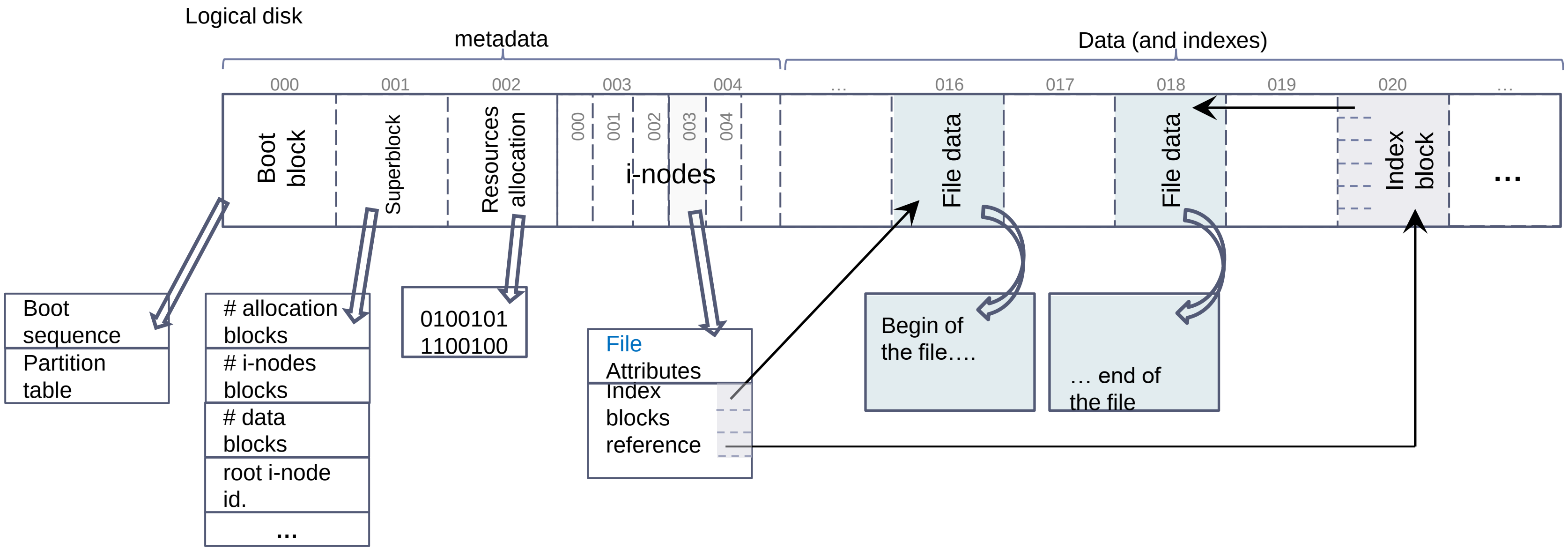
How elements are represented

- Files

- Directories

- Links

File system:
Unix-like representation: files



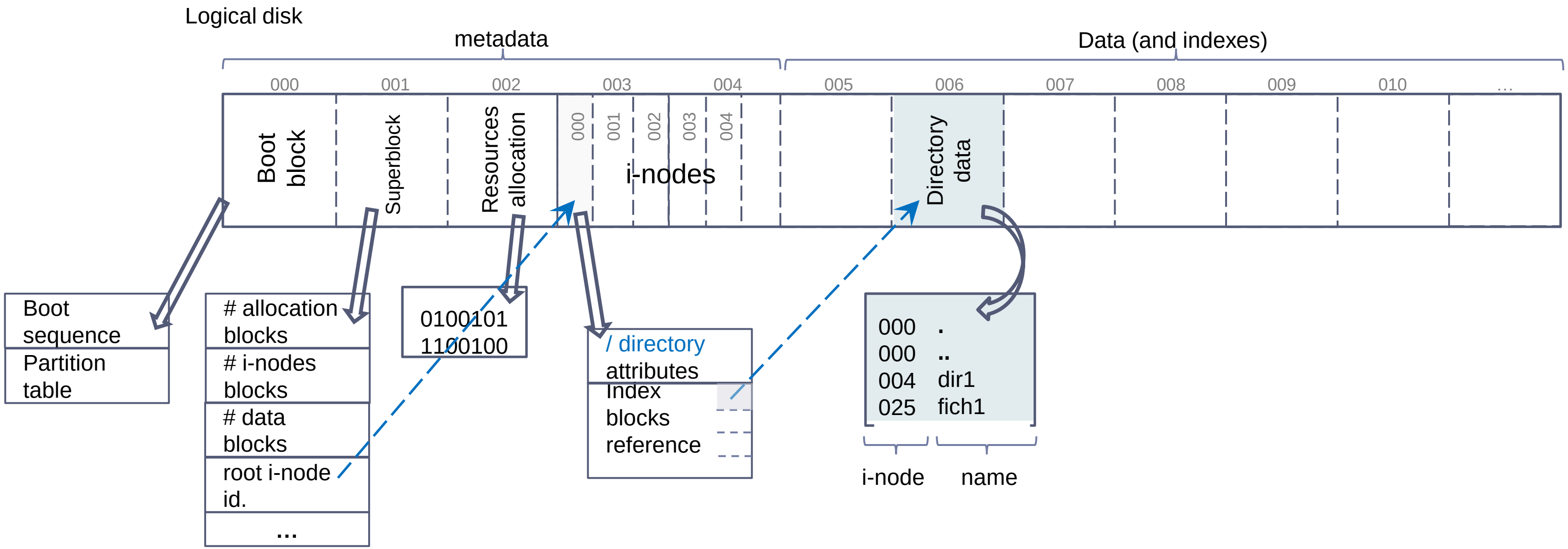
How elements are represented

- Files

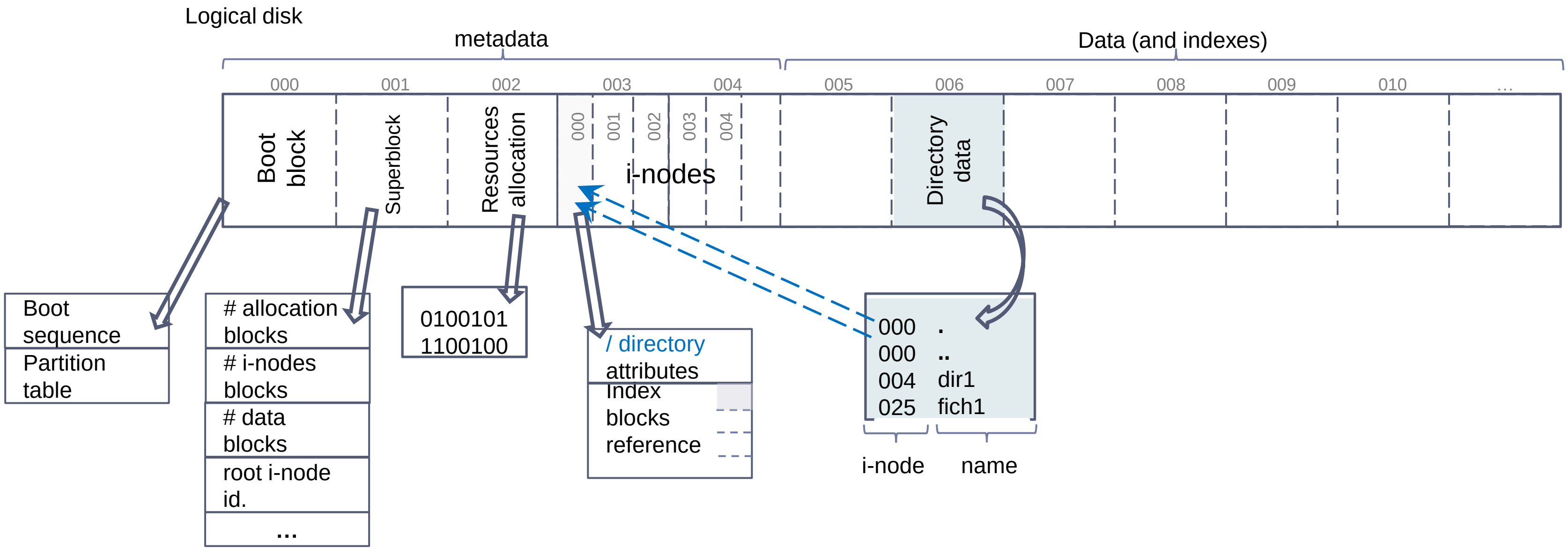
- Directories

- Links

File system:
Unix-like representation: directories

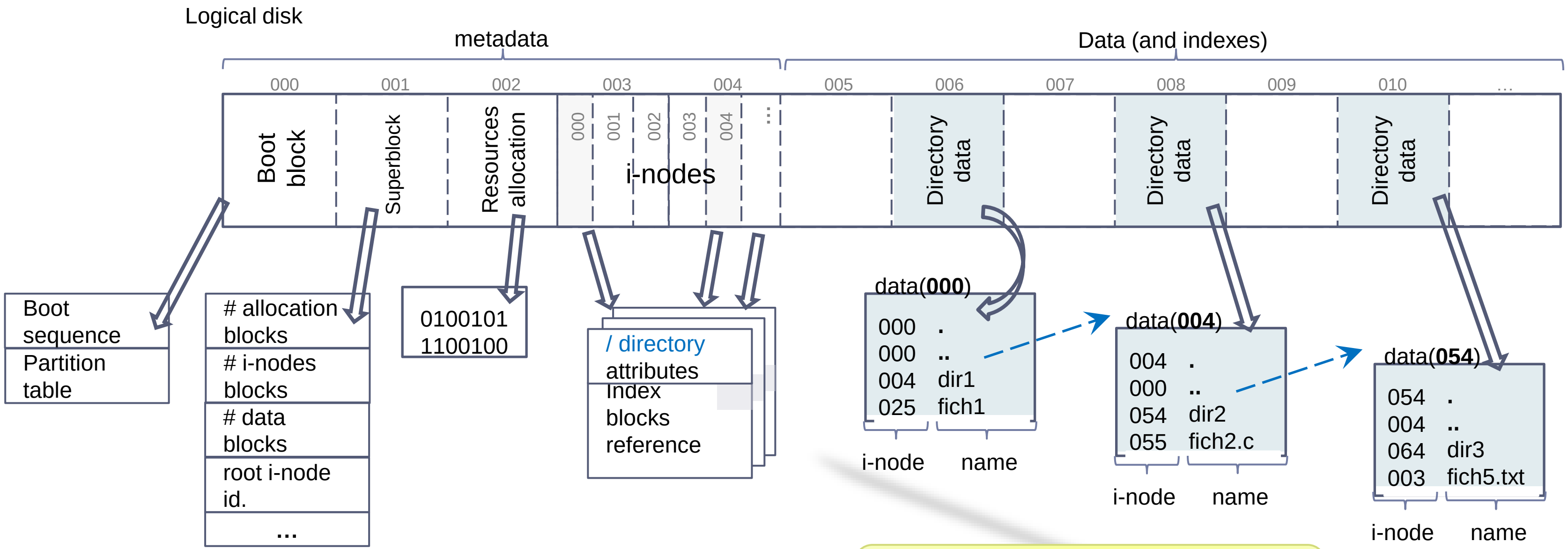


File system:
Unix-like representation: directories



Alejandro Calderón Mateos

File system:
Unix-like representation: directories



ls –l /dir1/dir2/fich5.txt

- / + dir1 + dir2 + fich5.txt
- 4 i-nodes + 3 data blocks

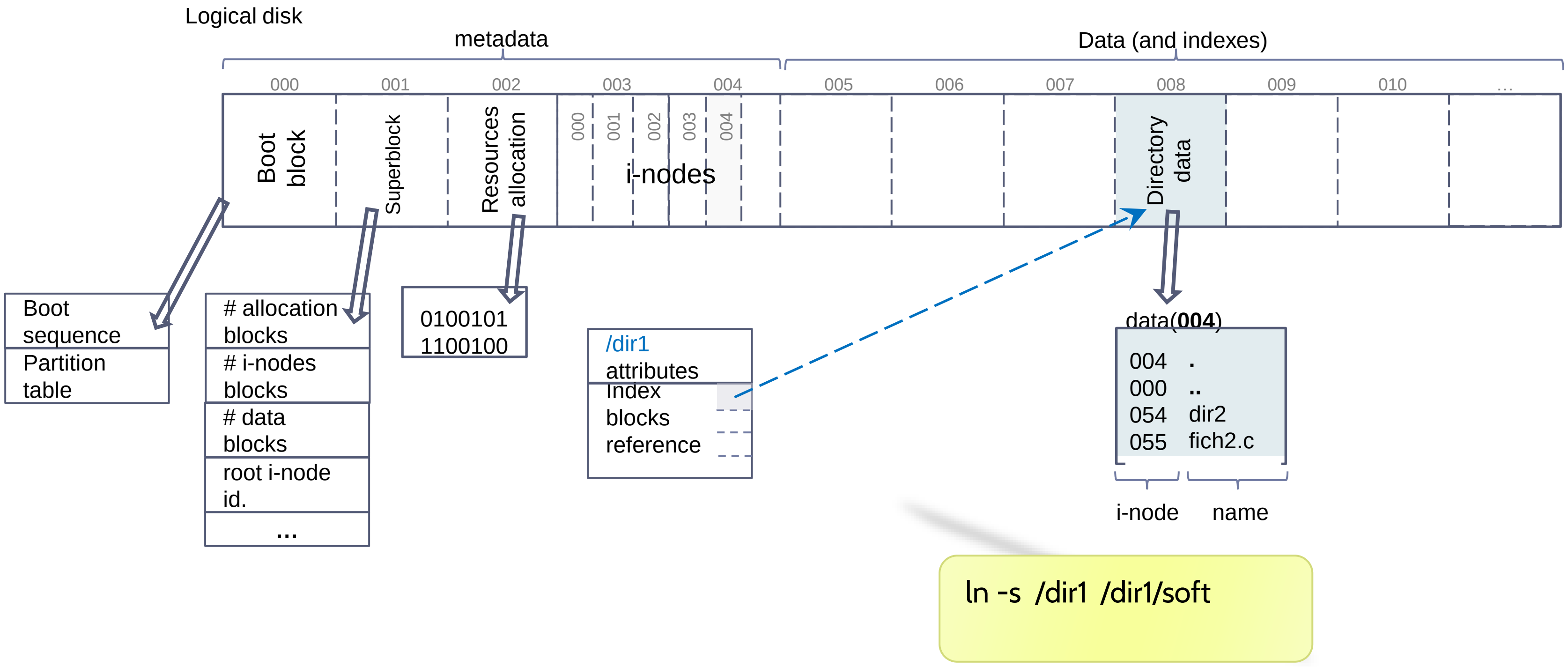
How elements are represented

- Files

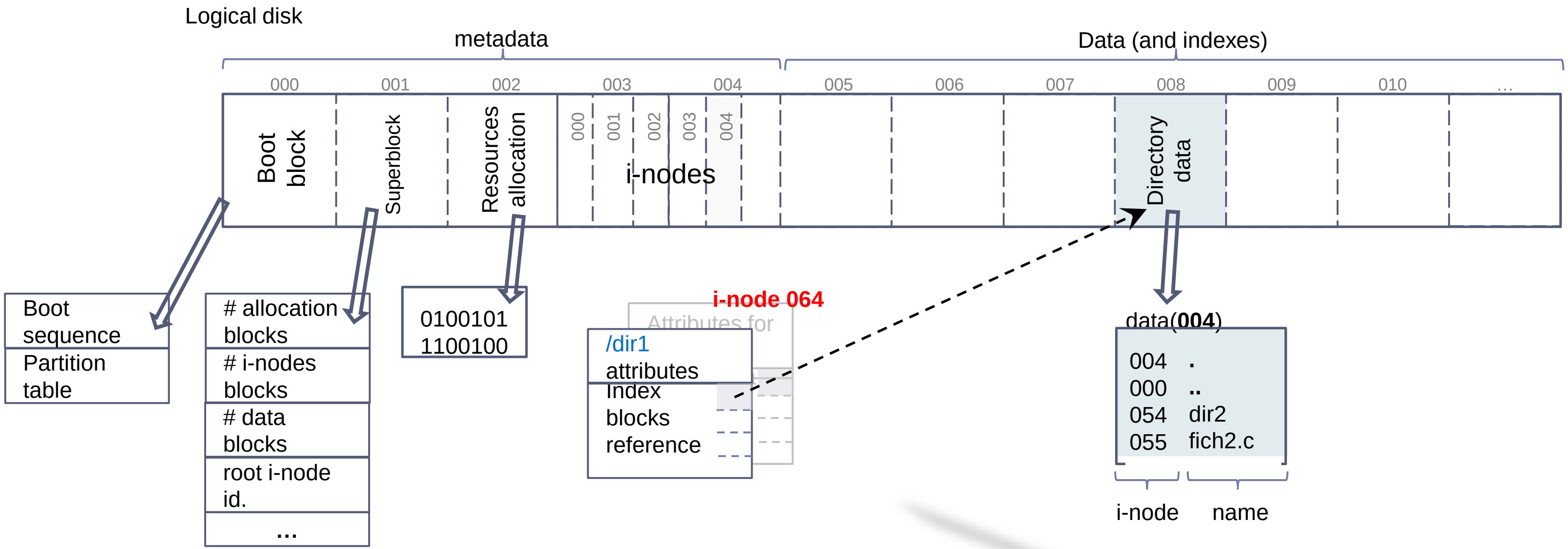
- Directories

- Links

File system:
Unix-like representation: Symbolic link (soft link)

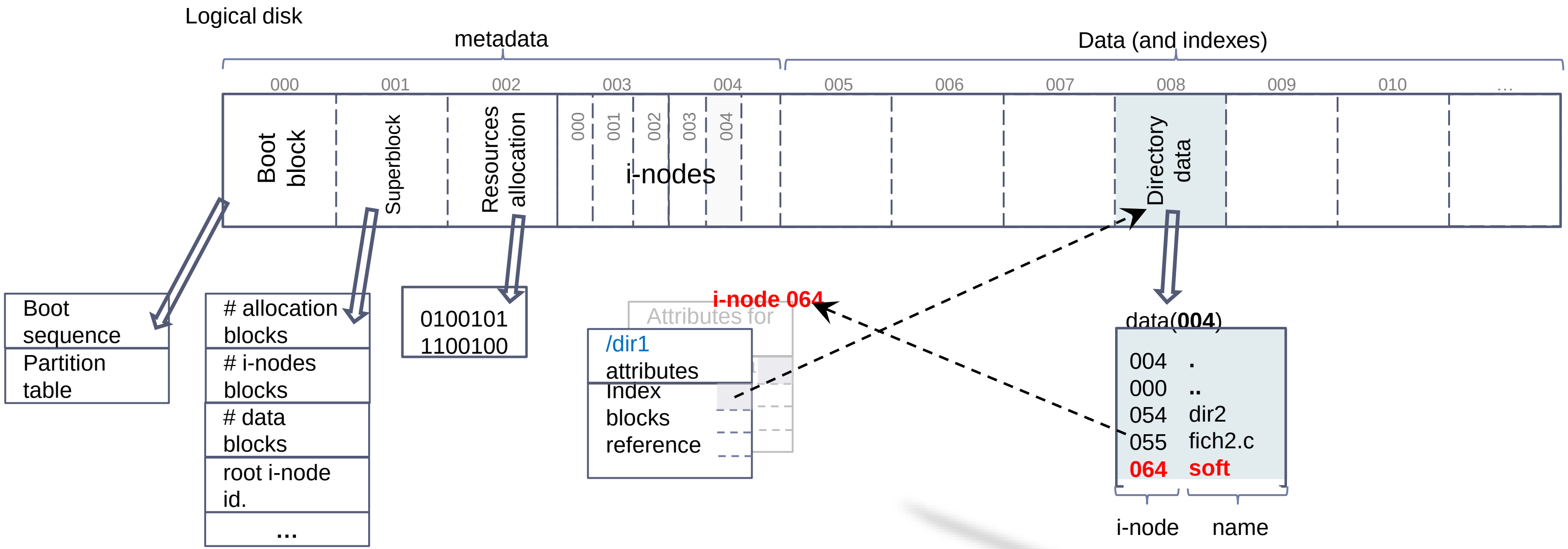


File system:
Unix-like representation: Symbolic link (soft link)



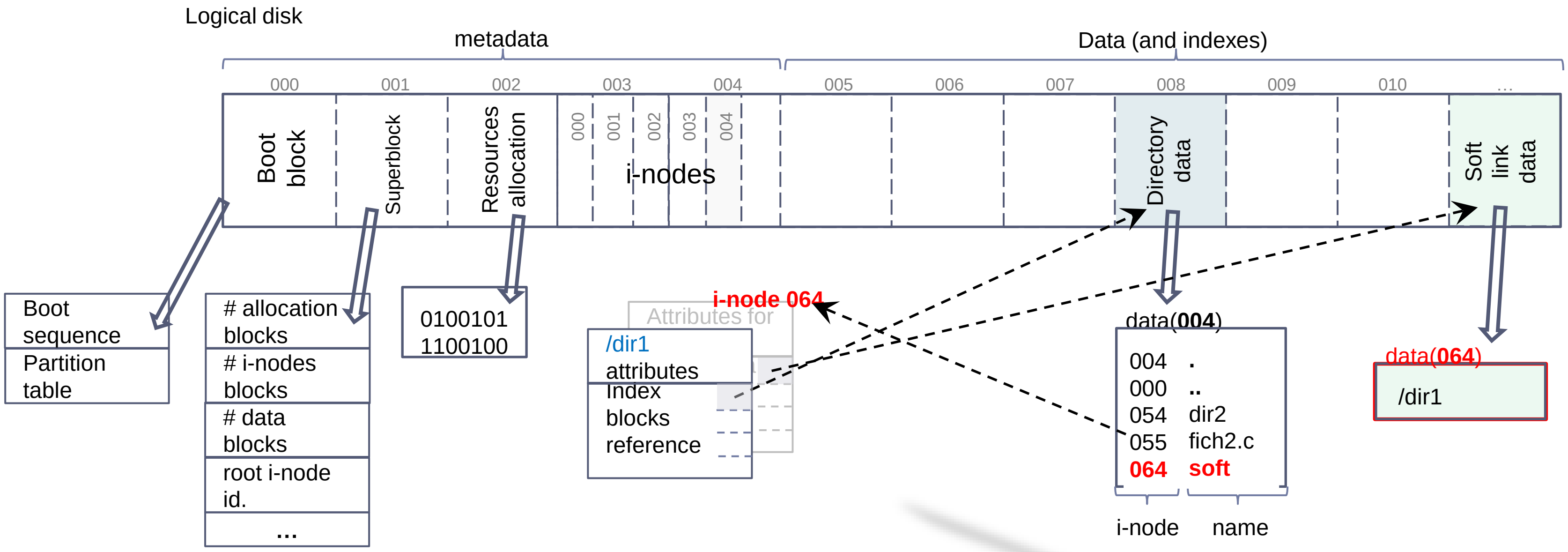
ln -s /dir1 /dir1/soft

File system:
Unix-like representation: Symbolic link (soft link)



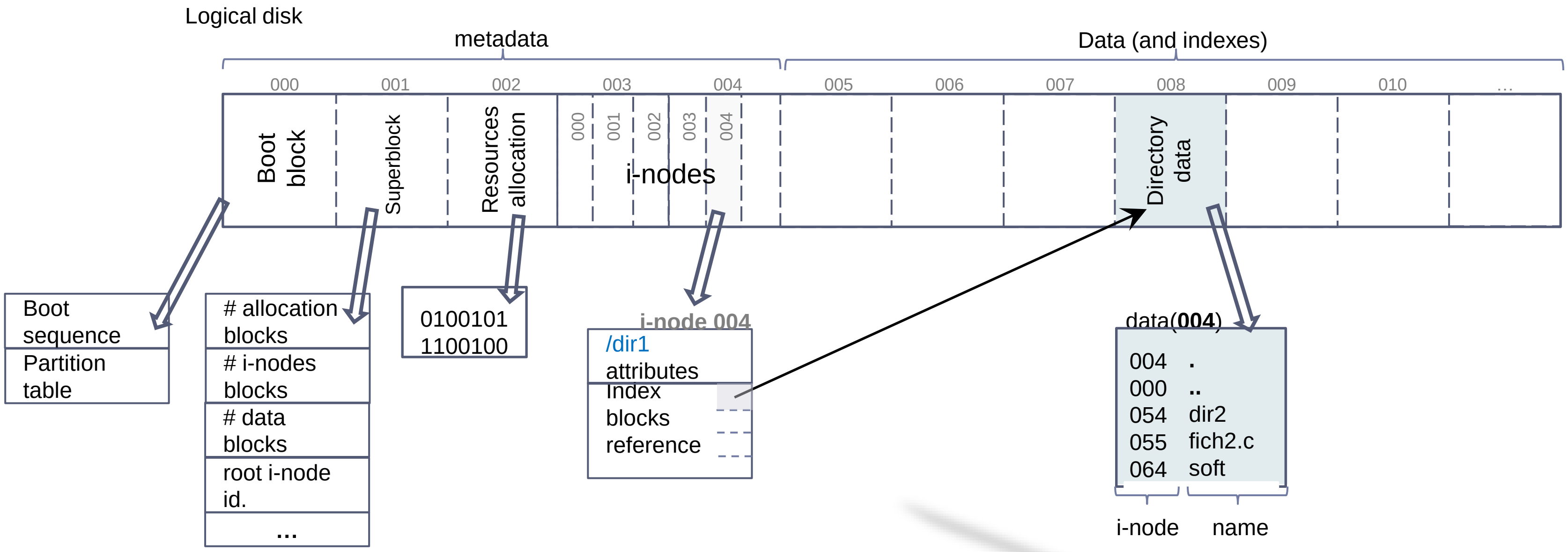
ln -s /dir1 /dir1/soft

File system:
Unix-like representation: Symbolic link (soft link)



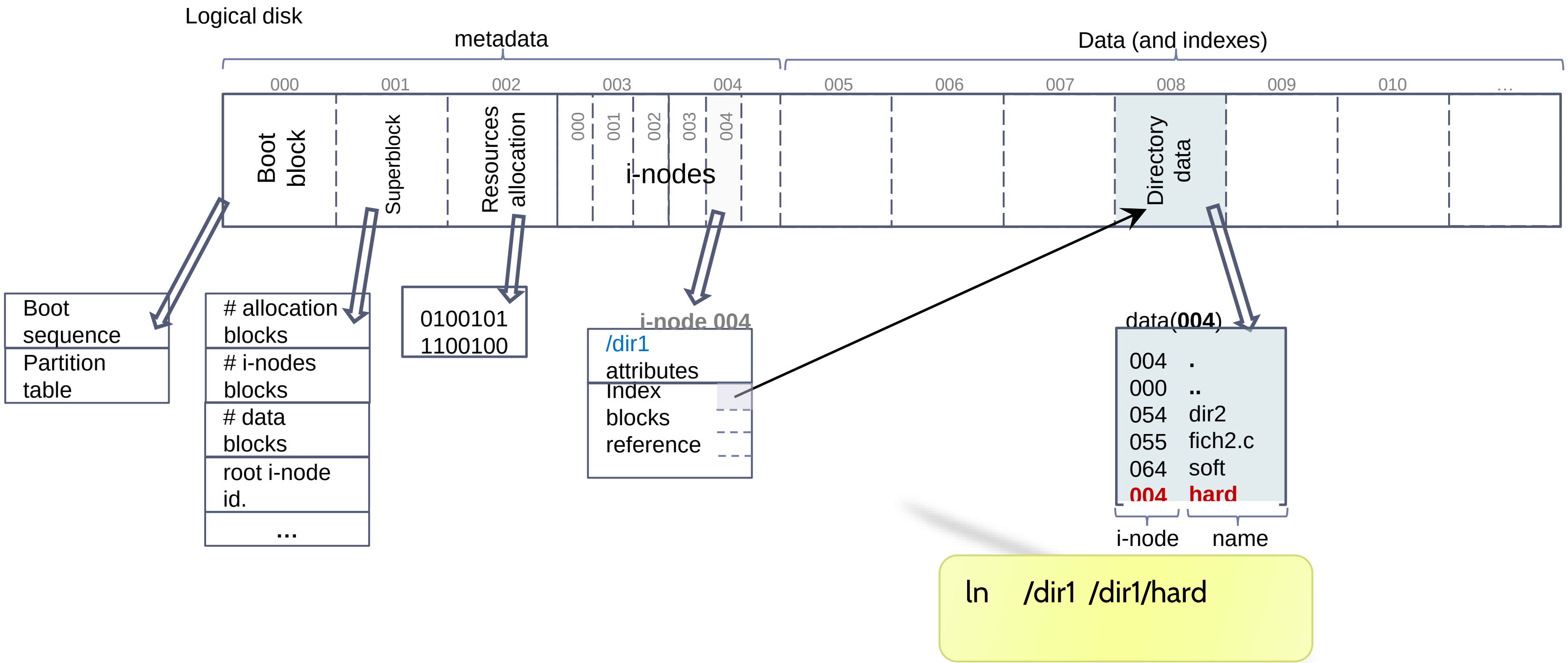
ln -s /dir1 /dir1/soft

File system:
Unix-like representation: hard link

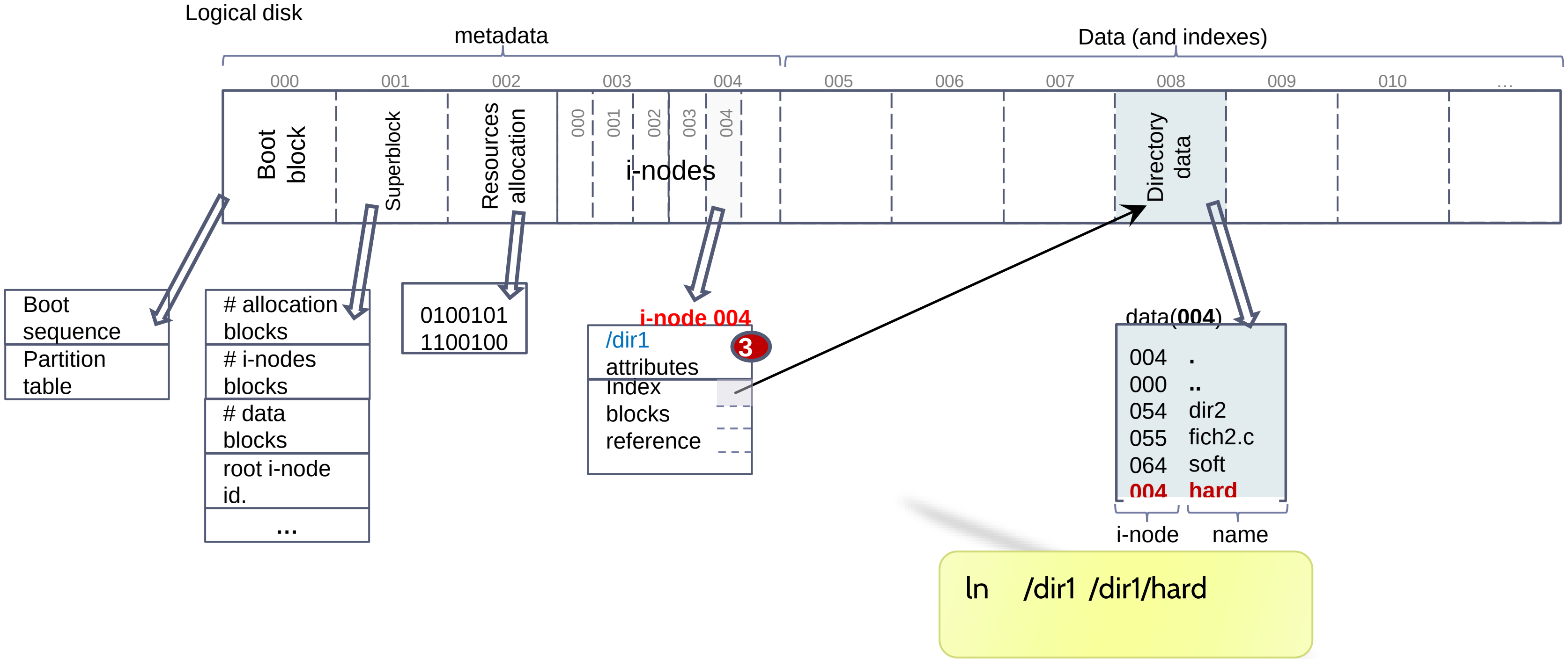


ln /dir1 /dir1/hard

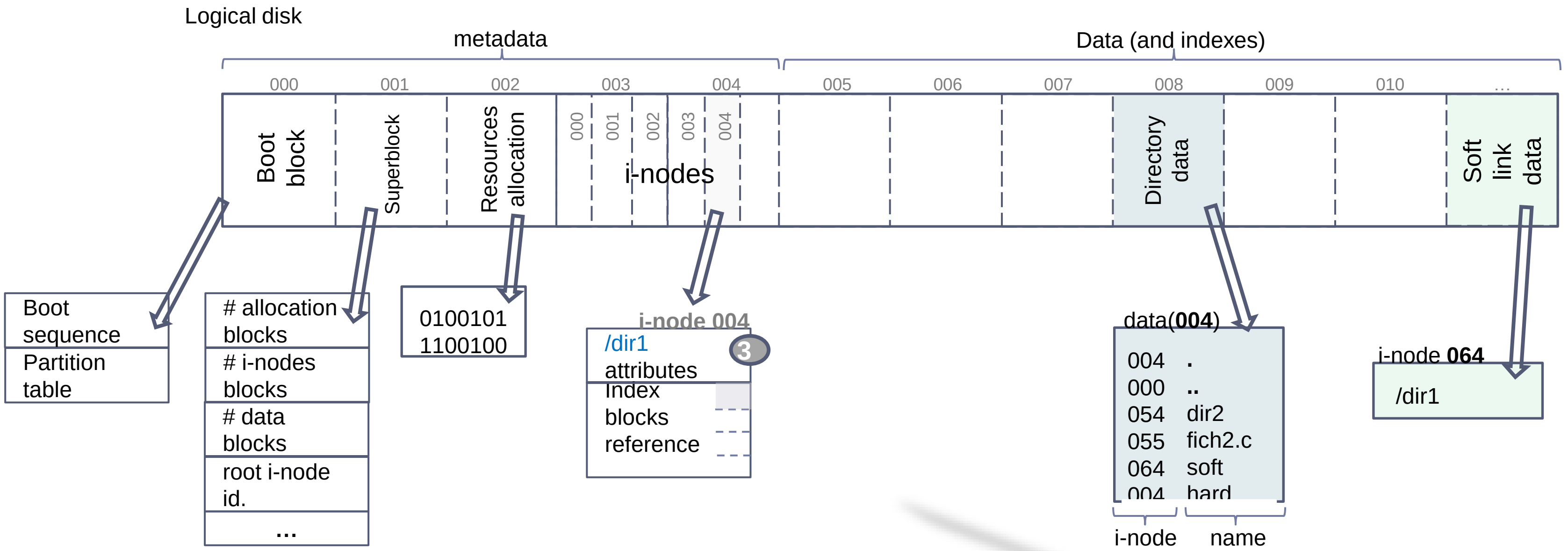
File system:
Unix-like representation: hard link



File system:
Unix-like representation: hard link

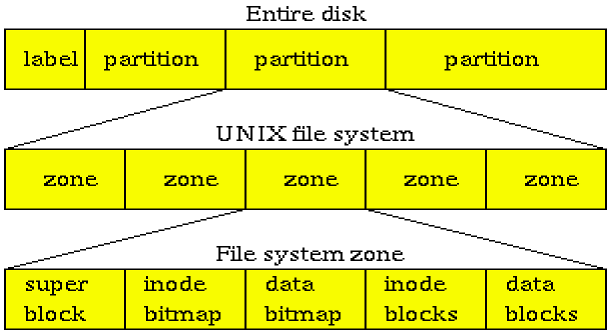


File system:
hard link vs soft link



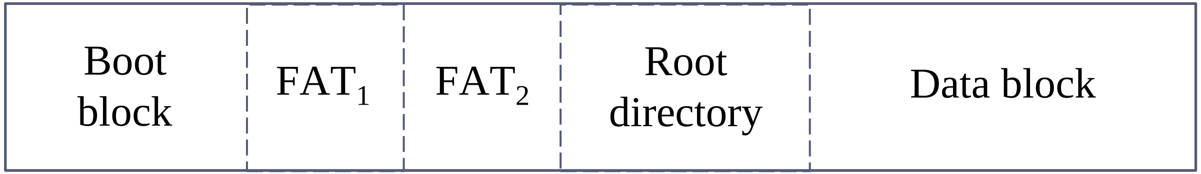

```
ln -s  /dir1  /dir1/soft
ln     /dir1  /dir1/hard
```

File system structures

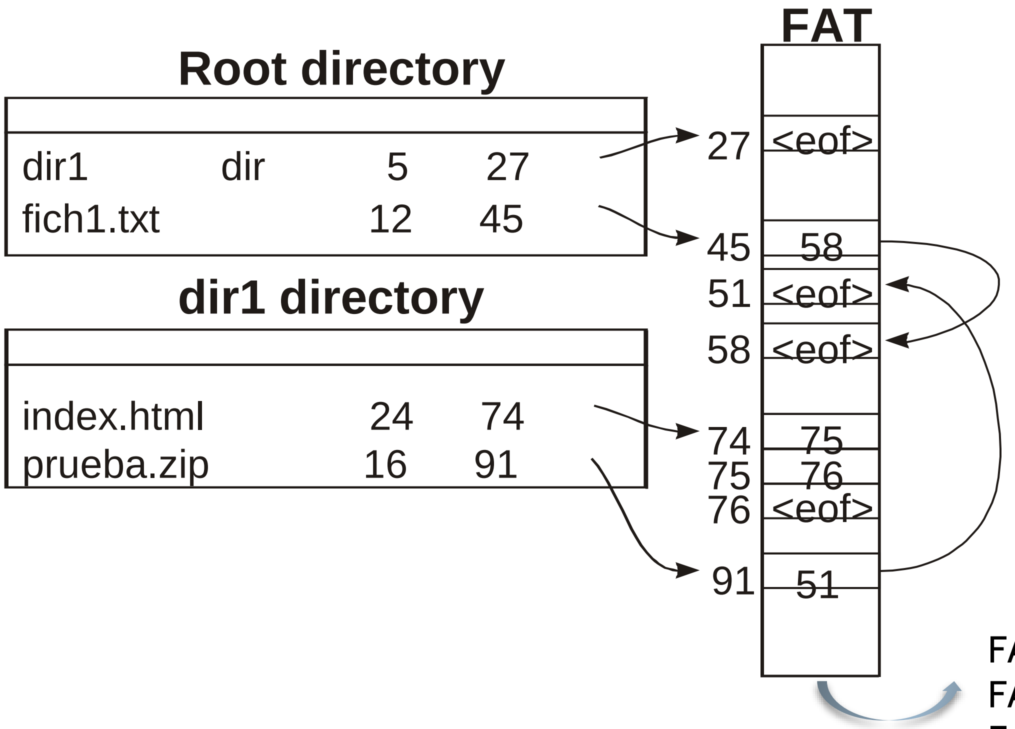


- UNIX/Linux
- FAT

File sytem structures:
FAT

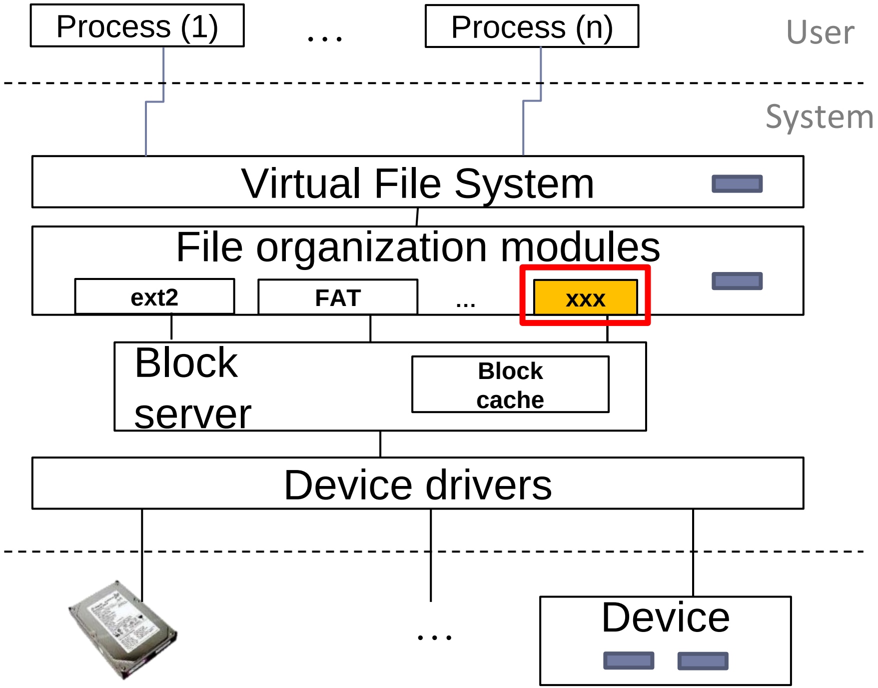


Files and directories representation:
FAT



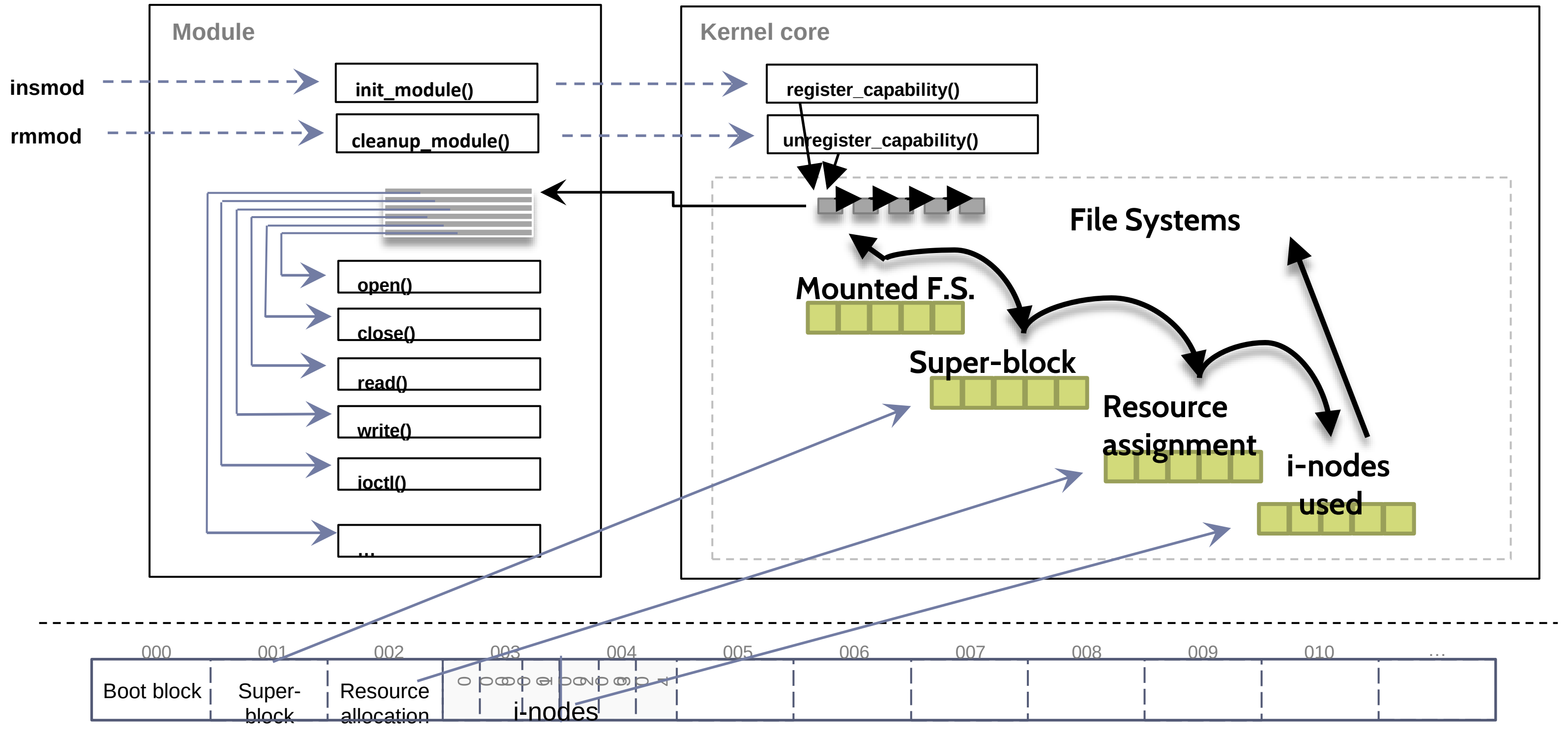
FAT 12 bits (2^{12} blocks)
FAT 16 bits (2^{16} blocks)
FAT 32 bits (2^{32} blocks)

Design and development of a file system

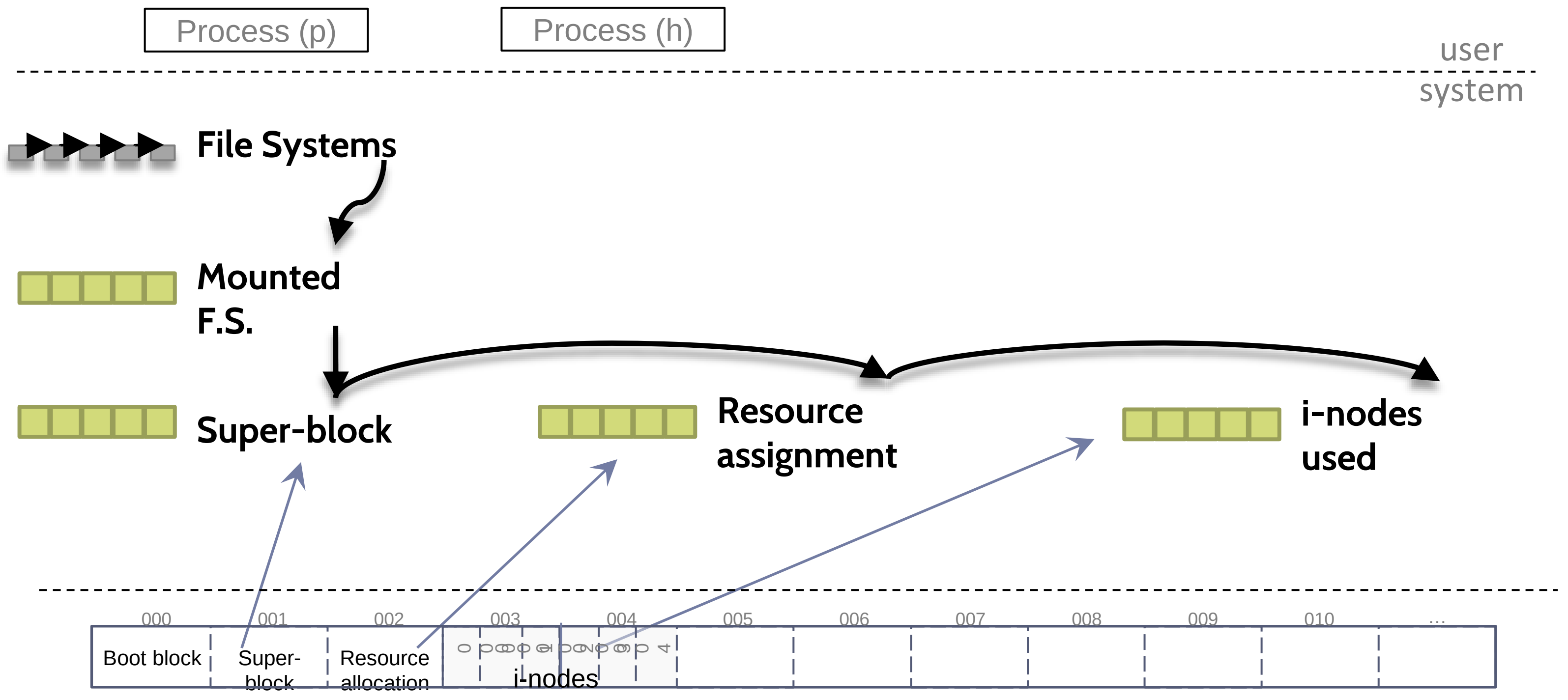


- File system requirements
- Main data structures in the secondary memory
- **Main data structures in the main memory**
- Block management
- Internal (and service) functions

Initial design: load disk metadata in memory…



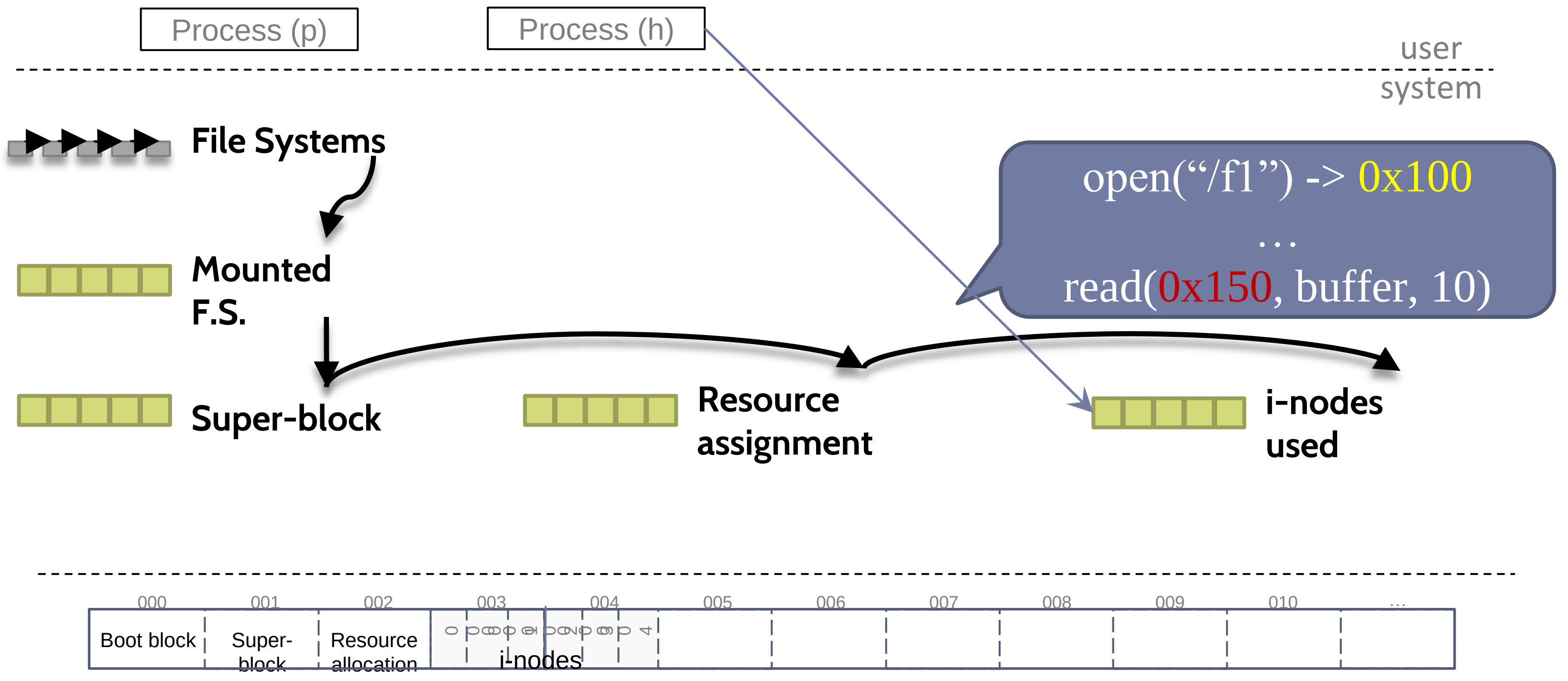
Initial design: load disk metadata in memory…



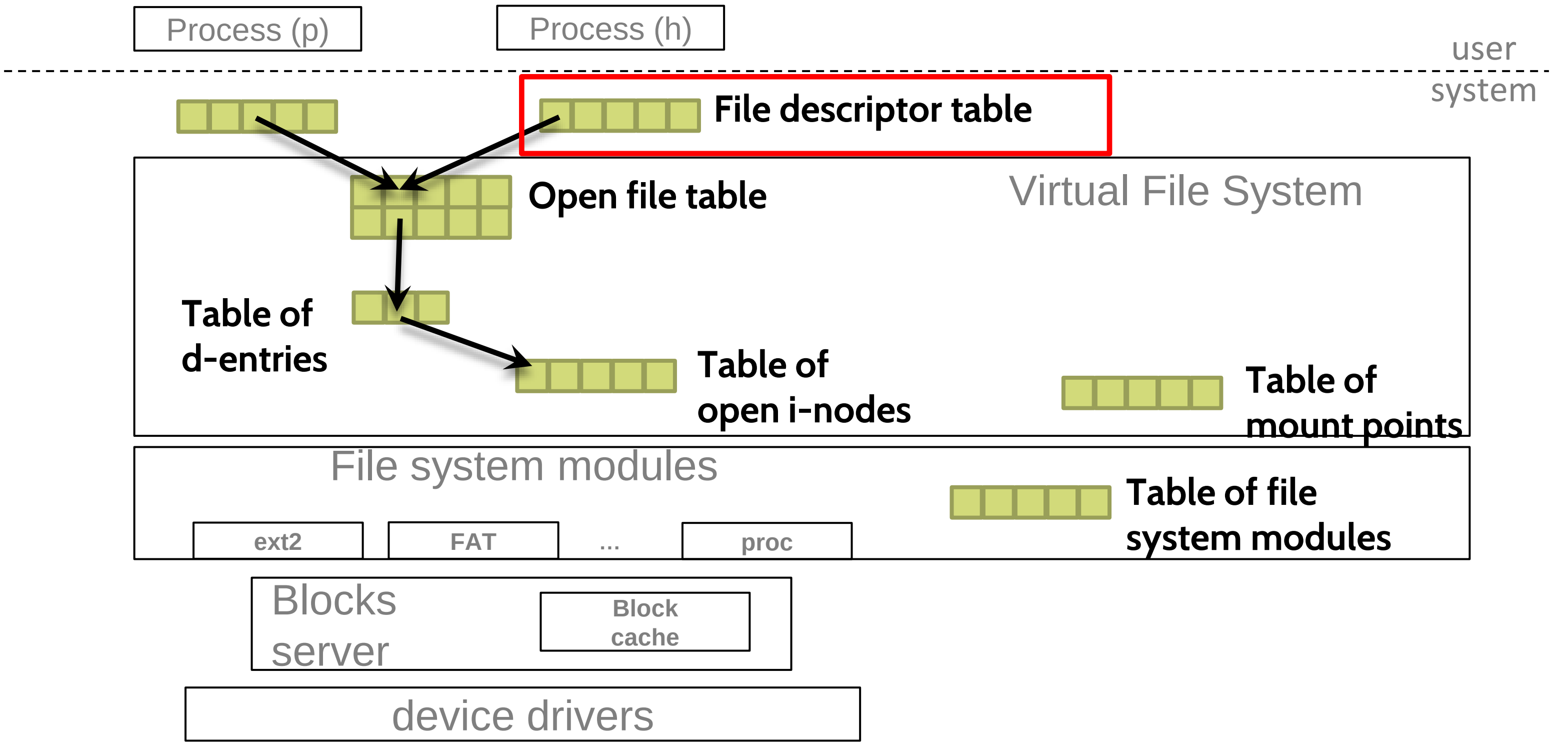
Main goals
(for a Unix-like file system)

- Processes have to use a secure interface, without direct access to the kernel data structures.
- Share the file offset position among processes from the same parent that open the file.
- Offer functionality for working with a file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file systems registered in the kernel, and keep track of the mount point of these file systems.

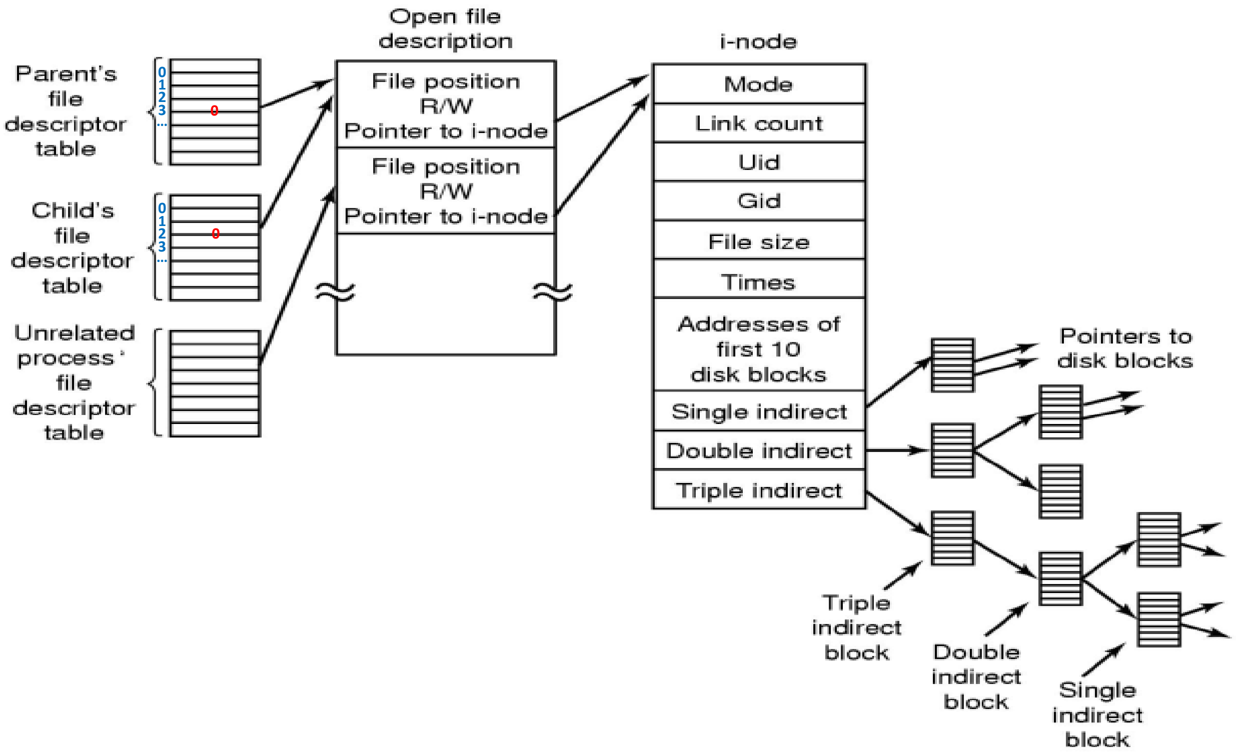
Example of direct access to kernel address...



Main management structures



Main management structures



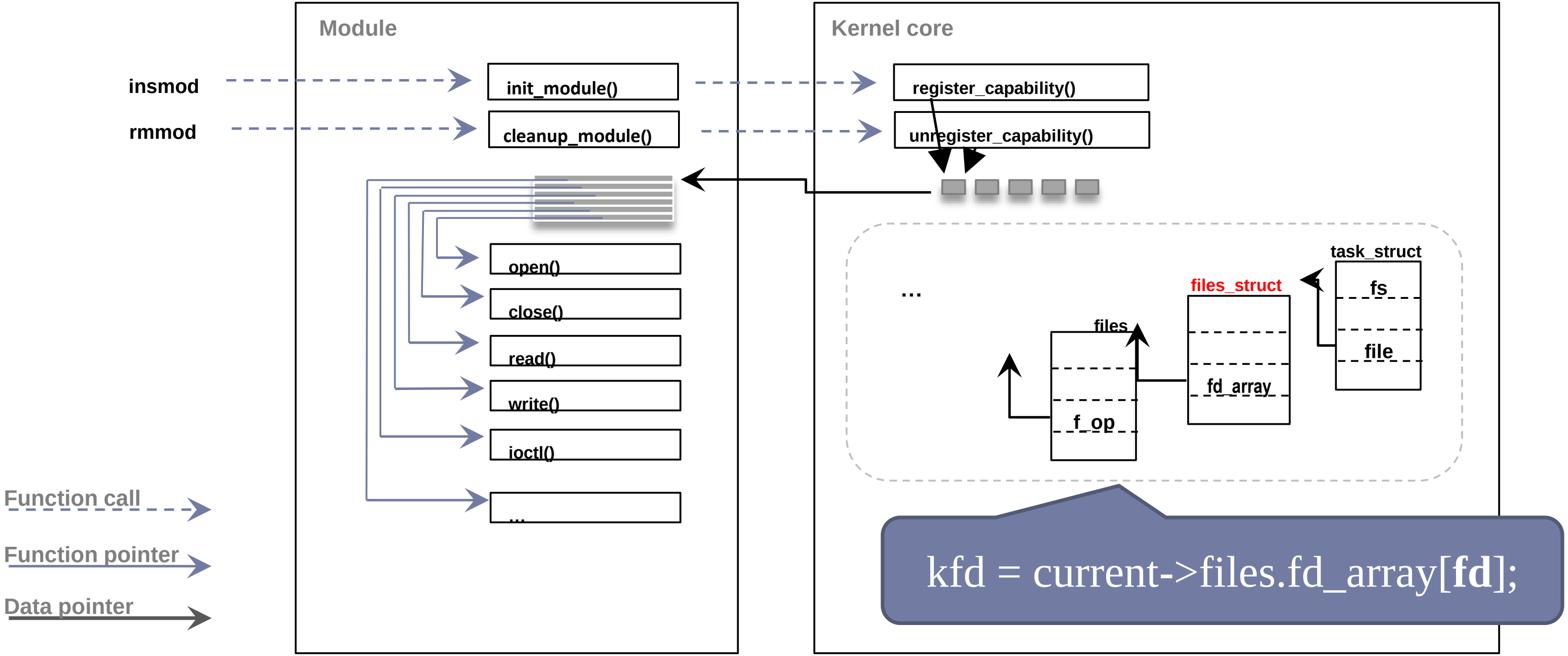
Main management structures

File descriptor table: Linux

```
struct fs_struct {
    atomic_t    count;              /* structure's usage count */
    spinlock_t  file_lock;          /* lock protecting this structure */
    int         max_fds;            /* maximum number of file objects */
    int         max_fdset;          /* maximum number of file descriptors */
    int         next_fd;            /* next file descriptor number */
    struct file **fd;               /* array of all file objects */
    fd_set      *close_on_exec;     /* file descriptors to close on exec() */
    fd_set      *open_fds;          /* pointer to open file descriptors */
    fd_set      close_on_exec_init; /* initial files to close on exec() */
    fd_set      open_fds_init;      /* initial set of file descriptors */
    struct file *fd_array[NR_OPEN_DEFAULT]; /* array of file objects */
};
```

Main management structures

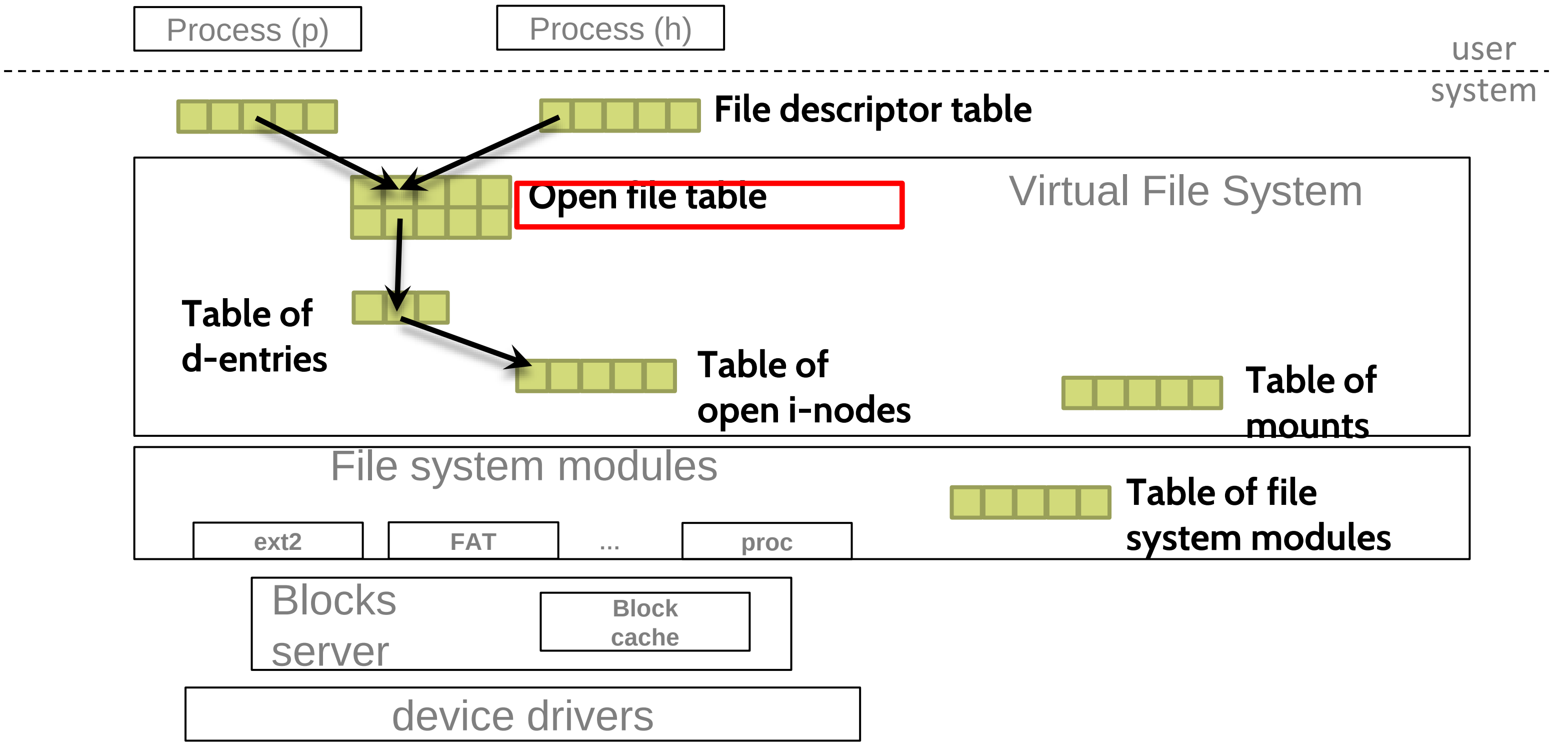
Descriptors table (open files): Linux



Main goals
(for a Unix-like file system)

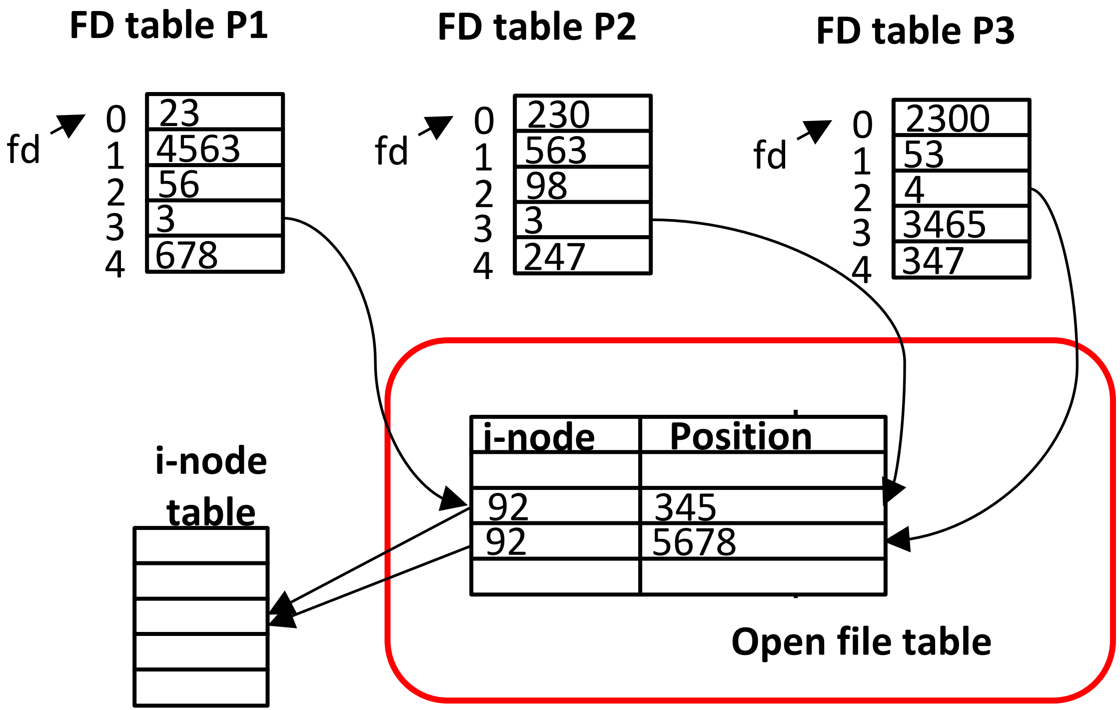
- The processes have to use a secure interface, without direct access to the kernel representation.
- To share the file offset among process from the same parent that open the file.
- To have a working session with the file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file system registered in the kernel, and keep track of the mount points of these file systems.

Main management structures



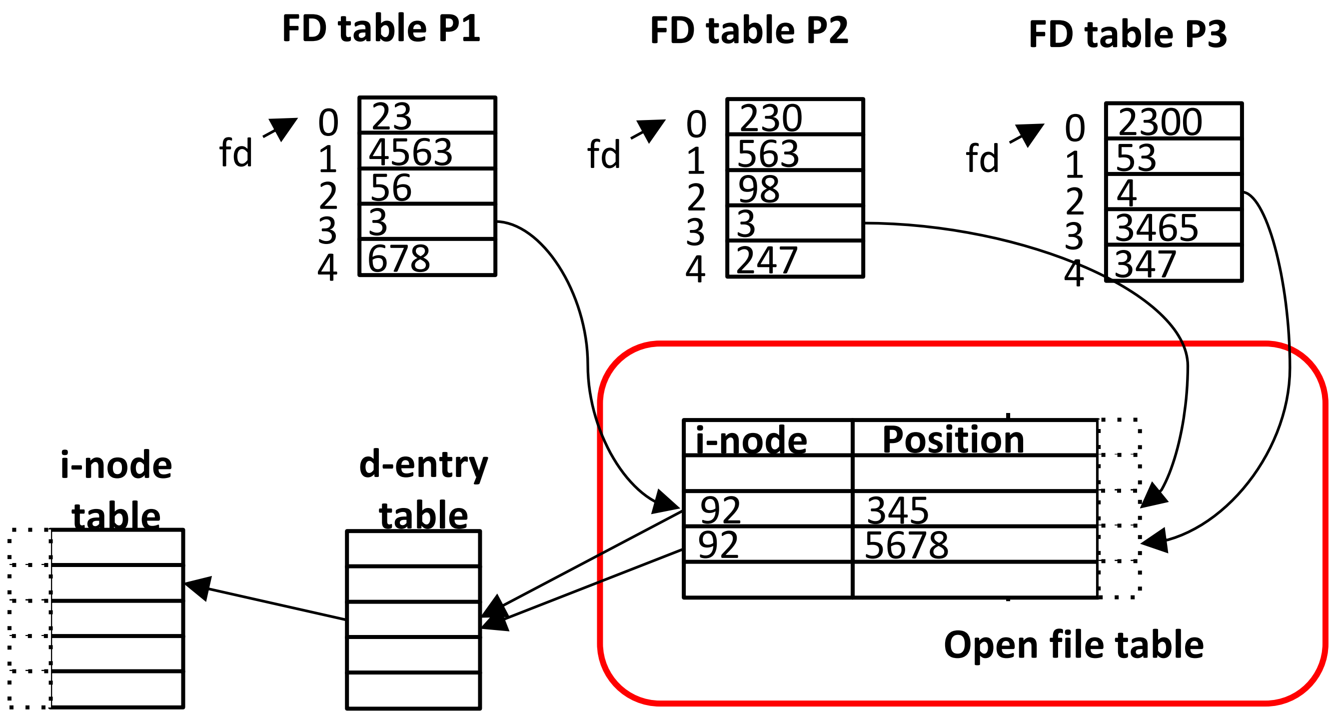
Main management structures

Seek pointers table



Main management structures

Seek pointers table: Linux



Main management structures

File table: Linux

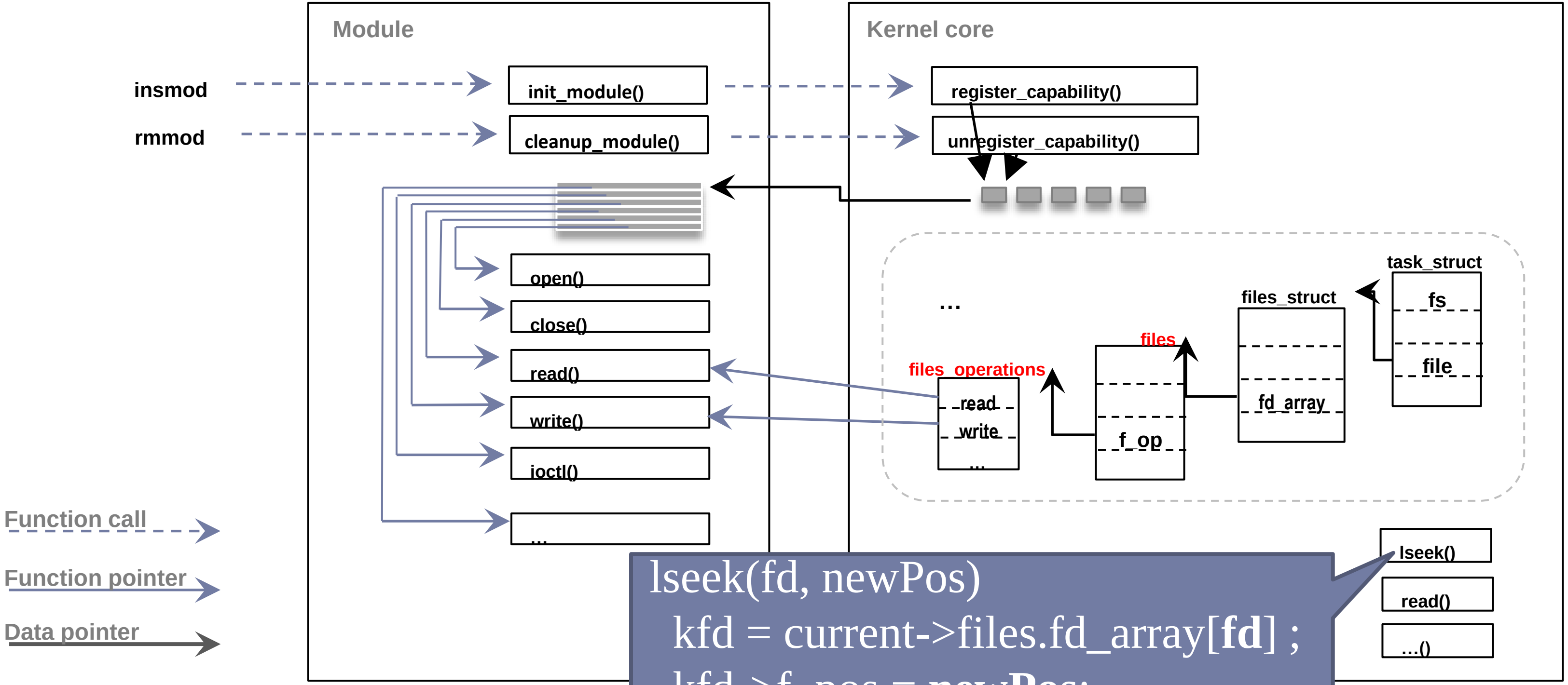
```
struct file {
        struct dentry             *f_dentry;
        struct vfsmount           *f_vfsmnt;
        struct file_operations    *f_op;
        mode_t                    f_mode;
        loff_t                    f_pos;
        struct fown_struct        f_owner;
        unsigned int              f_uid, f_gid;
        unsigned long             f_version;
        …
} ;
```

```
struct file_operations {
    int       (*open)    (struct inode *, struct file *);
    ssize_t   (*read)    (struct file *, char *, size_t, loff_t *);
    ssize_t   (*write)   (struct file *, const char *, size_t, loff_t *);
    loff_t    (*llseek)  (struct file *, loff_t, int);
    int       (*ioctl)   (struct inode *, struct file *,
                          unsigned int, ulong);
    int       (*readdir) (struct file *, void *, filldir_t);
    int       (*mmap)    (struct file *, struct vm_area_struct *);
    …
};
```

http://www.win.tue.nl/~aeb/linux/lk/lk-8.html

Main management structures

File table: Linux

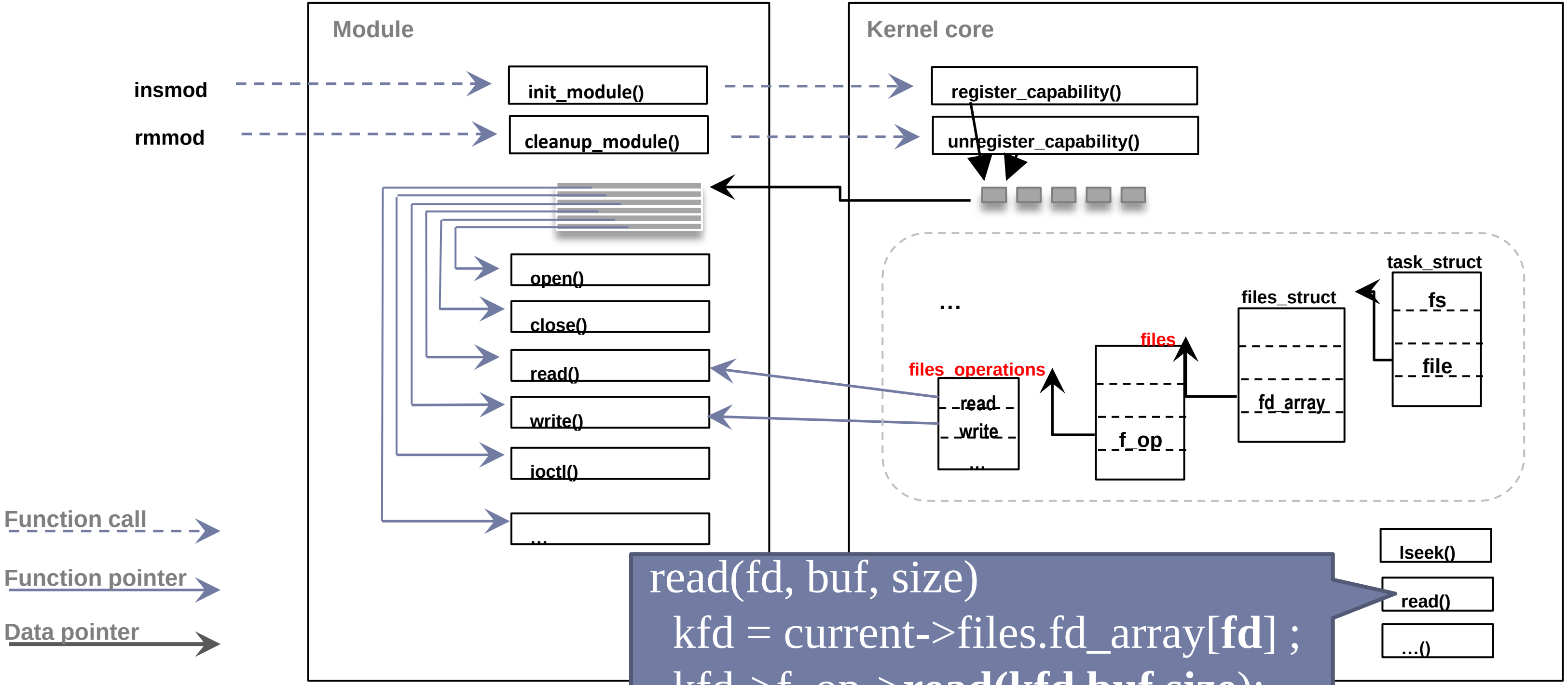


lseek(fd, newPos)
kfd = current->files.fd_array[**fd**] ;
kfd->f_pos = **newPos**;

http://www.xml.com/ldd/chapter/book/ch02.html

Main management structures

File table: Linux

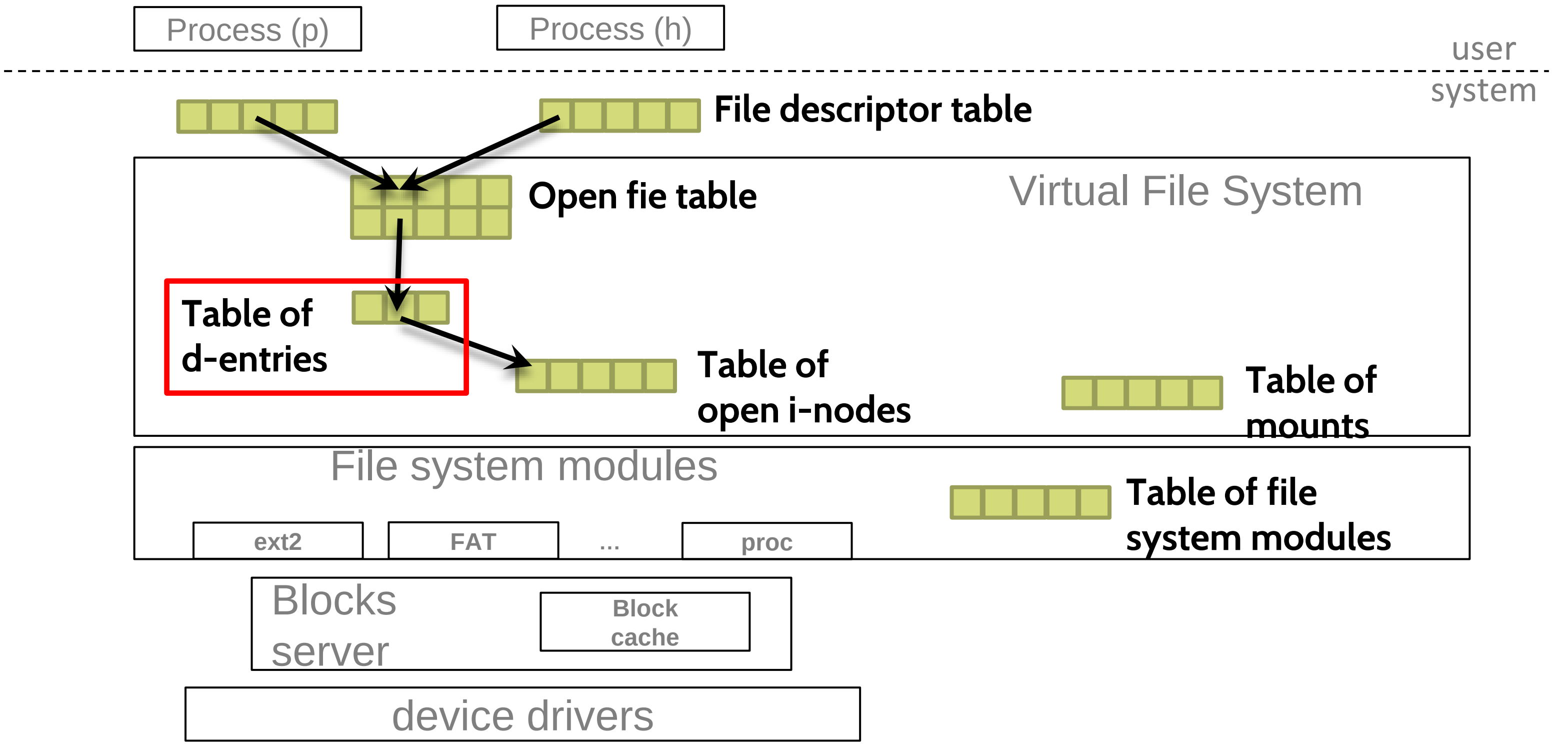


read(fd, buf, size)
  kfd = current->files.fd_array[**fd**] ;
  kfd->f_op->**read(kfd,buf,size)**;

Main goals
(for a Unix-like file system)

- The processes have to use a secure interface,
  without direct access to the kernel representation.
- To share the file offset among process from the same parent that open the file.
- To have a working session with the file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file system registered in the kernel, and keep track of the mount points of these file systems.

Main management structures



Main management structures

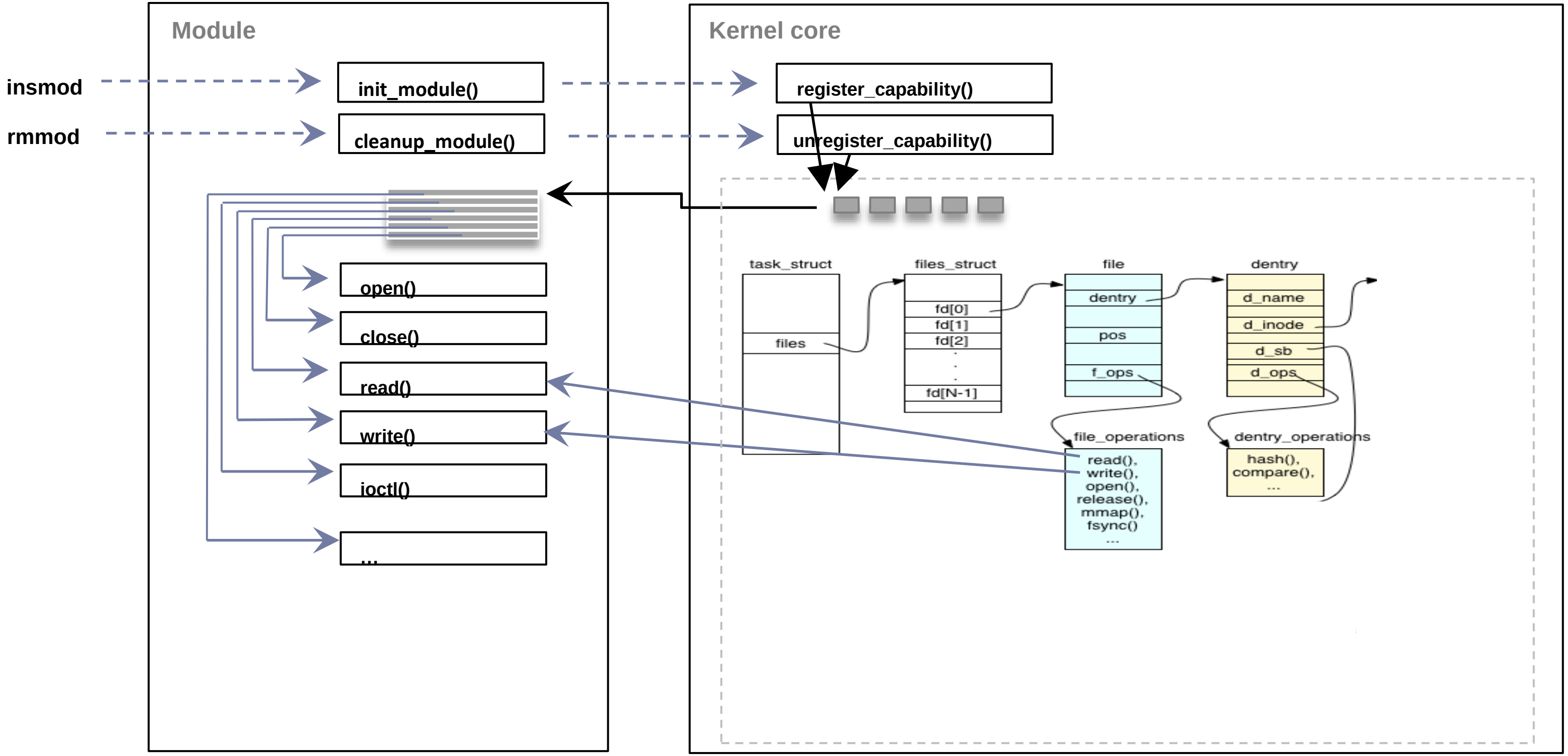
Table of d-entries (directory entries): Linux

```
struct dentry {
        struct  inode             *d_inode;
        struct  dentry            *d_parent;
        struct  qstr              d_name;
        struct  dentry_operations *d_op;
        struct  super_block       *d_sb;
        struct  list_head         d_subdirs;
        ...
}
```

```
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 *);
    int (*d_delete)     (struct dentry *);
    void (*d_release)   (struct dentry *);
    void (*d_iput)      (struct dentry *,
                         struct inode *);
}
```

Main management structures

Table of d-entries (directory entries): Linux

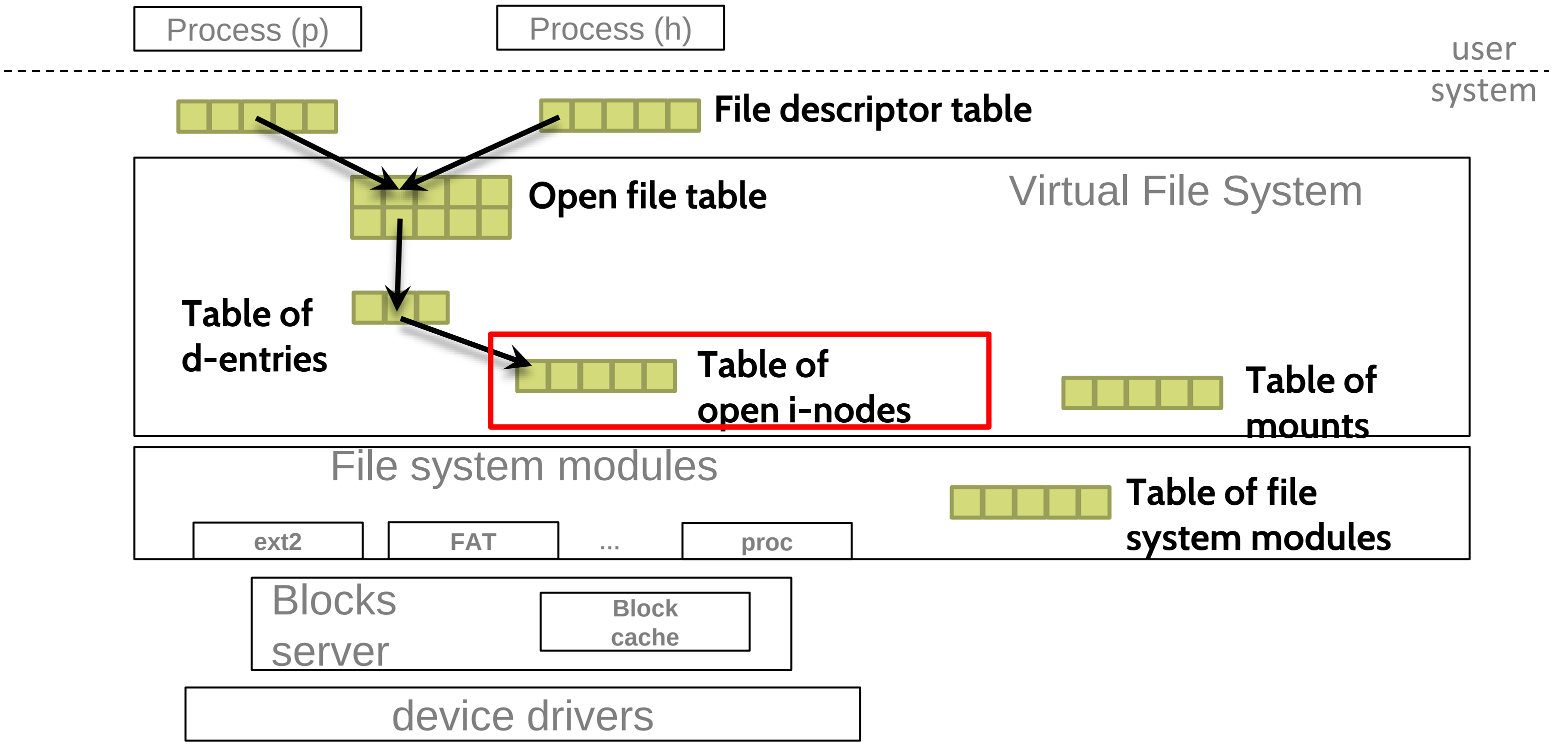


http://www.coins.tsukuba.ac.jp/~yas/classes/os2-2012/2013-02-19/index.html

Main goals
(for a Unix-like file system)

- The processes have to use a secure interface,
  without direct access to the kernel representation.
- To share the file offset among process from the same parent that open the file.
- To have a working session with the file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file system registered in the kernel, and keep track of the mount points of these file systems.

Main management structures



Main management structures

Table of i-nodes: Linux

```
struct inode {
    unsigned long       i_ino;
    umode_t             i_mode;
    uid_t               i_uid;
    gid_t               i_gid;
    kdev_t              i_rdev;
    loff_t              i_size;
    struct timespec     i_atime;
    struct timespec     i_ctime;
    struct timespec     i_mtime;
    struct super_block      *i_sb;
    struct inode_operations  *i_op;
    struct address_space     *i_mapping;
    struct list_head         i_dentry;
    ...
};
```

Main management structures

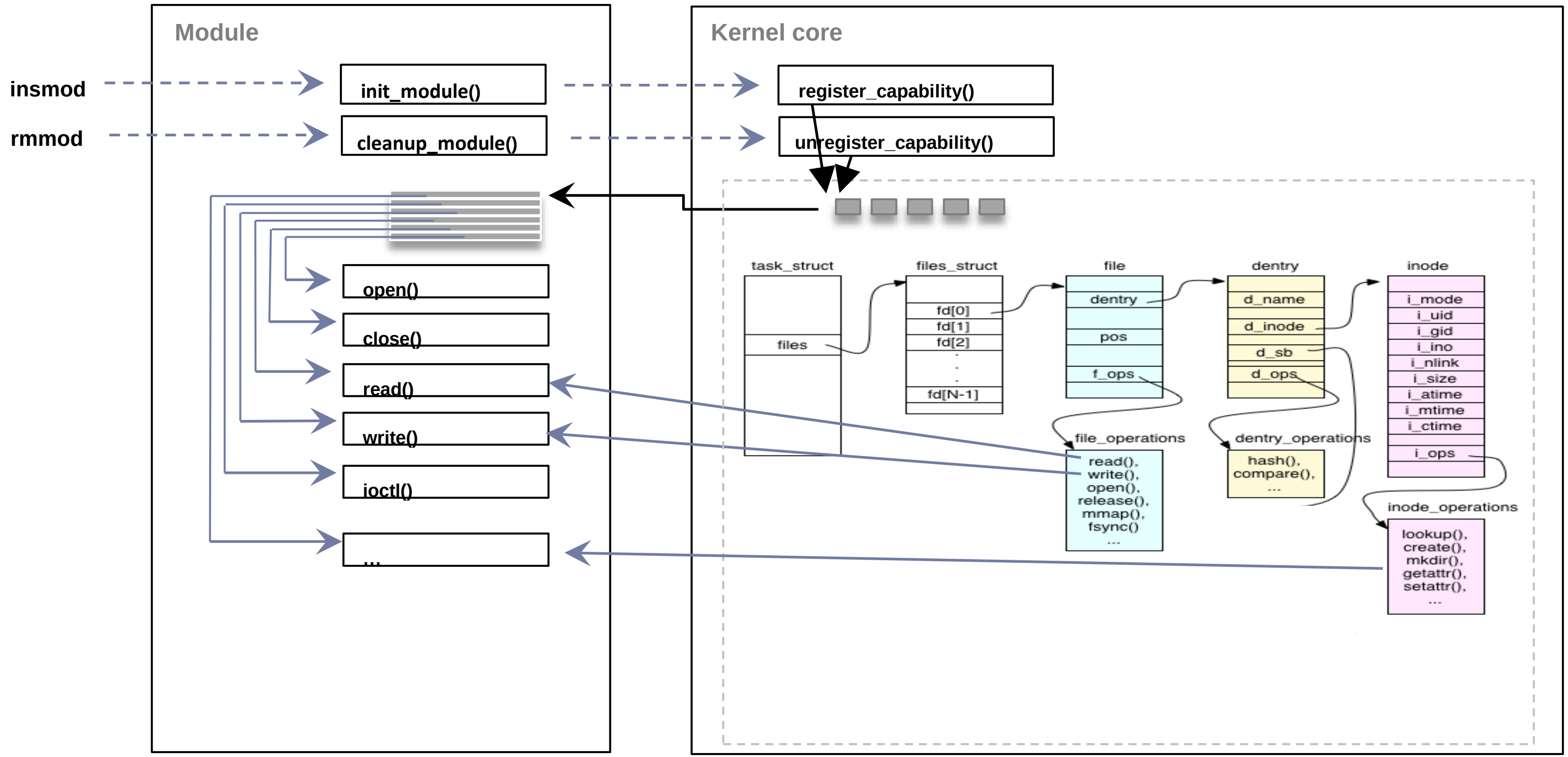
Table of i-nodes: Linux

```
struct inode_operations {

    int (*create) (struct inode *,
                   struct dentry *, int);
    int (*unlink) (struct inode *,
                   struct dentry *);
    int (*mkdir) (struct inode *,
                  struct dentry *, int);
    int (*rmdir) (struct inode *,
                  struct dentry *);
    int (*mknod) (struct inode *,
                  struct dentry *,
                  int, dev_t);
    int (*rename) (struct inode *,
                   struct dentry *,
                   struct inode *,
                   struct dentry *);
    void (*truncate) (struct inode *);
    struct dentry * (*lookup) (struct inode *,
                               struct dentry *);

    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *,
                    struct iattr *);
    int (*getattr) (struct vfsmount *mnt,
                    struct dentry *,
                    struct kstat *);
    int (*setxattr) (struct dentry *,
                     const char *,
                     const void *,
                     size_t, int);
    ssize_t (*getxattr) (struct dentry *,
                         const char *,
                         void *, size_t);
    ssize_t (*listxattr) (struct dentry *,
                          char *, size_t);
    int (*removexattr) (struct dentry *,
                        const char *);

    int     (*link) (struct dentry *,
                     struct inode *,
                     struct dentry *);
    int (*symlink) (struct inode *,
                    struct dentry *,
                    const char *);
    int (*readlink) (struct dentry *,
                     char *, int);
    int (*follow_link) (struct dentry *,
                        struct nameidata *);

};
```

http://www.win.tue.nl/~aeb/linux/lk/lk-8.html

Main management structures

Table of i-nodes: Linux



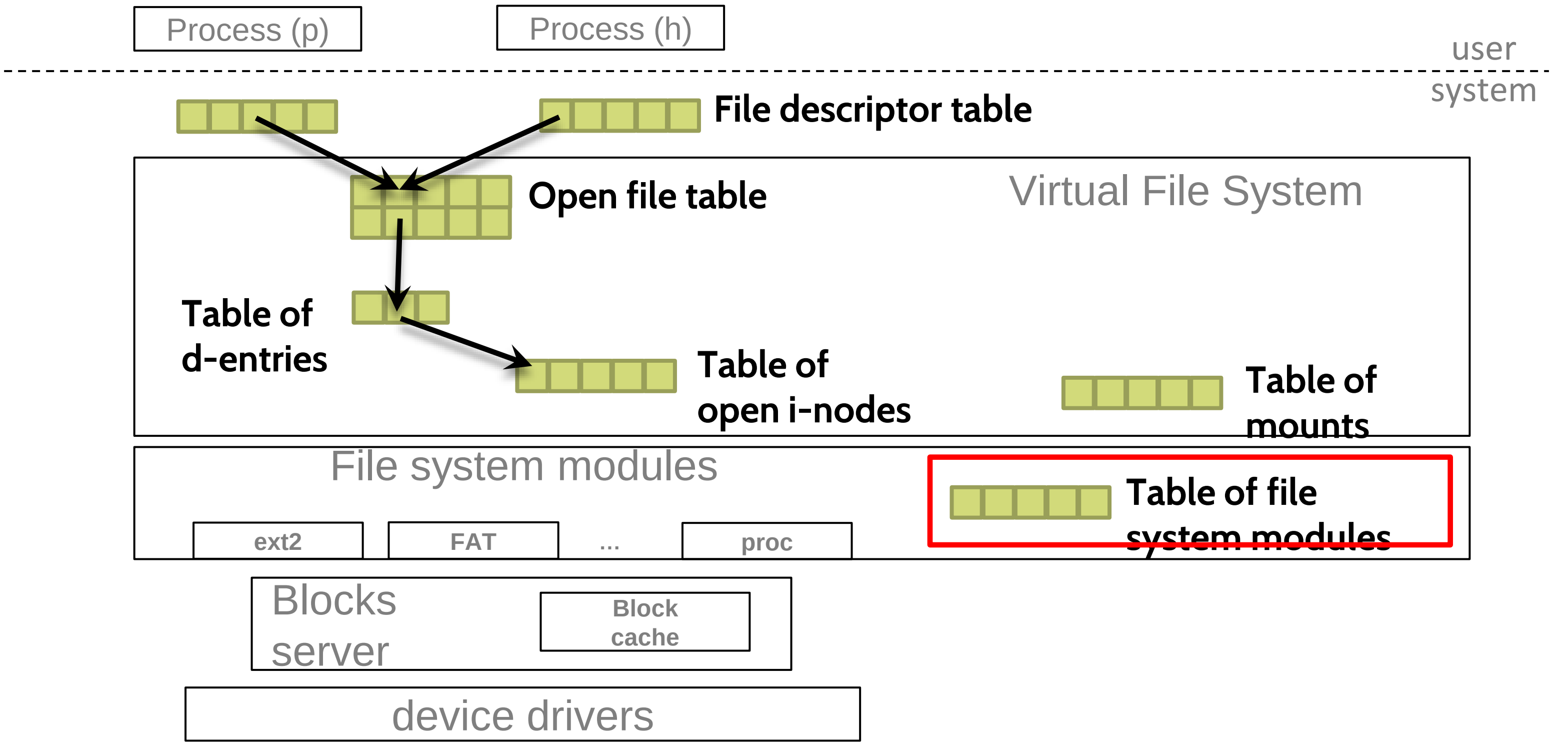
http://www.coins.tsukuba.ac.jp/~yas/classes/os2-2012/2013-02-19/index.html

Main goals
(for a Unix-like file system)

- The processes have to use a secure interface,
  without direct access to the kernel representation.
- To share the file offset among process from the same parent that open the file.
- To have a working session with the file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file systems registered in the kernel, and keep track of the mount points of these file systems.

Alejandro Calderón Mateos

Main management structures



Main management structures

File system table: Linux

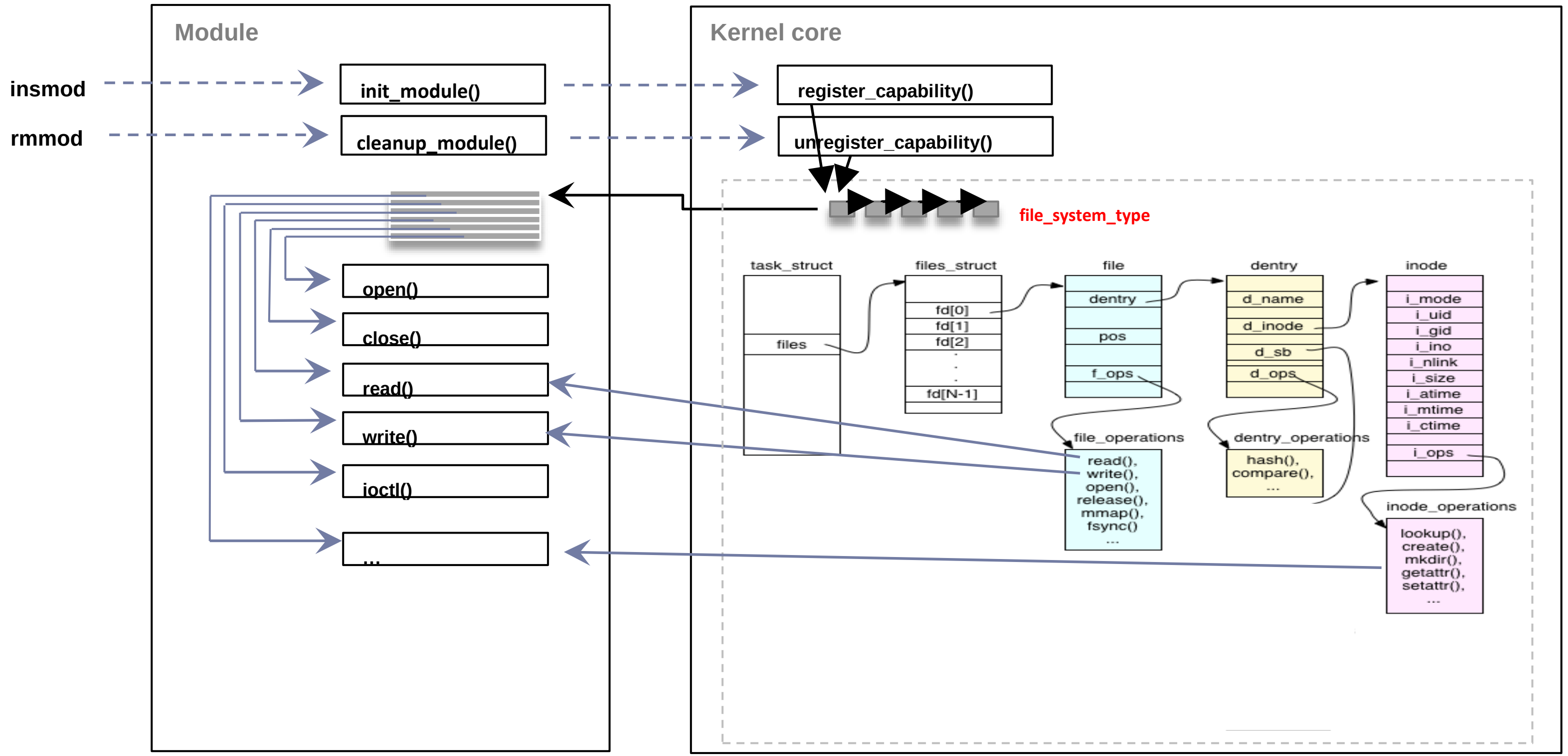
file_systems →

```
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;
    struct  list_head              fs_supers;
    struct  lock_class_key       s_lock_key;
    ...
}
```

http://www.ibm.com/developerworks/library/l-linux-filesystem/

Main management structures

File system table: Linux



http://www.coins.tsukuba.ac.jp/~yas/classes/os2-2012/2013-02-19/index.html

Main management structures

Table of mounts: Linux

current->namespace->list

```
struct vfsmount {
    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_hash;
    struct list_head    mnt_mounts;  /* list of children, anchored here */
    struct list_head    mnt_child;     /* and going through their mnt_child */
    struct list_head    mnt_list;
    atomic_t            mnt_count;
    int                 mnt_flags;
    char               *mnt_devname;  /* Device name, e.g. /dev/hda1 */
};
```


http://www.win.tue.nl/~aeb/linux/lk/lk-8.html

Main management structures

Superblock table: Linux

```
struct super_block {
    dev_t                         s_dev;
    unsigned long                 s_blocksize;
    struct file_system_type       *s_type;
    struct super_operations       *s_op;
    struct dentry                 *s_root;
    ...
};
```

current->namespace->list->mnt_sb

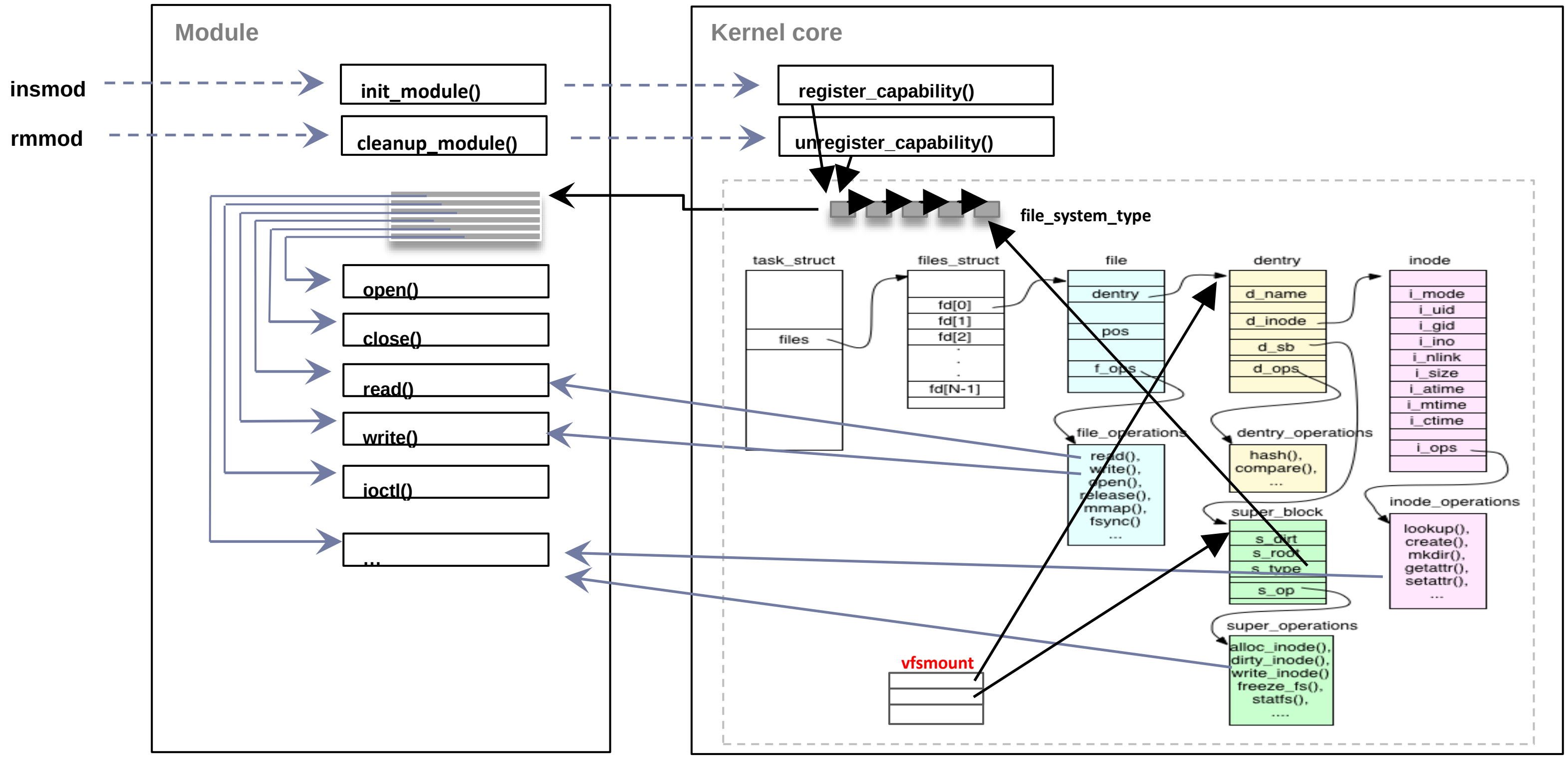
Main management structures
Superblock table: Linux

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*read_inode) (struct inode *);
  void (*dirty_inode) (struct inode *);
  void (*write_inode) (struct inode *, int);
  void (*put_inode) (struct inode *);
  void (*drop_inode) (struct inode *);
  void (*delete_inode) (struct inode *);
  void (*clear_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  void (*write_super) (struct super_block *);
  int  (*sync_fs)(struct super_block *sb, int wait);
  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 (*umount_begin) (struct super_block *);
  int (*show_options)(struct seq_file *, struct vfsmount *);
};
```

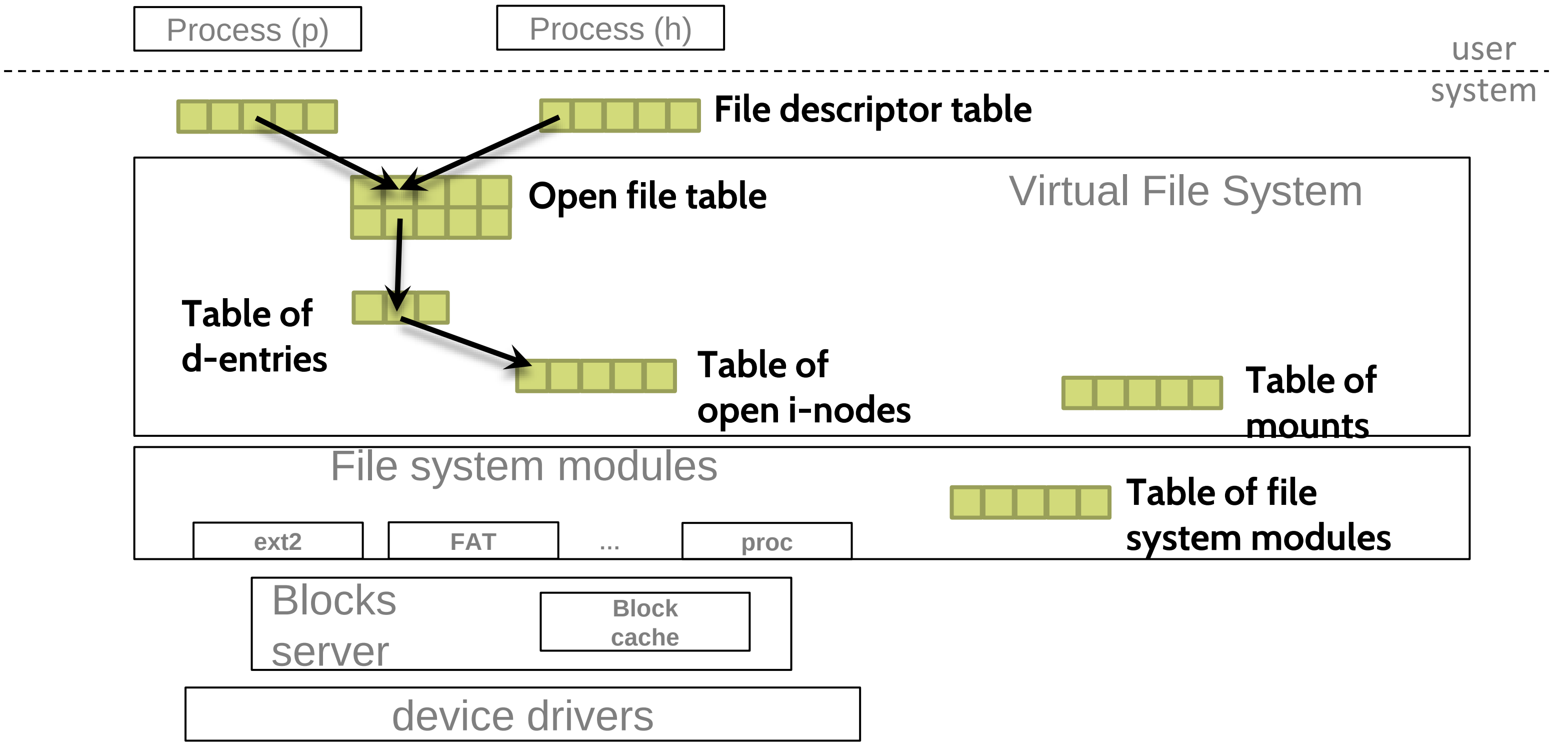

http://www.win.tue.nl/~aeb/linux/lk/lk-8.html

Main management structures

Table of mounts: Linux

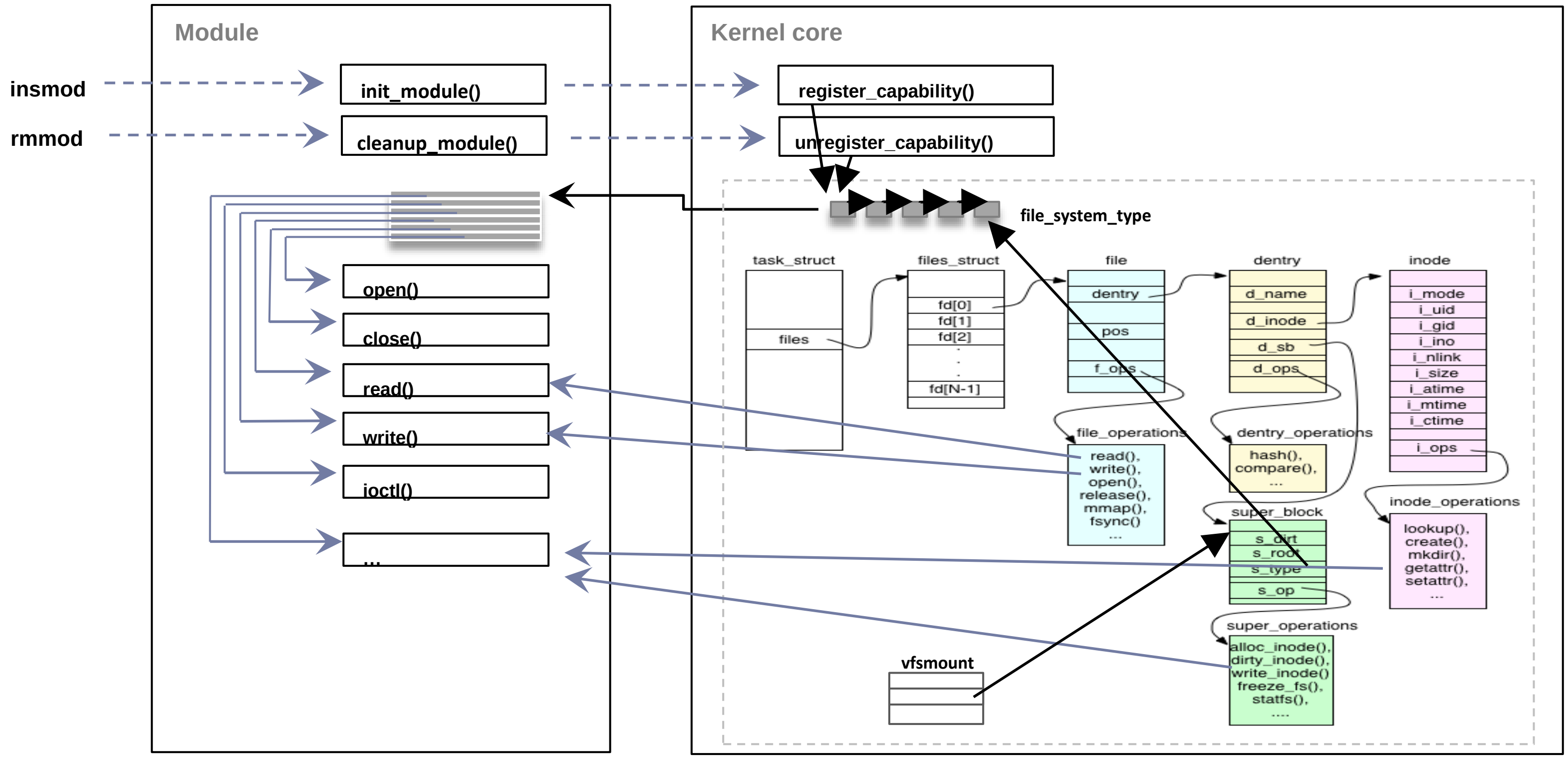


Main management structures
summary

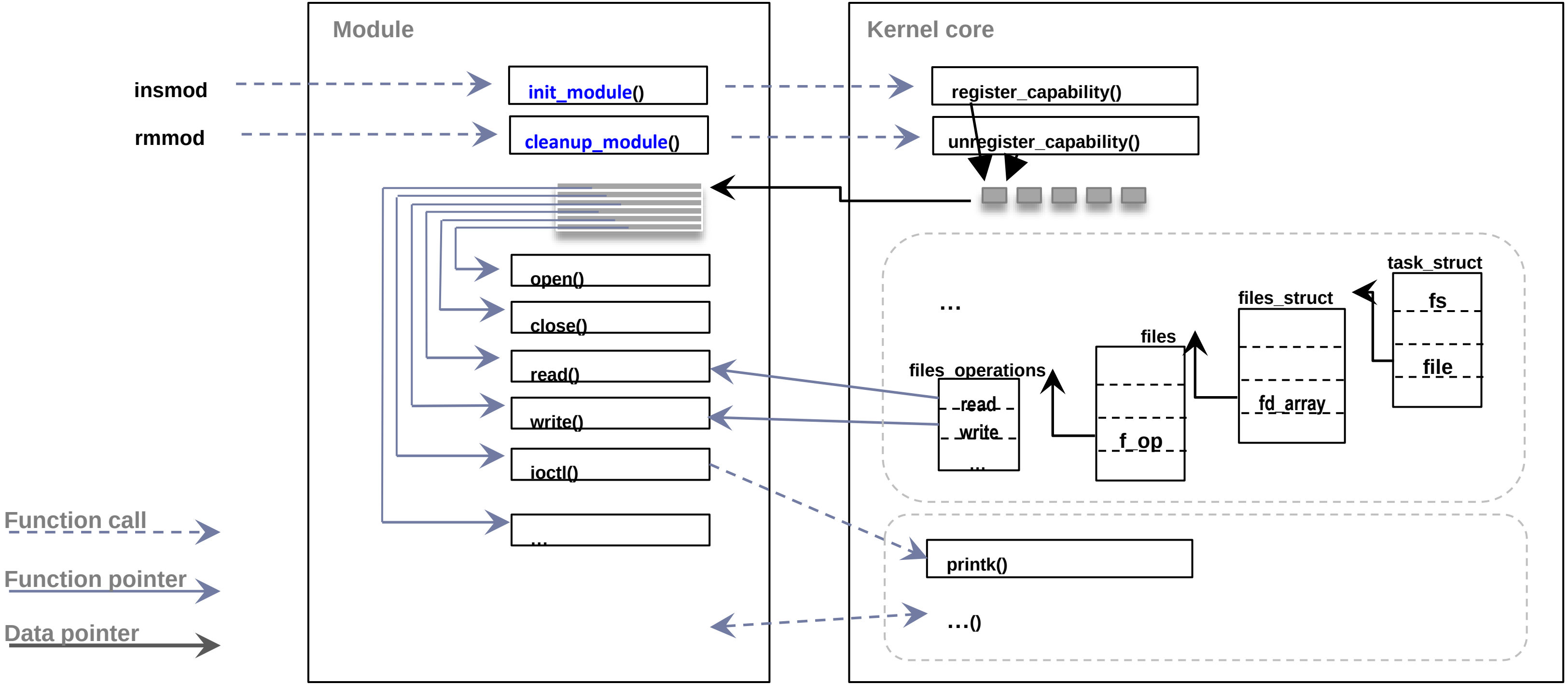


Main management structures

summary



Main management structures
summary (usage)

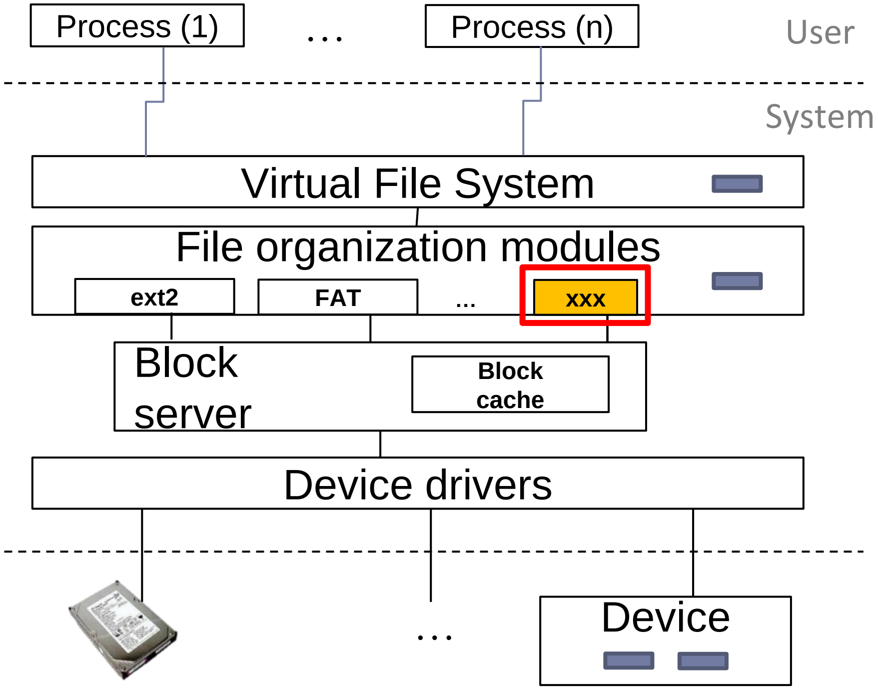


http://www.xml.com/ldd/chapter/book/ch02.html

Main goals
(for a Unix-like file system)

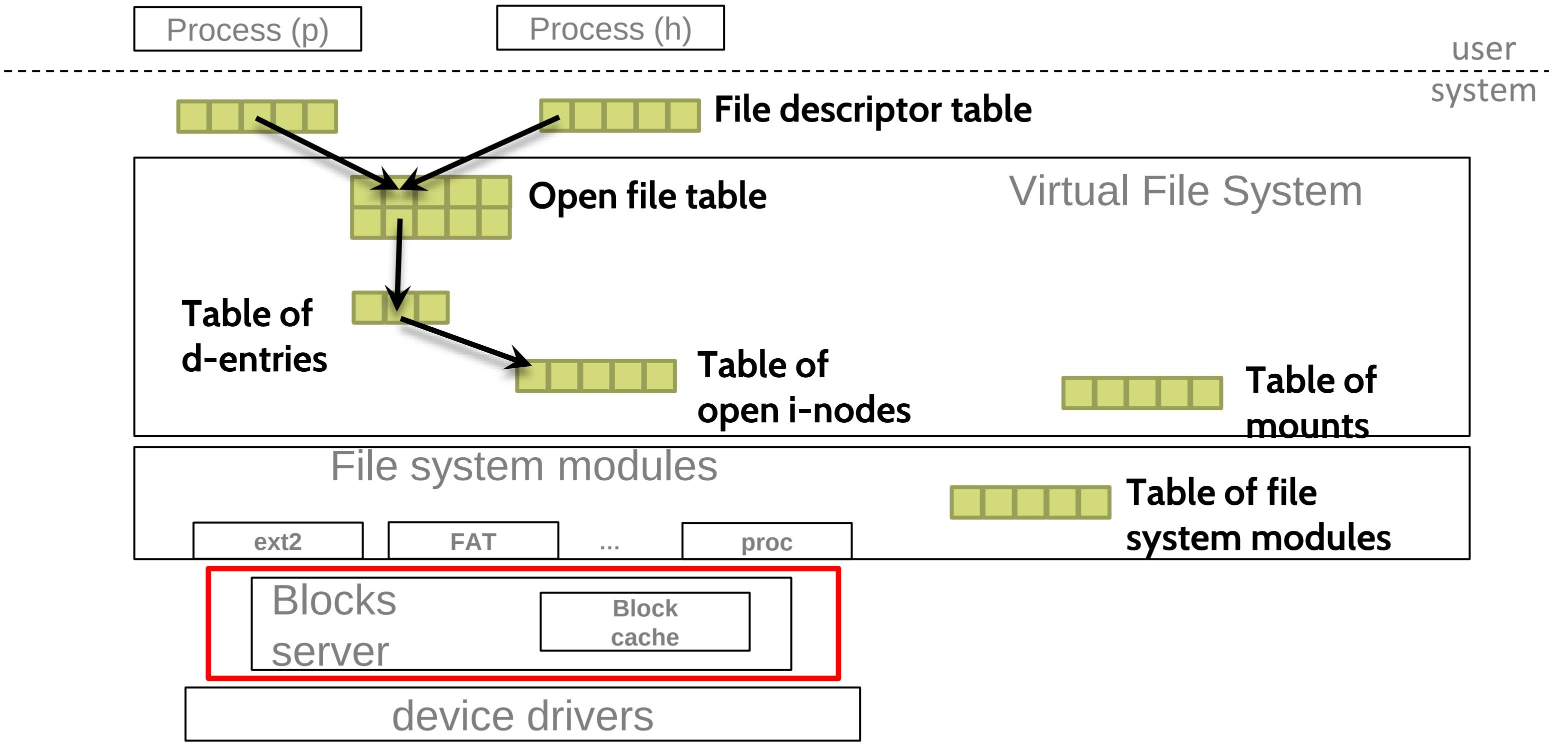
- The processes have to use a secure interface, without direct access to the kernel representation.
- To share the file offset among process from the same parent that open the file.
- To have a working session with the file/directory in order to update the information that it contains.
- Go back and forth in the file system directory tree.
- Offer persistency of user data, seeking to minimize the impact on the performance and the space needed for the metadata.
- Keep track of the file system registered in the kernel, and keep track of the mount points of these file systems.

Design and development of a file system

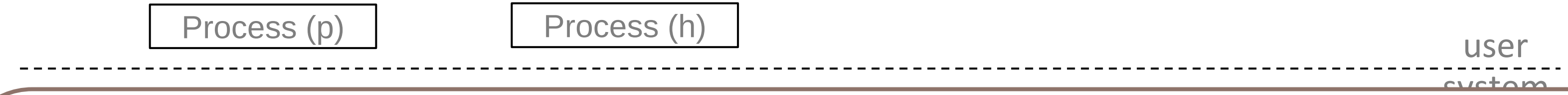


- File system requirements
- Main data structures in the secondary memory
- Main data structures in the main memory
- **Block management**
- Internal (and service) functions

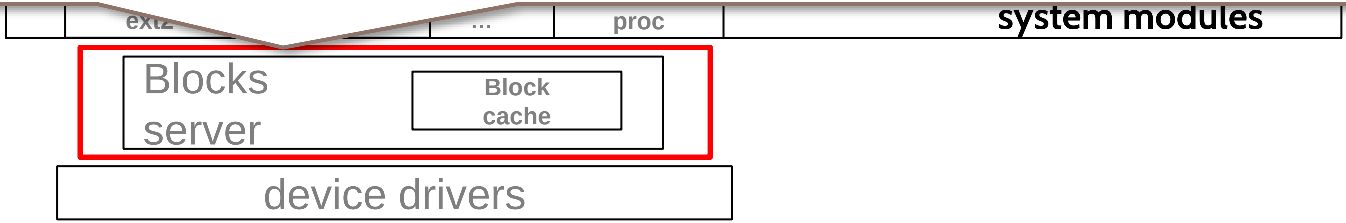
Main management structures



Main management structures



- **getblk**: find/reserve in cache a v-node block with its offset and size.
- **brelse**: to free a buffer and to insert it into the free list.
- **bwrite**: to write a cache block to the disk.
- **bread**: to read a disk block and store it in cache.
- **breada**: to read a block (and the following one) from disk to cache.



Block server

- It is responsible for:
  - Issuing commands to read and write device drivers blocks (by using the specific device routines)
  - Optimizing the I/O requests.
    - E.g.: Block cache.
  - Offering a logical device namespace.
    - E.g.: /dev/hda3 (third partition of the first disk)

Block server

- General behavior:
  - If the block is in the cache
    - Copy the content (and update the block usage metadata)
  - If it is not in the cache
    - To read the block from the device and store it in cache
    - To copy the content (and to update the block metadata)
    - If the block has been modified (*dirty*)
      - Cache write policy
    - If the cache is full, it is necessary get some free slots
      - Cache replacement policy

Block server

- General behavior:
  - If the block is in the cache

○ **Read-ahead**:
  ○ Read the following blocks into the cache (in order to improve the performance on sequential accesses)

  - To read the block from the device and store it in cache
  - To copy the content (and to update the block metadata)
  - If the block has been modified (*dirty*)
    - Cache write policy
  - If the cache is full, it is necessary get some free slots
    - Cache replacement policy

Block server

- General behavior:

- ○ **write-through**:
  - ○ Each time a block is modified it is also flushed to disk (lower performance)
- ○ **write-back**:
  - ○ The blocks are flushed to disk only when the block has to be evicted from the cache and it was dirty (better performance but reliability problems)
- ○ **delayed-write**:
  - ○ The modified blocks are saved to disk periodically (e.g., every 30 seconds in Unix) (trade-off for the former options)
- ○ **write-on-close**:
  - ○ When the file descriptor is closed, all file blocks are flushed to disk.

  - If the bloc... been modified (*dirty*)
    - Cache write policy
  - If the cache is full, it is necessary get some free slots
    - Cache replacement policy

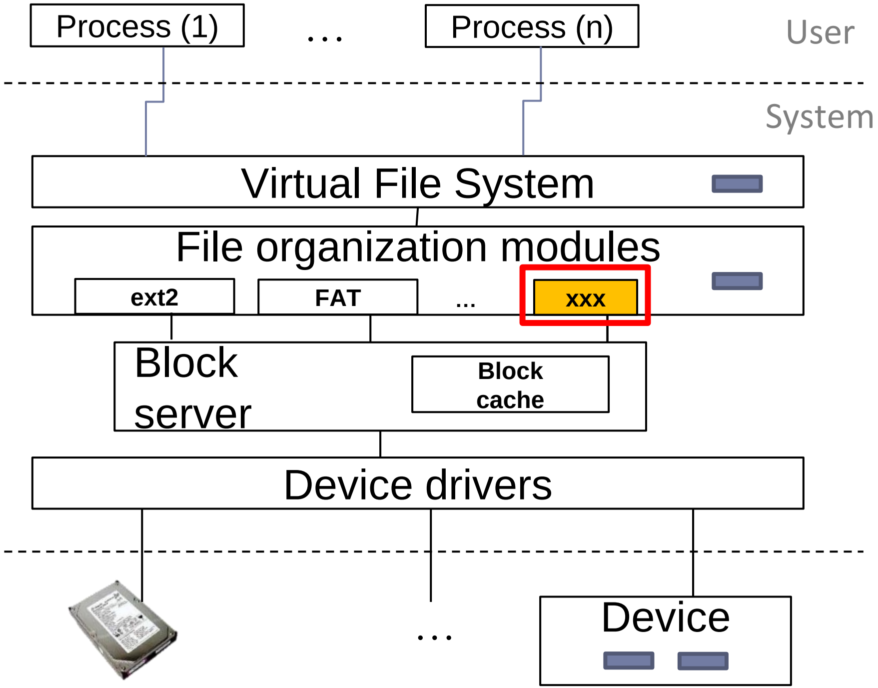
Block server

- General behavior:
  - If the block is in the cache
    - To copy the content (and to update the block usage metadata)
  - If it is not in the cache
    - To read the block from the device into the cache

- **FIFO** *(First in First Out)*
- **Clock algorithm** *(Second opportunity)*
- **MRU** *(Most Recently Used)*
- **LRU** *(Least Recently Used)*

    - If the cache is full, it is necessary get some free slots
      - Cache replacement policy

Design and development of a file system



- File system requirements
- Main data structures in the secondary memory
- Main data structures in the main memory
- Block management
- **Internal (and service) functions**

Example of management routines

File system syscalls

Descriptors	namei usage		i-node alloc.	File Attr.	I/O	File Sys.	View
open	open	stat	creat	chown	read	mount	chdir
creat	creat	link	mknod	chmod	write	umount	chroot
dup	chdir	unlink	link	stat	lseek		
pipe	chroot	mknod	unlink				
	chown	mount					
close	chmod	umount					

Low level algorithms of the file system

namei iget iput	ialloc ifree	alloc free	bmap

d-entries

montajes

file pointers

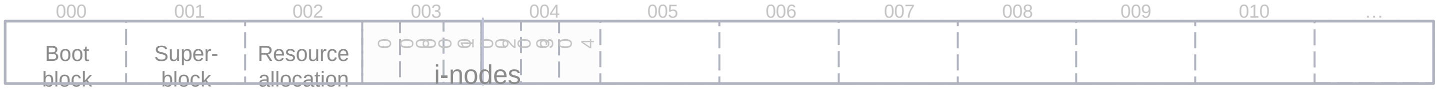
open files

in-use inodes

file system modules

Block/cache management algorithms

getblk	brelse	bread	breada	bwrite



Example of management routines

File system syscalls

Descriptors	namei usage		i-node alloc.	File Attr.	I/O	File Sys.	View
open creat dup pipe close	open creat chdir chroot chown chmod	stat link unlink mknod mount umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chroot

Low level algorithms of the file system

namei iget iput	ialloc ifree	alloc free	bmap

d-entries

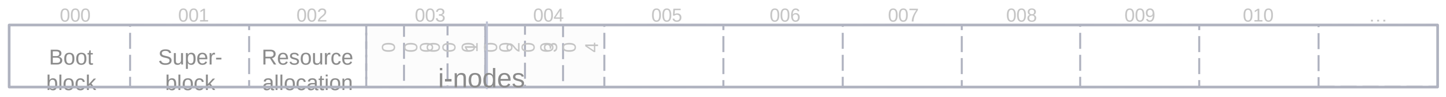
file pointers

open files

Block/cache ma

getblk brelse bwrite

1. Design the disk data structures & layout



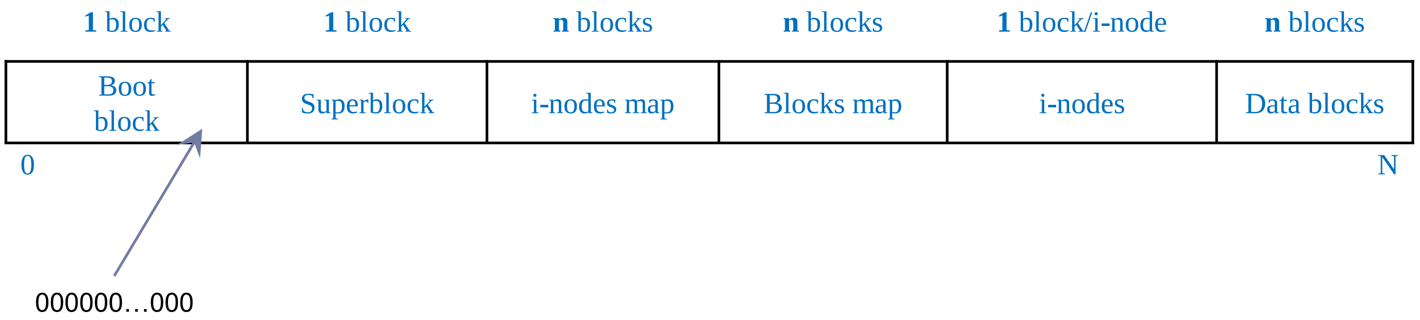
http://www.ual.es/~acorral/DSO/Tema_4.pdf

Example of disk organization

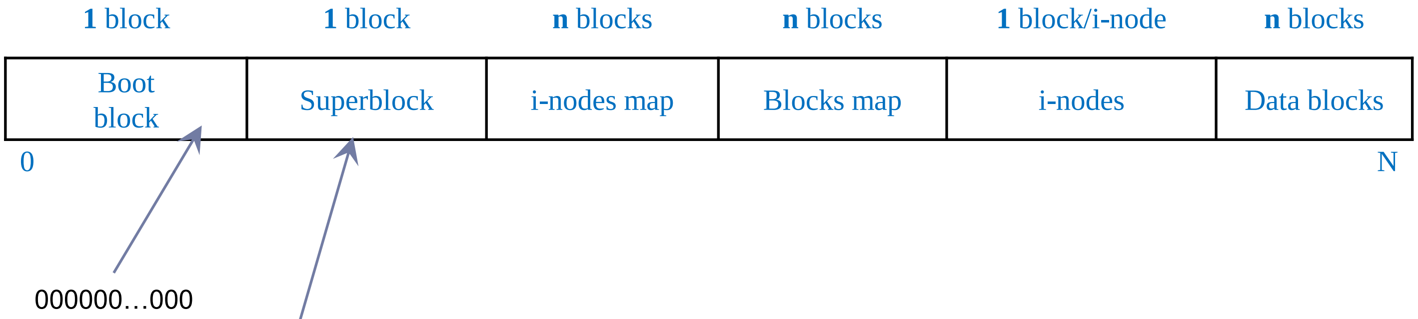
1 block	**1** block	**n** blocks	**n** blocks	**1** block/i-node	**n** blocks
Boot block	Superblock	i-nodes map	Blocks map	i-nodes	Data blocks

0 … N

Example of disk organization



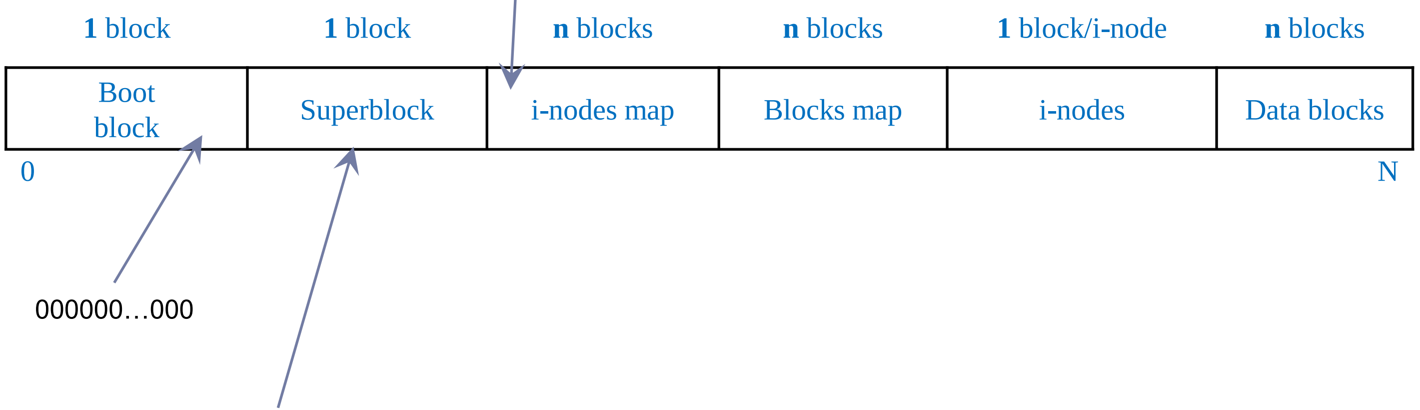
Example of disk organization




```
typedef struct {
    unsigned int magicNumber;             /* Supeblock magic number: 0x000D5500 */
    unsigned int numBlocksInodeMap;   /* Number of blocks of the inode map*/
    unsigned int numBlocksBlockMap;    /* Number of blocks of the data map */
    unsigned int numInodes;                  /* Number of inodes on the device */
    unsigned int firstInode;                 /* Block number of of the first inode on the device (root inode) */
    unsigned int numDataBlocks;           /* Number of data blocks on the device */
    unsigned int firstDataBlock;         /* Block number of the first block*/
    unsigned int deviceSize;                 /* Total device size in bytes*/
    char padding[992];                        /* Padding for filling a block */
} SuperblockType;
```

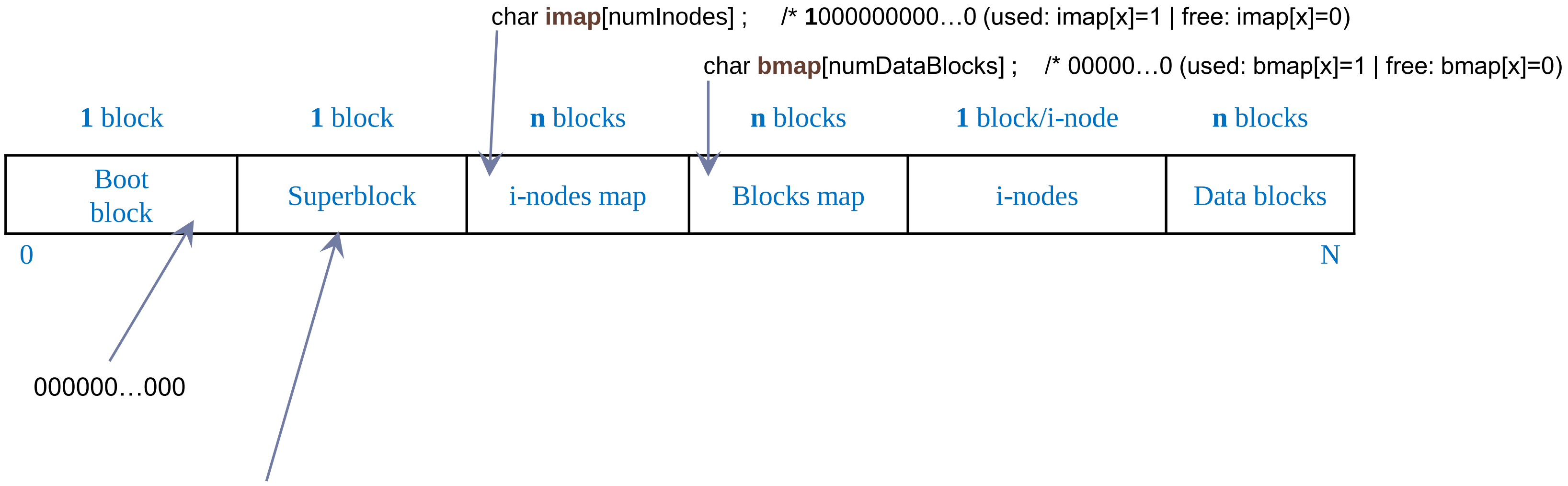
Example of disk organization

char **imap**[numInodes] ;   /* **1**000000000…0 (used: imap[x]=1 | free: imap[x]=0)




```
typedef struct {
    unsigned int magicNumber;           /* Supeblock magic number: 0x000D5500 */
    unsigned int numBlocksInodeMap;     /* Number of blocks of the inode map*/
    unsigned int numBlocksBlockMap;     /* Number of blocks of the data map */
    unsigned int numInodes;             /* Number of inodes on the device */
    unsigned int firstInode;            /* Block number of of the first inode on the device (root inode) */
    unsigned int numDataBlocks;         /* Number of data blocks on the device */
    unsigned int firstDataBlock;        /* Block number of the first block*/
    unsigned int deviceSize;            /* Total device size in bytes*/
    char padding[992];                  /* Padding for filling a block */
} SuperblockType;
```

Example of disk organization




```
typedef struct {
    unsigned int magicNumber;            /* Supeblock magic number: 0x000D5500 */
    unsigned int numBlocksInodeMap;      /* Number of blocks of the inode map*/
    unsigned int numBlocksBlockMap;      /* Number of blocks of the data map */
    unsigned int numInodes;              /* Number of inodes on the device */
    unsigned int firstInode;             /* Block number of of the first inode on the device (root inode) */
    unsigned int numDataBlocks;          /* Number of data blocks on the device */
    unsigned int firstDataBlock;         /* Block number of the first block*/
    unsigned int deviceSize;             /* Total device size in bytes*/
    char padding[992];                   /* Padding for filling a block */
} SuperblockType;
```

Example of disk organization

char **imap**[numInodes] ; /* 1000000000…0 (used: imap[x]=1 | free: imap[x]=0)

char **bmap**[numDataBlocks] ; /* 00000…0 (used: bmap[x]=1 | free: bmap[x]=0)

1 block	1 block	n blocks	n blocks	1 block/i-node	n blocks
Boot block	Superblock	i-nodes map	Blocks map	i-nodes	Data blocks

0 … N

000000…000

```
typedef struct {
    unsigned int type;                  /*  T_FILE o T_DIRECTORY */
    char name[200];                     /* name of the associated file/directory*/
    unsigned int inodeTable[200];   /* type==dir: list of inodes from the directory */
    unsigned int size;                  /* File size in bytes */
    unsigned int directBlock;           /* Direct block number */
    unsigned int indirectBlock;         /* Indirect block number */
    char padding[PADDING_INODO];          /* Padding for filling a block */
} InodeDiskType;
```

```
typedef struct {
    unsigned int magicNumber;            /* Supeblock magic number: 0x000D5500 */
    unsigned int numBlocksInodeMap;   /* Number of blocks of the inode map*/
    unsigned int numBlocksBlockMap;    /* Number of blocks of the data map */
    unsigned int numInodes;               /* Number of inodes on the device */
    unsigned int firstInode;              /* Block number of of the first inode on the device (root inode) */
    unsigned int numDataBlocks;         /* Number of data blocks on the device */
    unsigned int firstDataBlock;      /* Block number of the first block*/
    unsigned int deviceSize;              /* Total device size in bytes*/
    char padding[992];                    /* Padding for filling a block */
} SuperblockType;
```

Example of management routines

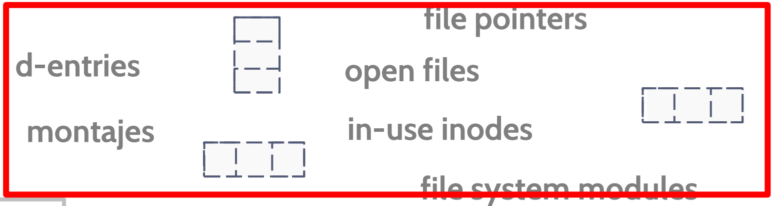
File system syscalls

Descriptors	namei usage		i-node alloc.	File Attr.	I/O	File Sys.	View
open	open	stat	creat	chown	read	mount	chdir
			link	stat			
pipe	chown	mount	unlink				
close	chmod	umount					

2. Design in-memory data structures

Low level algorithms of the file system

namei iget iput	ialloc ifree	alloc free	bmap



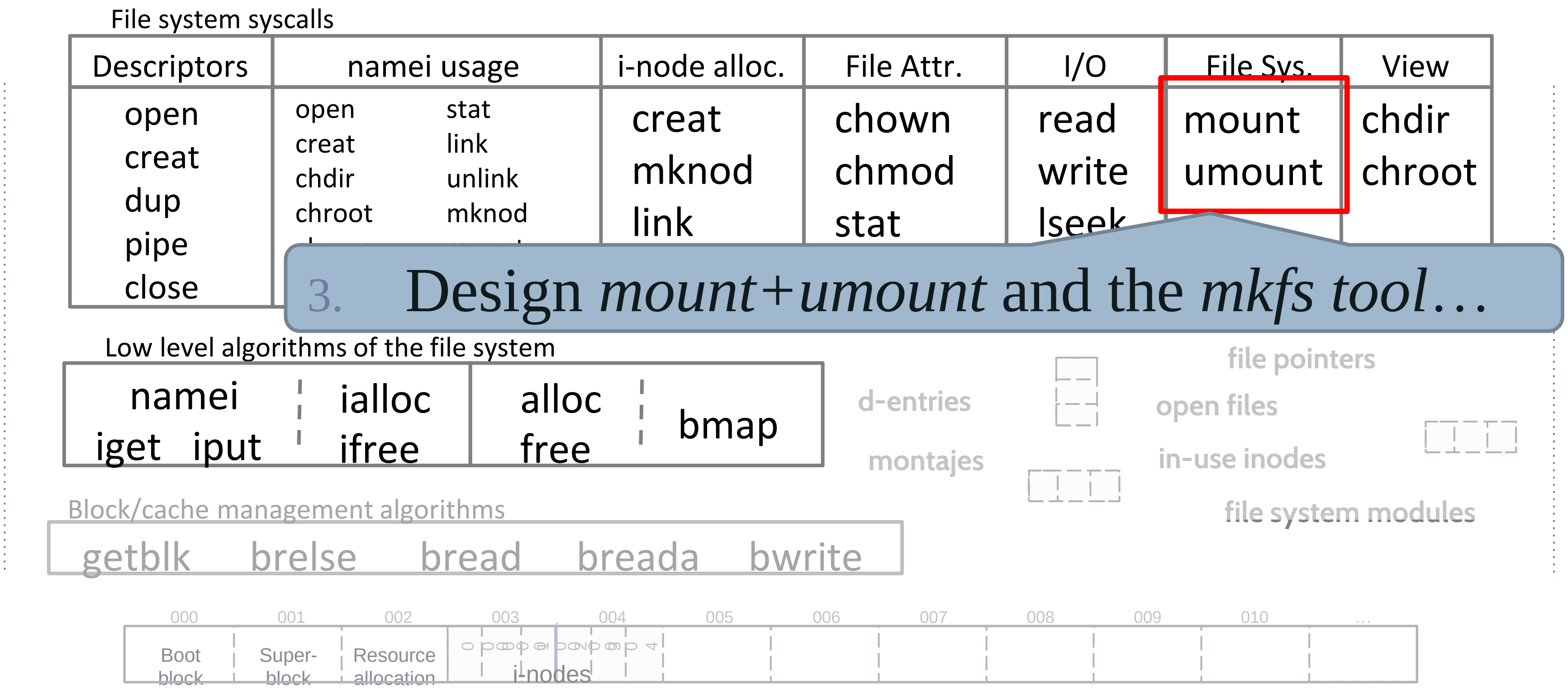
Block/cache management algorithms

getblk brelse bread breada bwrite

Example of variables…

```
// Information read from disk
SuperblockType sBlocks [1] ;
char imap [numInodes] ;
char dbmap [numDataBlocks] ;
InodeDiskType inodos [numInodes] ;

// Additional in-memory Information
struct {
    int file_pointer;
    int open;
} inmemory_inode_table [numInodes] ;

…
```

Example of management routines



http://www.ual.es/~acorral/DSO/Tema_4.pdf

Example: mount

```
int mount ( void )
{
    // To read 0 block from disk into sBlocks[0]
    bread(DISK, 0, &(sBlocks[0]) );

    // To read the i-node map from disk
    for (int=0; i<sBlocks[0].numBlocksInodeMap; i++)
          bread(DISK, 1+i, ((char *)imap + i*BLOCK_SIZE) ;

    // To read the block map from disk
    for (int=0; i<sBlocks[0].numBlocksBlockMap; i++)
         bread(DISK, 1+i+sBlocks[0].numBlocksInodeMap, ((char *)dbmap + i*BLOCK_SIZE);

    // To read the i-nodes to main memory
    for (int=0; i<(sBlocks[0].numInodes*sizeof(InodeDiskType)/BLOCK_SIZE); i++)
         bread(DISK, i+sBlocks[0].firstInode, ((char *)inodes + i*BLOCK_SIZE);

    return 1;
}
```

Example: umount

```
int umount ( void )
{
  // To write block 0 from sBlocks[0] into disk
  bwrite(DISK, 0, &(sBlocks[0]) );

  // To write the i-node map to disk
  for (int=0; i<sBlocks[0].numBlocksInodeMap; i++)
       bwrite(DISK, 1+i, ((char *)imap + i*BLOCK_SIZE) ;

  // To write the block map to disk
  for (int=0; i<sBlocks[0].numBlocksBlockMap; i++)
      bwrite(DISK, 1+i+sBlocks[0].numBlocksInodeMap, ((char *)dbmap + i*BLOCK_SIZE);

  // To write the i-nodes to disk
  for (int=0; i<(sBlocks[0].numInodes*sizeof(InodeDiskType)/BLOCK_SIZE); i++)
      bwrite(DISK, i+sBlocks[0].firstInode, ((char *)inodes + i*BLOCK_SIZE);

  return 1;
}
```


Alejandro Calderón Mateos

Example: mkfs

```
int mkfs ( void )
{
    // setup with default values the superblock, maps, and i-nodes
    sBlocks[0].magicNumber = 1234;
    sBlocks[0].numInodes = 50;
    …
    for (int=0; i<sBlocks[0].numInodes; i++)
          imap[i] = 0; // free
    for (int=0; i<sBlocks[0].numDataBlocks; i++)
          bmap[i] = 0; // free
    for (int=0; i<sBlocks[0].numInodes; i++)
         memset(&(inodos[i]), 0, sizeof(InodeDiskType) );

    // to write the default file system into disk
    umount();

    return 1;
}
```

Example of management routines

File system syscalls

Descriptors	namei usage		i-node alloc.	File Attr.	I/O	File Sys.	View
open	open	stat	creat	chown	read	mount	chdir
creat	creat	link	mknod	chmod	write	umount	chroot
dup	chdir	unlink	link	stat	lseek		
pipe	chroot	mknod	unlink				
close	chown	mount					
	chmod	umount					

Low level algorithms of the file system

namei / iget iput	ialloc / ifree	alloc / free	bmap

d-entries

montajes

file pointers

open files

in-use inodes

file system modules

Block/cache management al...

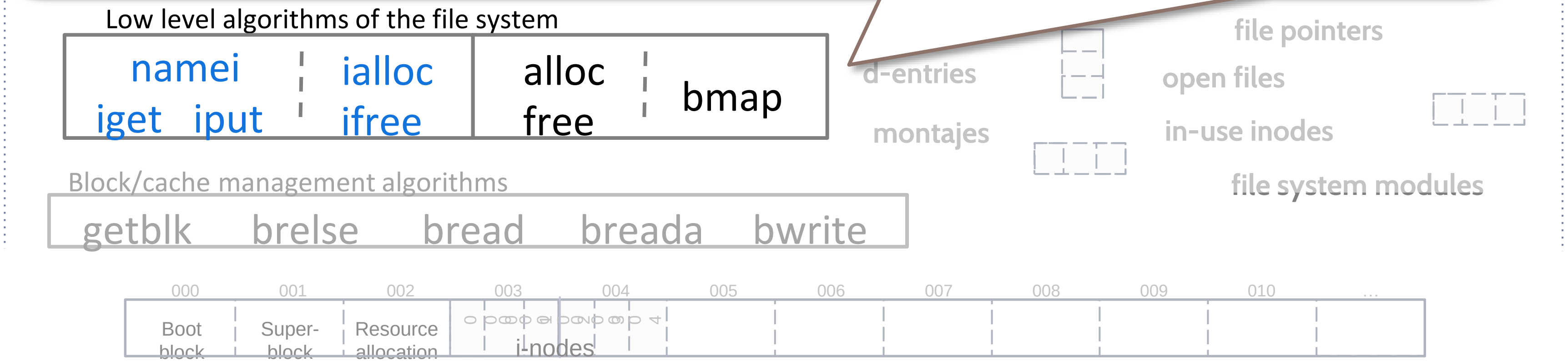
4. To design the (internal) management routines
   - Read/write to/from disk into the in-memory data structures

http://www.ual.es/~acorral/DSO/Tema_4.pdf

Alejandro Calderón Mateos

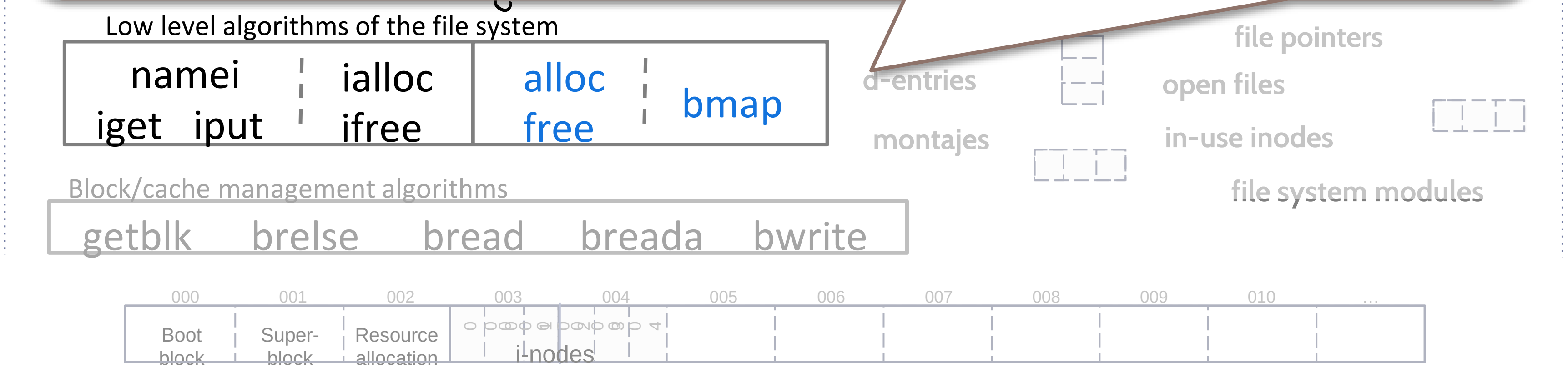
Management routines

- **namei**: convert the full path into the associated i-node.
- **iget**: return a i-node (from the i-node table), and it can read from secondary memory, into a free element form the i-node table.
- **iput**: free an i-node from the i-node table, and if it is necessary then to update in secondary memory.
- **ialloc**: allocate an i-node for a file.
- **ifree**: free an i-node previously allocated for the file.



Management routines

- **bmap**: to compute the disk block associated with a given file offset. Translate logical address (file offset) into physical address (disk block).
- **alloc**: to allocate a free block for the file.
- **free**: to free a previously allocated block.



http://www.buet.ac.bd/iict/iictcourses/ict6005/lecture9.ppt

Example: ialloc y alloc

```
int ialloc ( void )
{
    // to search for a free i-node
    for (int=0; i<sBlocks[0].numInodes; i++)
    {
        if (imap[i] == 0) {
            // i-node busy right now
            imap[i] = 1;
            // default values for the i-node
            memset(&(inodes[i]),0,
                   sizeof(InodeDiskType));
            // return the i-node indentification
            return i;
        }
    }

    return -1;
}
```

```
int alloc ( void )
{
    char b[BLOCK_SIZE];

    for (int=0; i<sBlocks[0].numDataBlocks; i++)
    {
        if (bmap[i] == 0) {
            // busy block right now
            bmap[i] = 1;
            // default values for the block
            memset(b, 0, BLOCK_SIZE);
            bwrite(DISK, i, b);
            // it returns the block id
            return i;
        }
    }
    return -1;
}
```


http://lsi.ugr.es/~jlgarrid/so2/pdf/tem2-1-2.pdf

Example: ifree y free

```
int ifree ( int inode_id )
{
    // to check the inode_id vality
    if (inode_id > sBlocks[0].numInodes)
        return -1;

    // free i-node
    imap[inode_id] = 0;

    return -1;
}
```

```
int free ( int block_id )
{
    // to check inode_id the vality
    if (block_id > sBlocks[0].numDataBlocks)
        return -1;

    // free block
    bmap[block_id] = 0;

    return -1;
}
```


http://mycsvtunotes.weebly.com/uploads/1/0/1/7/10174835/unix_unit4.pdf

Example: namei y bmap

```
int namei ( char *fname )
{
   // search the inode with name fname
   for (int=0; i<sBlocks[0].numInodes; i++)
   {
         if (! strcmp(inodos[i].name, fname))
               return i;
   }

    return -1;
}
```

```
int bmap ( int inode_id, int offset )
{
   int b[BLOCK_SIZE/4];

   // check the validity of inode_id
   if (inode_id > sBlocks[0].numInodes)
      return -1;

   // find the associated data block
   if (offset < BLOCK_SIZE)
      return inodos[inode_id].directBlock;
   if (offset < BLOCK_SIZE*BLOCK_SIZE/4) {
       bread(DISK, inodos[inode_id].indirectBlock, b);
       offset = (offset – BLOCK_SIZE) / BLOCK_SIZE;
       return b[offset] ;
   }

   return -1;
}
```

Example of management routines

File system syscalls

Descriptors	namei usage		i-node alloc.	File Attr.	I/O	File Sys.	View
open	open	stat	creat	chown	read	mount	chdir
creat	creat	link	mknod	chmod	write	umount	chroot
dup	chdir	unlink	link	stat	lseek		
pipe	chroot	mknod	unlink				
close	chown	mount					
	chmod	umount					

Low level algorithms of the file system

namei

iget

file pointers

5. Develop the file system routines for system calls

Block/c

getblk brelse bread breada bwrite

000 001 002 003 004 005 006 007 008 009 010 ...

Boot block | Super-block | Resource allocation | i-nodes

http://www.ual.es/~acorral/DSO/Tema_4.pdf

Management routines

File system syscalls

descriptore	namei usage		i-nodes alloc.	File Attr.	I/O	File Sys.	View
open creat dup pipe close	open creat chdir chroot chown chmod	stat link unlink mknod mount umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chroot

Low level algorithms of the file system

namei iget iput	ialloc ifree	alloc free	bmap

d-entries
file pointers
en files
inodes
mo

- **open**: find the associated i-node for the file name, ...
- **read**: find the data block, read the data block, ...
- **write**: find the data block, write the data block, ...
- ...

http://mycsvtunotes.weebly.com/uploads/1/0/1/7/10174835/unix_unit4.pdf

Example: open y close

```
int open ( char *name )
{
    int inode_id ;

    inode_id = namei(name) ;
    if (inode_id < 0)
        return inode_id ;

    inmemory_inode_table[inode_id].file_pointer = 0;
    inmemory_inode_table[inode_id].open   = 1;

    return inode_id;
}
```

```
int close ( int fd )
{
     if (fd < 0)
         return fd ;

     inmemory_inode_table[fd].file_pointer = 0;
     inmemory_inode_table[fd].open   = 0;

     return 1;
}
```


http://mycsvtunotes.weebly.com/uploads/1/0/1/7/10174835/unix_unit4.pdf

Example: creat y unlink

```
int creat ( char *name )
{
    int b_id, inode_id ;

    inode_id = ialloc() ;
    if (inode_id < 0) { return -1 ; }
    b_id = alloc();
    if (b_id < 0) { ifree(inode_id); return b_id ; }

    inodos[inode_id].tipo = 1 ; // FILE
    strcpy(inodos[inode_id].name, name);
    inodos[inode_id].directBlock = b_id ;
    inmemory_inode_table[inode_id].file_pointer = 0;
    inmemory_inode_table[inode_id].open   = 1;

    return 1;
}
```

```
int unlink ( char * name )
{
    int inode_id ;

    inode_id = namei(name) ;
    if (inode_id < 0)
        return inode_id ;

    free(inodos[inode_id].directBlock);
    memset(&(inodes[inode_id]),
                0,
                sizeof(InodeDiskType));
    ifree(inode_id) ;

   return 1;
}
```


http://mycsvtunotes.weebly.com/uploads/1/0/1/7/10174835/unix_unit4.pdf

Example: read y write

```
int read ( int fd, char *buffer, int size )
{
   char b[BLOCK_SIZE] ;
   int b_id ;

   if (inmemory_inode_table[fd].file_pointer+size > inodos[fd].size)
       size = inodos[fd].size - inmemory_inode_table[fd].file_pointer;
   if (size <= 0)
       return 0;

   b_id = bmap(fd, inmemory_inode_table[fd].file_pointer);
   bread(DISK, b_id, b);
   memmove(buffer,
                  b+inmemory_inode_table[fd].file_pointer,
                  size);
   inmemory_inode_table[fd].file_pointer += size;

   return 1;
}
```

```
int write ( int fd, char *buffer, int size )
{
   char b[BLOCK_SIZE] ;
   int b_id ;

   if (inmemory_inode_table[fd].file_pointer+size > BLOCK_SIZE)
       size = BLOCK_SIZE - inmemory_inode_table[fd].file_pointer;
   if (size <= 0)
       return 0;

   b_id = bmap(fd, inmemory_inode_table[fd].file_pointer);
   bread(DISK, b_id, b);
   memmove(b+inmemory_inode_table[fd].file_pointer,
                  buffer, size);
   bwrite(DISK, b_id, b);
   inmemory_inode_table[fd].file_pointer += size;

   return 1;
}
```


http://mycsvtunotes.weebly.com/uploads/1/0/1/7/10174835/unix_unit4.pdf

Management routines

summary

File system syscalls

descriptore	namei usage		i-nodes alloc.	File Attr.	I/O	File Sys.	View
open	open	stat	creat	chown	read	mount	chdir
creat	creat	link	mknod	chmod	write	umount	chroot
dup	chdir	unlink	link	stat	lseek		
pipe	chroot	mknod	unlink				
close	chown	mount					
	chmod	umount					

Low level algorithms of the file system

namei iget iput	ialloc ifree	alloc free	bmap

d-entries

montajes

file pointers

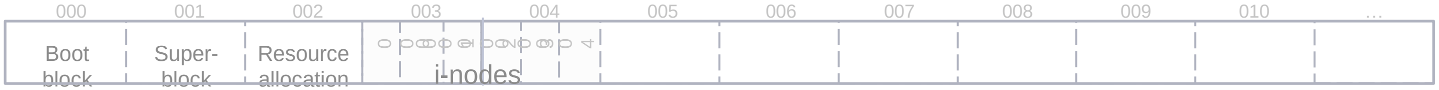
open files

in-use inodes

file system modules

Block/cache management algorithms

getblk	brelse	bread	breada	bwrite



http://www.ual.es/~acorral/DSO/Tema_4.pdf

Alejandro Calderón Mateos

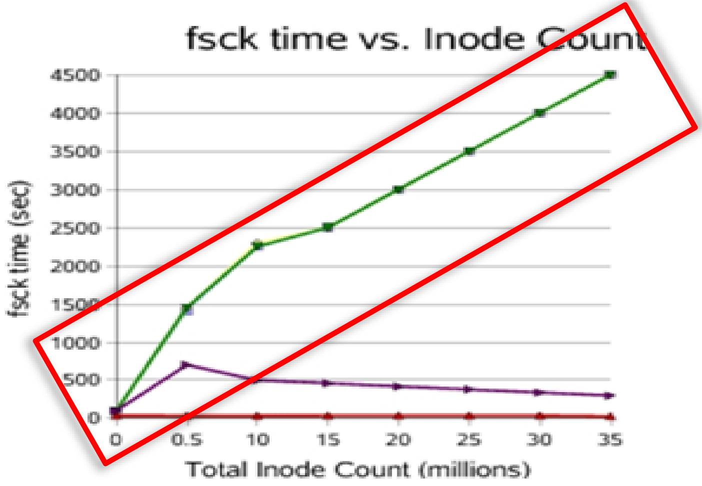
Overview

1. Introduction
2. File system internals and framework
3. Design and development of a file system
4. **Complementary aspects**

Advanced features

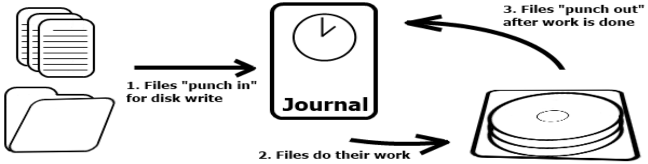
- Journaling
- Snapshots
- Dynamic file system expansion

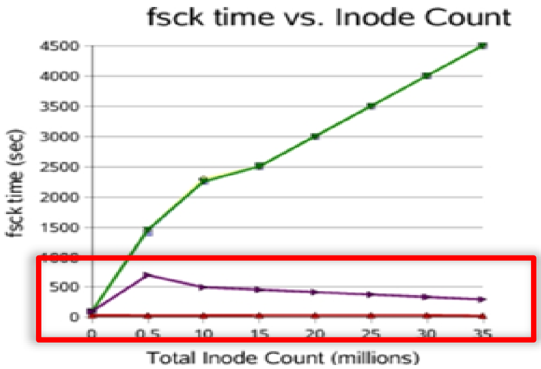
Without *Journaling*



- If the computer is shut down abruptly, the file system might remain be inconsistent.
- In order to repair the file system, all metadata has to be reviewed:
  - The required time depends of the file system size (all the metadata has to be reviewed, the more metadata to be reviewed the more time is needed).

With *Journaling*





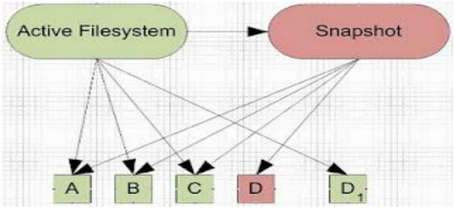
- The file system writes the changes in a log before changing the file.
- If the computer is shut down abruptly, the file system checks has to review the log for the pending changes, and do these changes (commit):
  - The time needed depends of the number of pending changes in the log, and does not depend on the file system size.
  - From hours to seconds…

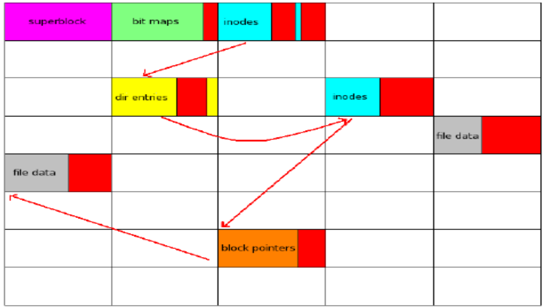
http://www.howtogeek.com/howto/33552/htg-explains-which-linux-file-system-should-you-choose/

Advanced features

- Journaling
- Snapshots
- Dynamic file system expansion

Snapshot





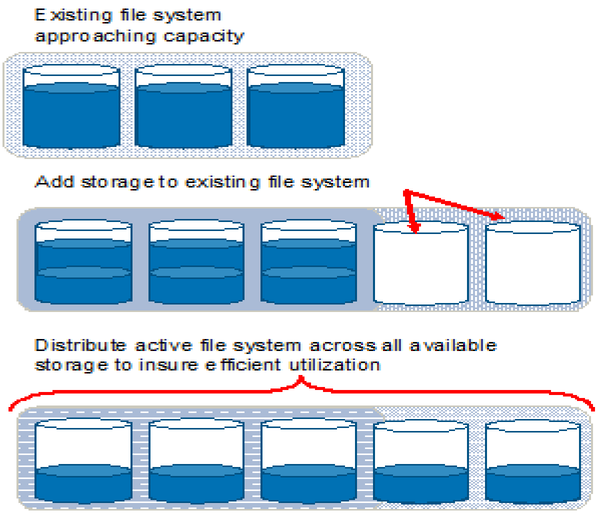
- A Snapshot represents the state of the file system at a point of time:
  - In a few seconds is done.
  - It is possible to access to all the file system snapshots on this disk.
- E.g.: system updates, backups, etc.

Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS… (Página 139)
http://lwn.net/Articles/342892/, http://sannasanalyst.blogspot.com.es/2010_06_01_archive.html

Advanced features

- Journaling
- Snapshots
- Dynamic file system expansion

Dynamic file system expansion



- It is important to design the file system in a way that it could be resized (add more space, remove space, etc.) without losing information.
  - Dynamic and flexible structures
  - Metadata is distributed along the disk

http://www.storagenetworks.com/documents/productdatasheets/x9000_nas_proposal.htm

ARCOS Group
Computer Science and Engineering Department
Universidad Carlos III de Madrid

Lesson 5
File Systems

Operating System Design
Degree in Computer Science and Engineering, Double Degree CS&E + BA