<Slides download>
http://www.pf.is.s.u-tokyo.ac.jp/class.html

Advanced Operating Systems

#14

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1.	2018.9.28	Introduction + Linux Kernel (Kato)
2.	2018.10.5	Linux Kernel (Chishiro)
3.	2018.10.12	Linux Kernel (Kato)
4.	2018.10.19	Linux Kernel (Kato)
5.	2018.10.26	Linux Kernel (Kato)
6.	2018.11.2	Advanced Research (Chishiro)
7.	2018.11.9	Advanced Research (Chishiro)
8.	2018.11.16	(No Class)
9.	2018.11.23	(Holiday)
10.	2018.11.30	Advanced Research (Chishiro)
11.	2018.12.7	Advanced Research (Kato)
12.	2019.12.14	(No Class)
13.	2018.12.21	Advanced Research (Kato)
14.	2019.1.11	(No Class)
15.	2019.1.18	10:25-12:10 Linux Kernel
16.	2019.1.25	13:00-14:45 Linux Kernel

File Systems

Abstracting Files – Virtual File Systems (VFS)

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

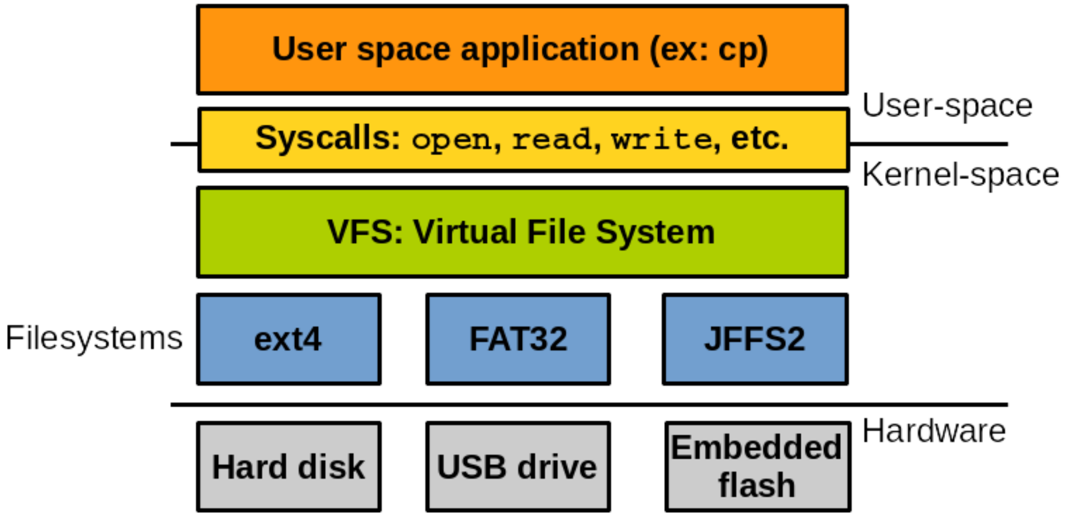
Outline

Outline

General Presentation

Generalities

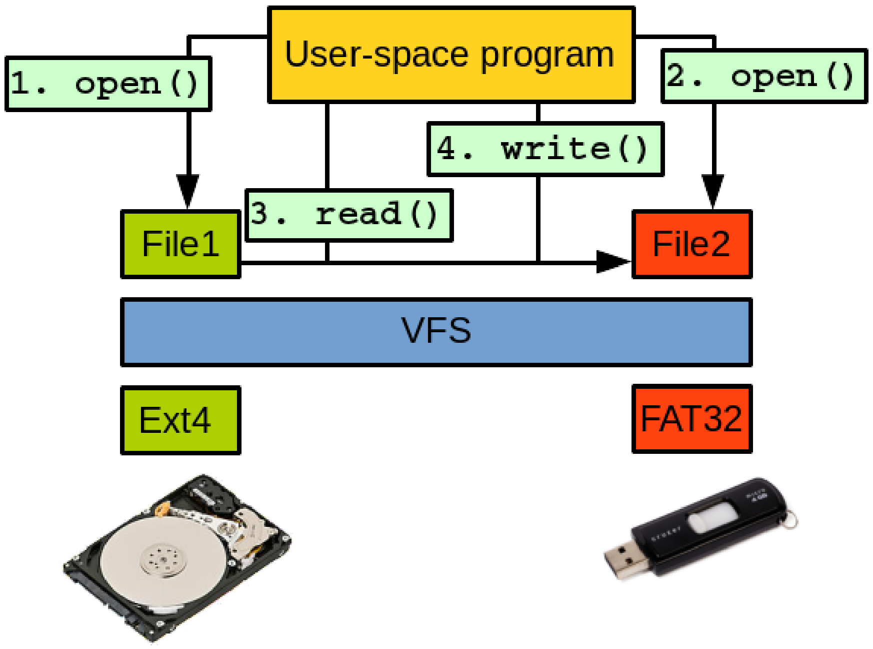
- **The Virtual File System (VFS)**
  - Abstracts all the filesystems models supported by Linux
  - Allow them to *coexist*
    - Example: user can have a USB drive formatted with FAT32 mounted at the same time as a HDD rootfs with ext4
  - Allow them to *cooperate*
    - Example: user can seamlessly copy a file between the FAT32 and Ext4 partitions



General Presentation

Common filesystem interface

- VFS allows user-space to access files **independently** of the concrete filesystem they are stored on, with a **common interface**:
  - Standard system calls: `open()`, `read()`, `write()`, `lseek()`, etc.
  - ”top” VFS interface (with user-space)
- Interface can work transparently between filesystems

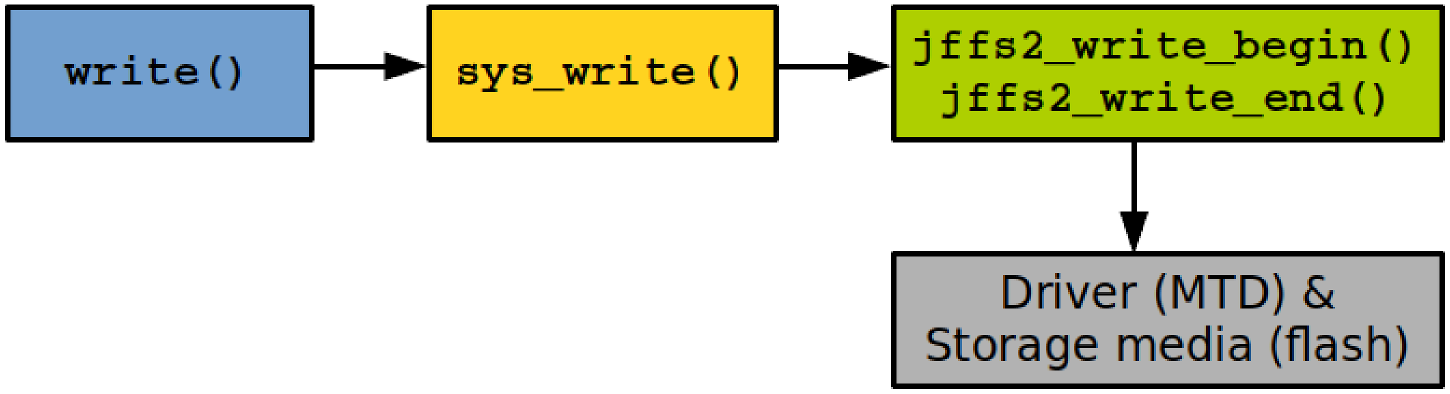


- `fd = open(path, flags)`
- `bytes written = write(fd, buf, count)`
- `bytes read = read(fd, buf, count)`
- `offset = lseek(fd, offset, whence)`
- `stat(path, struct stat ptr)`
- etc.

General Presentation

Filesystem abstraction layer

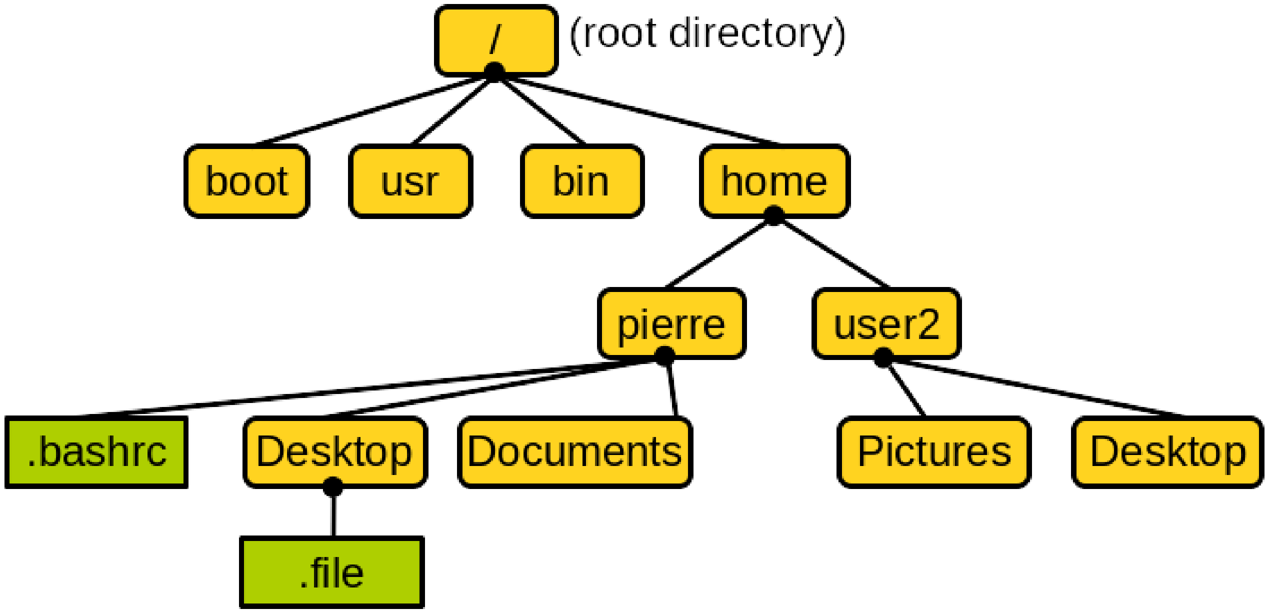
- VFS redirect user-space requests to the corresponding concrete filesystem
  - ”bottom” VFS interface (with the filesystem)
  - Developing a new filesystem for Linux means **conforming** with the bottom interface



General Presentation

Unix filesystems

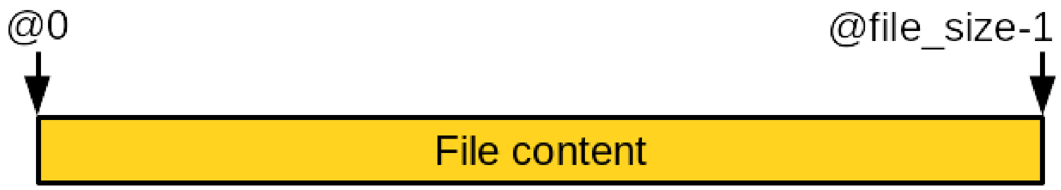
- The term *filesystem* can refer to a filesystem type or a partition
- Hierarchical tree of *files* organized into *directories*



General Presentation

Unix filesystems (2)

- **File**:
  - Ordered string of bytes from file address 0 to address (file size -1)



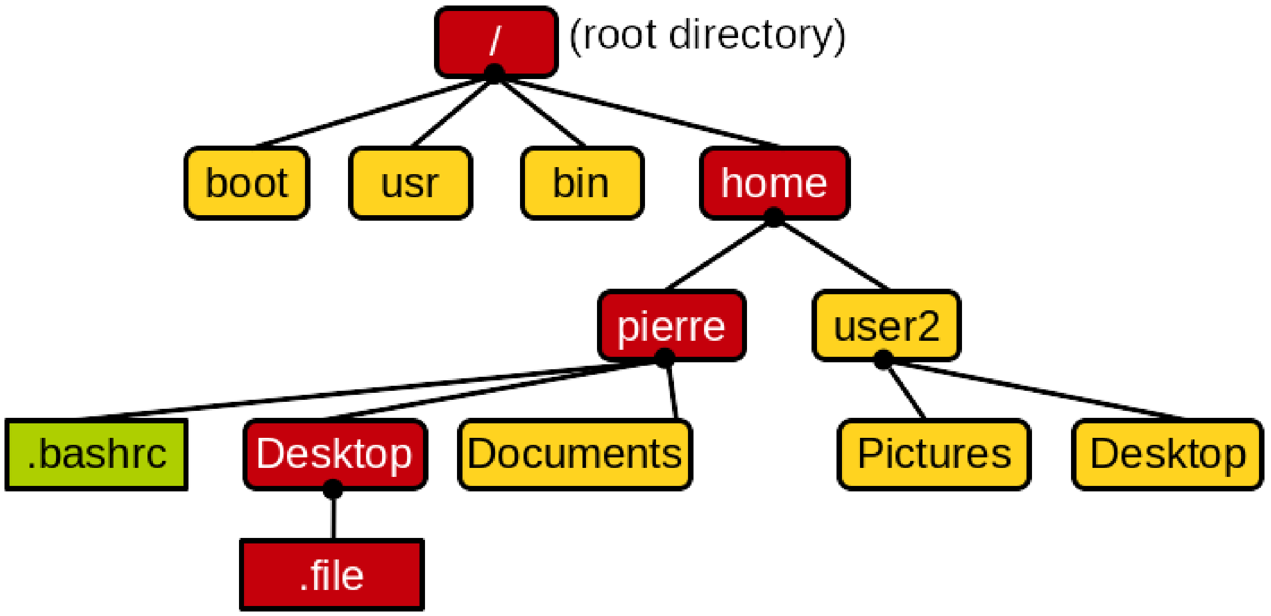
  - Associated metadata: name, access permissions, modification date, etc.
    - Separated from the file data into specific objects (*inodes*, *dentries*)
- **Directory**:
  - Folder containing files or other directories (sub-directories)
  - Sub-directories can be nested to create path:

```
/home/pierre/Desktop/file
```

General Presentation

Unix filesystems (3)

- Path example: `/home/pierre/Desktop/file`:



Outline

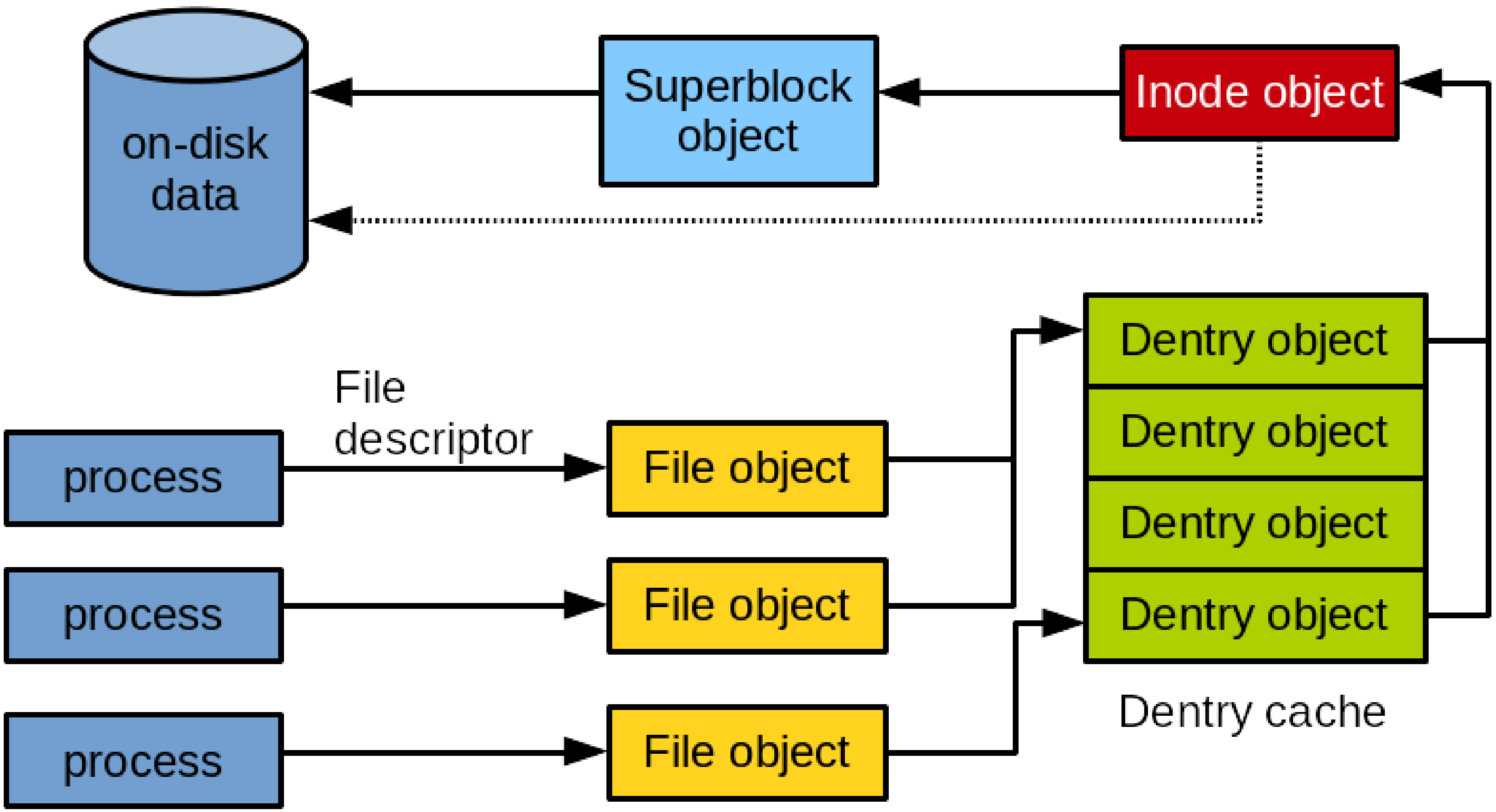
VFS data structures

Generalities

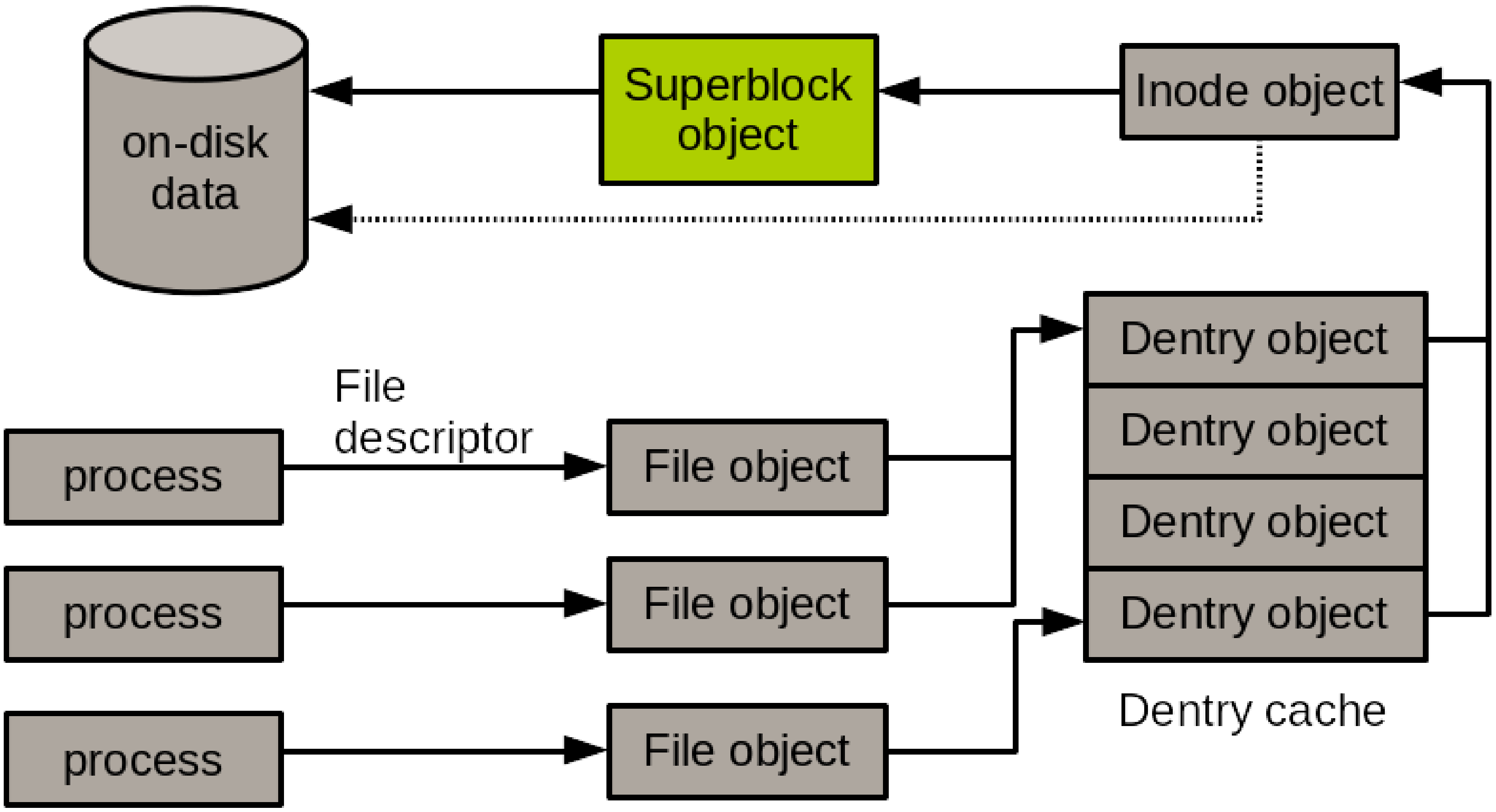
- **`inode`**: contains file/directory metadata
- **`dentry`**: contains file/directory name and hierarchical links defining the filesystem directory tree
- **`superblock`**: contains general information about the partition
- **`file`**: contains information about a file opened by a process
- Associated **operations**:
  - `super_operations`, `inode_operations`, `dentry_operations`, `file_operations`
  - Data structures containing function pointers
- **VFS implemented in a manner very close to object-oriented programming**

VFS data structures

Generalities (2)



VFS data structures



VFS data structures

Superblock object

- **Superblock**: contains global information about the filesystem (partition)
- Created by the filesystem and given to VFS at mount time:
  - Disk-based filesystem store it in a special location
  - Other filesystems have a way to generate it at mount time
- **`struct super_block`** defined in `include/ linux/fs.h`
  - Some fields:

```
struct super_block {
  struct list_head  s_list;          /* list of all superblocks */
  dev_t             s_dev;           /* identifier */
  unsigned long     s_blocksize;     /* block size (bytes) */
  unsigned long     s_blocksize_bits; /* block size (bits) */
  loff_t            s_maxbytes;      /* max file size */
  /* ... */
```

VFS data structures

Superblock object (2)

```c
    /* ... */
    struct file_system_type     *s_type;           /* filesystem type */
    struct super_operations     *s_op;             /* superblock operations */
    struct dquot_operations     *dq_op;            /* quota methods */
    struct quotactl_ops         *s_qcop;           /* quota control methods */
    unsigned long               s_flags;           /* mount flags */
    unsigned long               s_magic;           /* filesystem magic number */
    struct dentry               s_root;            /* directory mount point */
    struct rw_semaphore         s_umount;          /* umount semaphore */
    int                         s_count;           /* superblock reference count */
    atomic_t                    s_active;          /* active reference count */
    struct xattr_handler        **s_xattr;         /* extended attributes handler */
    struct list_head            s_inodes;          /* inodes list */
    struct hlist_bl_head        s_anon;            /* anonymous entries */
    struct list_lru             s_dentry_lru;      /* list of unused dentries */
    struct block_device         *s_bdev;           /* associated block device */
    struct mtd_info             *s_mtd;            /* embedded flash information */
    struct hlist_node           s_instances;       /* instances of this filesystem */
    struct quota_info           s_dquot;           /* quota-specific options */
    char                        s_id[32];          /* text name */
    void                        *s_fs_info;        /* filesystem-specific info */
    fmode_t                     s_mode;            /* mount permissions */
    /* ... */
}
```

VFS data structures

Superblock operations

- **`struct super_operations`**
  - Each field is a function pointer operating on a `struct super_block`
  - Usage: `sb->s_op->write_super(sb);`
  - C++ OOP equivalent would be `sb.write_super();`
- `include/linux/fs.h`:

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *, int flags);
  int (*write_inode) (struct inode *, struct writeback_control *wbc);
  int (*drop_inode) (struct inode *);
  void (*evict_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  /* ... */
}
```

VFS data structures

Superblock operations (2)

- `struct inode * alloc_inode(struct super_block *sb)`
  - Creates and initialize a new inode
- `void destroy_inode(struct inode *inode)`
  - Deallocate an inode
- `void dirty_inode(struct inode *inode)`
  - Marks an inode as dirty (Ext filesystems)
- `void write_inode(struct inode *inode, int wait)`
  - Writes the inode to disk, `wait` specifies if the write should be synchronous
- `void drop_inode(struct inode *inode)`
  - Called by VFS when the last reference to the inode is dropped
- `void put_super(struct super_block *sb)`
  - Called by VFS on unmount (holding `s_lock`)

VFS data structures

Superblock operations (3)

- `void write_super(struct super_block *sb)`
  - Update the on-disk superblock, caller must hold `s_lock`
- `int sync_fs(struct super_block *sb, int wait)`
  - Synchronize filesystem metadata with on-disk filesystem, `wait` specifies if the operation should be synchronous
- `void write_super_lockfs(struct super_block *sb)`
  - Prevents changes to the filesystem and update the on-disk superblock (used by the Logical Volume Manager)
- `void unlockfs(struct super_block *sb)`
  - Unlocks the filesystem locked by `write_super_lockfs()`

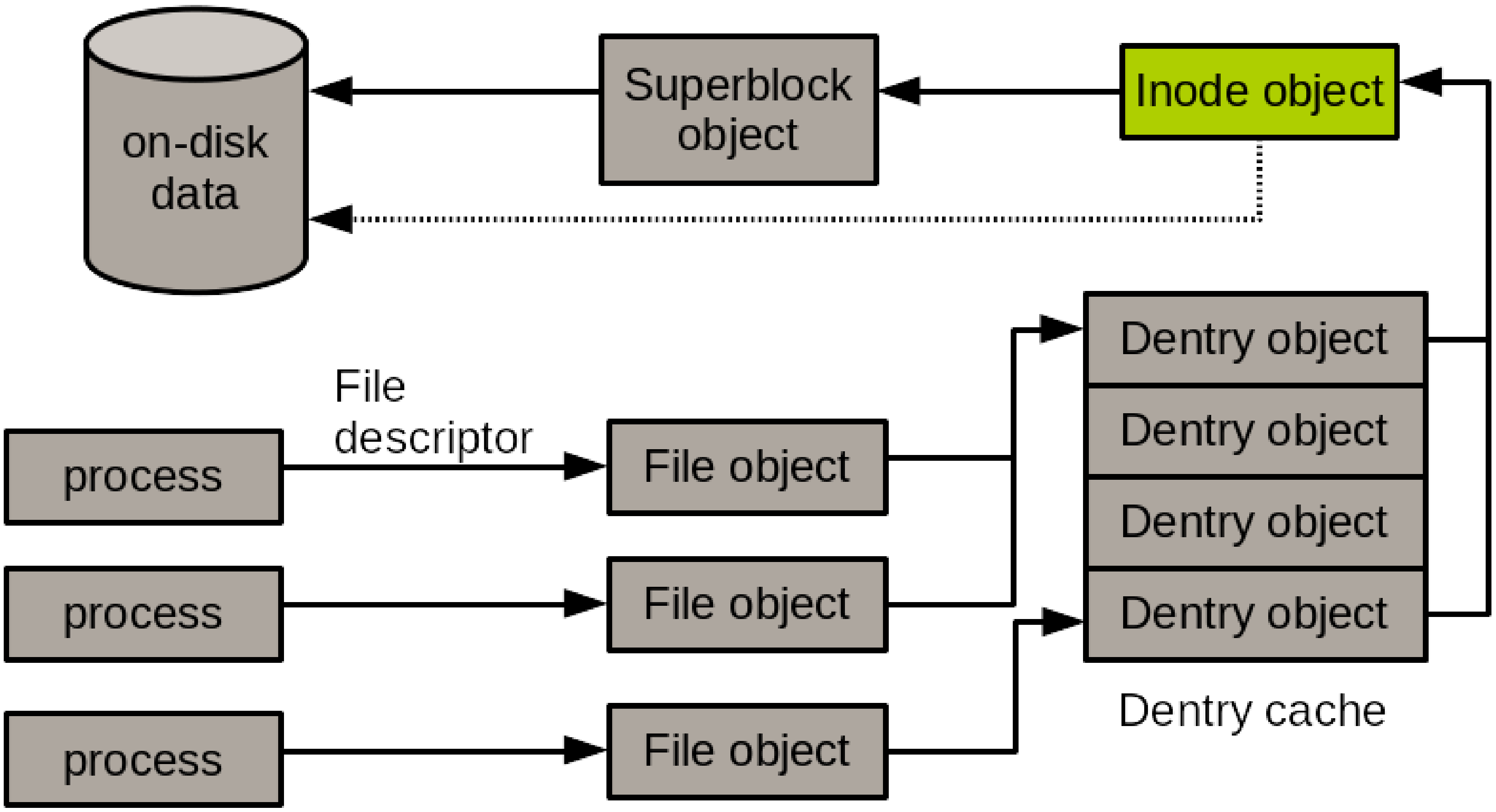
VFS data structures

Superblock operations (4)

- `int statfs(struct super_block *sb, struct statfs *statfs)`
  - Obtain filesystem statistics
- `int remount_fs(struct super_block *sb, int *flags, char *data)`
  - Remount the filesystem with new options, caller must hold `s_lock`
- `void clear_inode(struct inode *inode)`
  - Releases the inode and clear any page containing related data
- `void umount_begin(struct super_block *sb)`
  - Called by VFS to interrupt a mount operation (NFS)

- All of these functions are called by VFS and may block (except `dirty_inode()`

VFS data structures



VFS data structures

Inode object

- **Inode object**
  - Related to a file or directory, contains metadata plus information about how to manipulate the file/directory
  - Metadata: file size, owner id/group, etc
  - Must be produced by the filesystem on-demand when a file/directory is accessed:
    - Read from disk in Unix-like filesystem
    - Reconstructed from on-disk information for other filesystems
  - **`struct inode`** (`include/linux/fs.h`):

```
struct inode {
  struct hlist_node i_hash;     /* hash list */
  struct list_head  i_list;     /* list of inodes */
  struct list_head i_sb_list; /* list of superblock */
  struct list_head i_dentry; /* list of dentries */
  /* ... */
```

VFS data structures

Inode object (2)

```
/* ... */
unsigned long            i_ino;           /* inode number */
atomic_t                 i_count;         /* reference counter */
unsigned int             i_nlink;         /* number of hard links */
uid_t                    i_uid;           /* user id of owner */
gid_t                    i_gid;           /* group id of owner */
kdev_t                   i_rdev;          /* real device node */
u64                      i_version;       /* versioning number */
loff_t                   i_size;          /* file size in bytes */
seqcount_t               i_size_seqcount  /* seqlock for i_size */
struct timespec          i_atime;         /* last access time */
struct timespec          i_mtime;         /* last modify time (file content) */
struct timespec          i_ctime;         /* last change time (file or attributes content) */
unsigned int             i_blkbits;       /* block size in bits */
blkcnt_t                 i_blocks;        /* file size in blocks */
unsigned short           i_bytes          /* bytes consumed */
spinlock_t               i_lock;          /* inode spinlock */
struct rw_semaphore      i_alloc_sem;     /* nests inside of i_sem */
struct semaphore         i_sem;           /* inode semaphore */
struct inode_operations *i_ops;         /* inode operations */
struct file_operations  *i_fop;         /* file operations */
struct super_block   i_sb;    /* associated superblock */
/* ... */
```

VFS data structures

Inode object (3)

```
  /* ... */
  struct dquot          *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
  struct list_head       i_devices;          /* list of block device */
  union {
    struct pipe_inode_info *i_pipe; /* pipe information */
    struct block_device    *i_bdev; /* block device driver */
    struct cdev            *i_cdev; /* character device */
  };
  unsigned long          i_dnotify_mask; /* directory notify mask */
  struct dnotify_struct *i_dnotify;      /* dnotify */
  struct list_head       inotify_watches; /* inotify watches */
  struct mutex           inotify_mutex;  /* protects inotify_watches */
  unsigned long          i_state;        /* state flags */
  unsigned long          dirtied_when;   /* first dirtying time */
  unsigned int           i_flags;        /* filesystem flags */
  atomic_t               i_writecount;   /* count of writers */
  void *                 i_private;      /* filesystem private data */
  /* ... */
}
```

VFS data structures

Inode operations

- Operations that can be invoked on an inode object
- **`struct inode_operations`** defined in `include/linux/fs.h`

```
struct inode_operations {
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  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 *,umode_t);
  /* ... */
}
```

VFS data structures

Inode operations (2)

- `int create(struct inode *dir, struct dentry *dentry, int mode)`
  - Create a new inode with access mode `mode`
  - Called from `creat()` and `open()` syscalls
- `struct dentry * lookup(struct inode *dir, struct dentry *dentry)`
  - Searches a directory (inode) for a file/directory (dentry)
- `int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)`
  - Creates a hard link with name `dentry` in the directory `dir`, pointing to `old_dentry`
- `int unlink(struct inode *dir, struct dentry *dentry)`
  - Remove an inode (`dentry`) from the directory `dir`

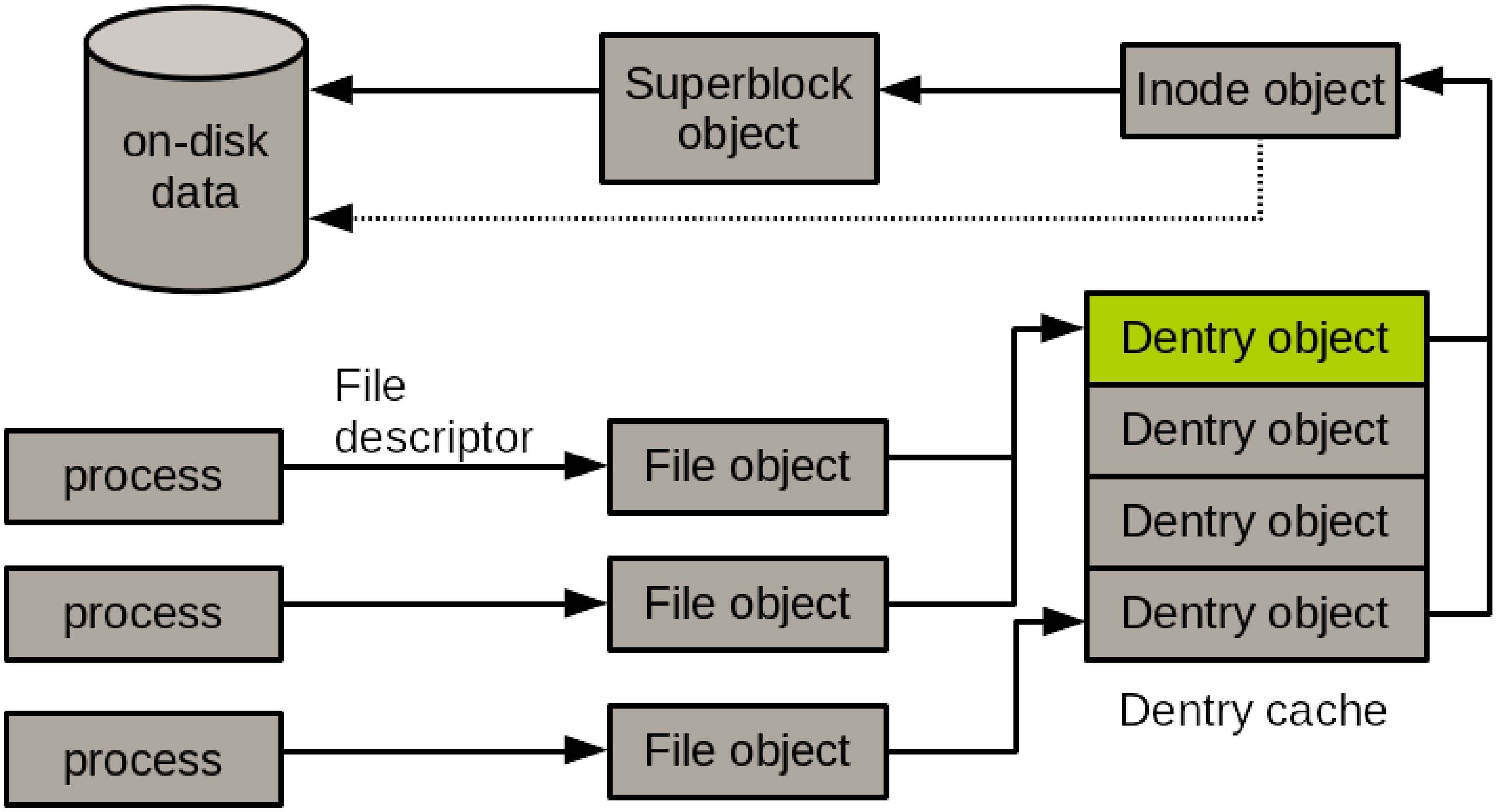
VFS data structures

Inode operations (3)

- `int symlink(struct inode *dir, struct dentry *dentry, const char *symname)`
  - Creates a symbolic link named `symname`, to the file `dentry` in directory `dir`
- `int mkdir(struct inode *dir, struct dentry *dentry, int mode)`
  - Creates a directory inside `dir` with name `dentry`
- `int rmdir(struct inode *dir, struct dentry *dentry)`
  - Removes a directory `dentry` from `dir`
  - `int mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t rdev)`
    - Creates a special file (device file, pipe, socket)
- `int rename(struct struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry)`
  - Moves a file

VFS data structures

Dentry object



VFS data structures

Dentry object

- **Dentry object**
  - Associated with a file or a directory to:
    - Store the file/directory **name**
    - Store its **location in the directory tree**
    - Perform directory specific operations, for example pathname lookup
  - `/home/pierre/test.txt`:
    - One dentry associated with each of: `/`, `home`, `pierre` and `test.txt`
  - Constructed on the fly as files and directories are accessed: generally no on-disk representation

VFS data structures

Dentry object

- `struct dentry` defined in `include/linux/dcache.h`

```
struct dentry {
    atomic_t           d_count;    /* usage count */
    unsigned int       d_flags;    /* dentry flags */
    spinlock_t         d_lock;     /* per-dentry lock */
    int                d_mounted; /* indicate if it is a mount point */
    struct inode       *d_inode; /* associated inode */
    struct hlist_node d_hash;     /* list of hash table entries */
    struct dentry      *d_parent; /* parent dentry */
    struct qstr        d_name;     /* dentry name */
    struct list_head   d_lru;      /* unused list */
    struct list_head          d_subdirs;     /* sub-directories */
    struct list_head          d_alias;       /* list of dentries pointing to the same inode */
    unsigned long             d_time;        /* last time validity was checked */
    struct dentry_operations *d_op;          /* operations */
    struct super_block        *d_sb;         /* superblock */
    void                      *d_fsdata;     /* filesystem private data */
    unsigned char             d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
    /* ... */
};
```

VFS data structures

Dentry states

- A dentry can be **used**, **unused** or **negative**
- **Used**: corresponds to a valid inode (pointed by `d_inode`) with one or more users (`d_count`)
  - Cannot be discarded to free memory
- **Unused**: valid inode, but no current users
  - Kept in RAM for caching
  - Can be discarded
- **Negative**: does not point to a valid inode
  - Ex: `open()` on a file that does not exists
  - Kept around for caching
  - Can be discarded

VFS data structures

The dentry cache

- Dentries are constructed on demand and **kept in RAM for quick future pathname lookups**
  - **Dentry cache** or Dcache
- Three parts:
  - Linked list of used dentries linked by the `i_dentry` field of their inode
    - One inode can have multiple links, thus multiple dentries
  - Linked list of LRU sorted unused and negative dentries
    - LRU: quick reclamation from the tail of the list
  - Hash table + hash function to quickly resolve a path into the corresponding dentry present in the dcache

VFS data structures

The dentry cache (2)

- Hash table: `dentry_hashtable` array
  - Each element is a pointer to a list of dentries hashing to the same value
- Hashing function: `d_hash()`
  - Filesystem can provide its own hashing function
- Dentry lookup in the dcache: `d_lookup()`
  - Returns dentry on success, `NULL` on failure
- Inodes are similarly cached in RAM, in the **inode cache**
  - Dentries in the dcache are pinning inodes in the inode cache

VFS data structures

Dentry operations

- `struct dentry operations` defined in `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 *,
       unsigned int, const char *, const struct qstr *);
   int (*d_delete)(const struct dentry *);
   int (*d_init)(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)(const struct path *, bool);
   struct dentry *(*d_real)(struct dentry *, const struct inode *,
          unsigned int);
 }_____cacheline_aligned;
```

VFS data structures

Dentry operations (2)

- `int d_revalidate(struct dentry *dentry, struct nameidata *)`
  - Determine if an entry to use from the dcache is valid
  - Generally set to `NULL`
- `int d_hash(struct dentry *dentry, struct qstr *name)`
  - Create a hash value for a dentry to insert in the dcache
- `int d_compare(struct dentry *dentry, struct qstr *name1, struct qstr *name2)`
  - Compare two filenames, requires `dcache_lock`
- `int d_delete (struct dentry *dentry)`
  - Called by VFS when `d_count` reaches zero, requires `dcache_lock` and `d_lock`

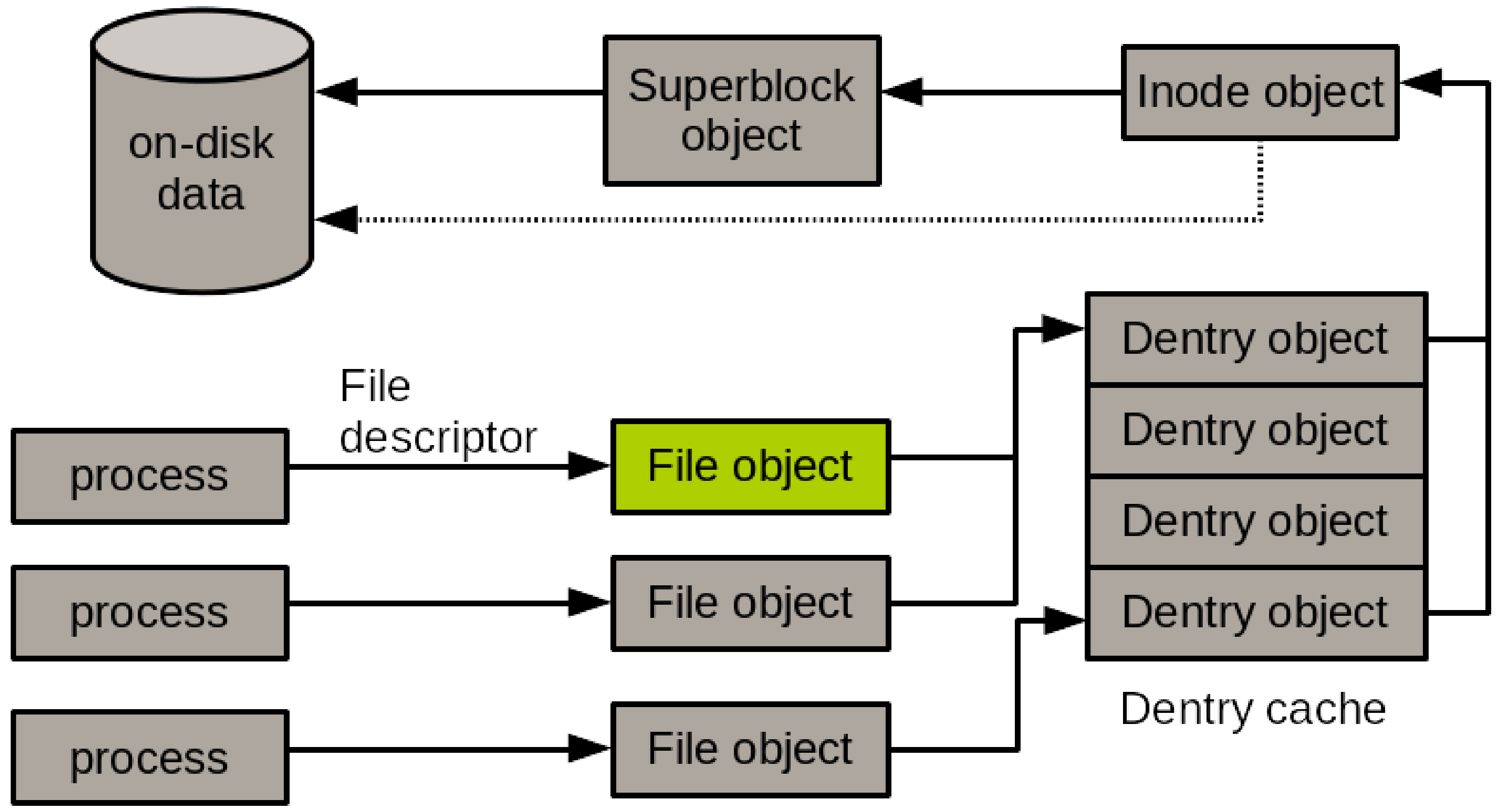
VFS data structures

Dentry operations (3)

- `void d_release(struct dentry *dentry)`
  - Called when the dentry is going to be freed
- `void d_iput(struct dentry *dentry, struct inode *inode)`
  - Called when the dentry looses its inode
  - Calls `iput()`

VFS data structures

File object



VFS data structures

File object

- The **file** object
  - Represents a file opened by a process
  - Created on `open()` and destroyed on `close()`
- 2 processes opening the same file:
  - Two file objects, pointing to the same unique dentry, that points itself on a unique inode
- No corresponding on-disk data structure

VFS data structures

File object (2)

- `struct file` defined in `include/linux/fs.h`

```
struct file {
 struct path   f_path;     /* contains the dentry */

 spinlock_t               f_lock;              /* lock */
 atomic_t                 f_count;             /* usage count */
 unsigned int             f_flags;             /* open flags */
 mode_t                   f_mode;              /* file access mode */
 logg_t                   f_pos;               /* file offset */
 struct fown_struct       f_owner;             /* owner data for signals */
 const struct cred         *f_cred;            /* file credentials */
 struct file_ra_state     f_ra;                /* read-ahead state */
 u64                      f_version;           /* version number */
 void                     *private_data;       /* private data */
 struct list_head         f_ep_link;           /* list of epoll links */
 spinlock_t               f_ep_lock;           /* epoll lock */
 struct address_space     *f_mapping;          /* page cache mapping */
 /* ... */
```

VFS data structures

File operations

- **`struct file operations`** 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_user *, size_t, loff_t *);  ssize_t
  (*write) (struct file *, const char_user *, size_t, loff_t *);  ssize_t
  (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  /* ... */

};
```

VFS data structures

File operations

- `loff_t llseek(struct file *file, loff_t offset, int origin)`
    - Update file offset
- `ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)`
    - Read operation
- `ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)`
    - Asynchronous read
- `ssize_t write(struct file *file, const char *buf, size_t count, loff_t *offset)`
    - Write operation
- `ssize_t aio_write(struct kiocb *iocb, const char *buf, size_t count, lofft offset)`
    - Asynchronous write

VFS data structures

File operations (2)

- `int readdir(struct file *file, void *dirent, filldir_t filldir)`
  - Read the next directory in a directory listing
- `unsigned int poll(struct file *file, struct poll_table_struct *poll_table)`
  - Sleeps waiting for activity on a given file
- `int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)`
  - Sends a command and arguments to a device
  - Unlocked/compat versions
- `int mmap(struct file *file, struct vm_area_struct *vma)`
  - Maps a file into an address space

VFS data structures

File operations (3)

- `int open(struct inode *inode, struct file *file)`
  - Opens a file
- `int flush(struct file *file)`
  - Called by VFS when the reference count of an open file decreases
- `int release(struct inode *inode, struct file *file)`
  - Called by VFS when the last reference to a file is destroyed (`close()`/`exit()`)
- `int fsync(struct file *file, struct dentry *dentry, int datasync)`
  - Flush cached data on disk
- `int aio_fsync(struct kiocb *iocb, int datasync)`
  - Flush aio cached data on disk

VFS data structures

File operations (4)

- `int lock(struct file *file, int cmd, struct file_lock *lock)`
  - Manipulate a file lock
- `ssize_t writev(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)`
- `ssize_t readv(struct file *file, const struct iovec *vector, unsigned long count)`
  - Vector read/write operations (used by the `readv` and `writev` family functions)
- `ssize_t sendfile(struct file *file, loff_t *offset, size_t size, read_actor_t actor, void *target)`
  - Copy data from one file to another entirely in the kernel

VFS data structures

File operations (5)

- `ssize_t sendpage(struct file *file, struct page *page, int offset, size_t size, loff_t *pos, int more)`
  - Send data from one file to another
- `unsigned long get_unmapped_area(struct file *file, unsigned long addr, unsigned long len, unsigned long offset, unsigned long flags)`
  - Get a section of unused address space to map a file
- `int flock(struct file *filp, int cmd, struct file_lock *fl)`
  - Used by the `flock()` syscall

Outline

Filesystem and process data structures

Filesystem data structures

- **`struct file_system_type`**: information about a specific concrete filesystem type
- One per filesystem supported (chosen at compile time) independently of the mounted filesystem
- Defined in `include/linux/fs.h`:

Filesystem and process data structures

Filesystem data structures (2)

```c
struct file_system_type  {
  const char *name;       /* name */
  int fs_flags;         /* flags */

  /* mount a partition */
  struct dentry *(*mount) (struct file_system_type *, int,
           const char *, void *);

  /* terminate access to the superblock */
  void (*kill_sb) (struct super_block *);
  struct module *owner;                     /* module owning the fs */
  struct file_system_type * next;           /* linked list of fs types */
  struct hlist_head fs_supers;              /* linked list of superblocks */

  /* runtime lock validation */
  struct lock_class_key s_lock_key;
  struct lock_class_key s_umount_key;
  struct lock_class_key s_vfs_rename_key;
  struct lock_class_key s_writers_key[SB_FREEZE_LEVELS];

  struct lock_class_key i_lock_key;
  struct lock_class_key i_mutex_key;
  struct lock_class_key i_mutex_dir_key;
};
```

Filesystem and process data structures

Filesystem data structures (3)

- When a filesystem is mounted, a `vfsmount` structure is created
  - Represent a specific instance of the filesystem: a mount point
- `include/linux/mount.h`

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

Filesystem and process data structures

Process data structure (4)

- `struct files_struct`: contains per-process information about opened files and file descriptors
  - `include/linux/fdtable.h`
- `struct fs_struct`: filesystem information related to a process
  - `include/linux/fs_struct.h`
- `struct mnt_namespace`: provide processes with unique views of a mounted filesystem
  - `fs/mount.h`

Outline

Additional information

- `Documentation/filesystems`
- *Understanding the Linux Kernel*, chapter 12
- *Linux Kernel Architecture*, chapter 8