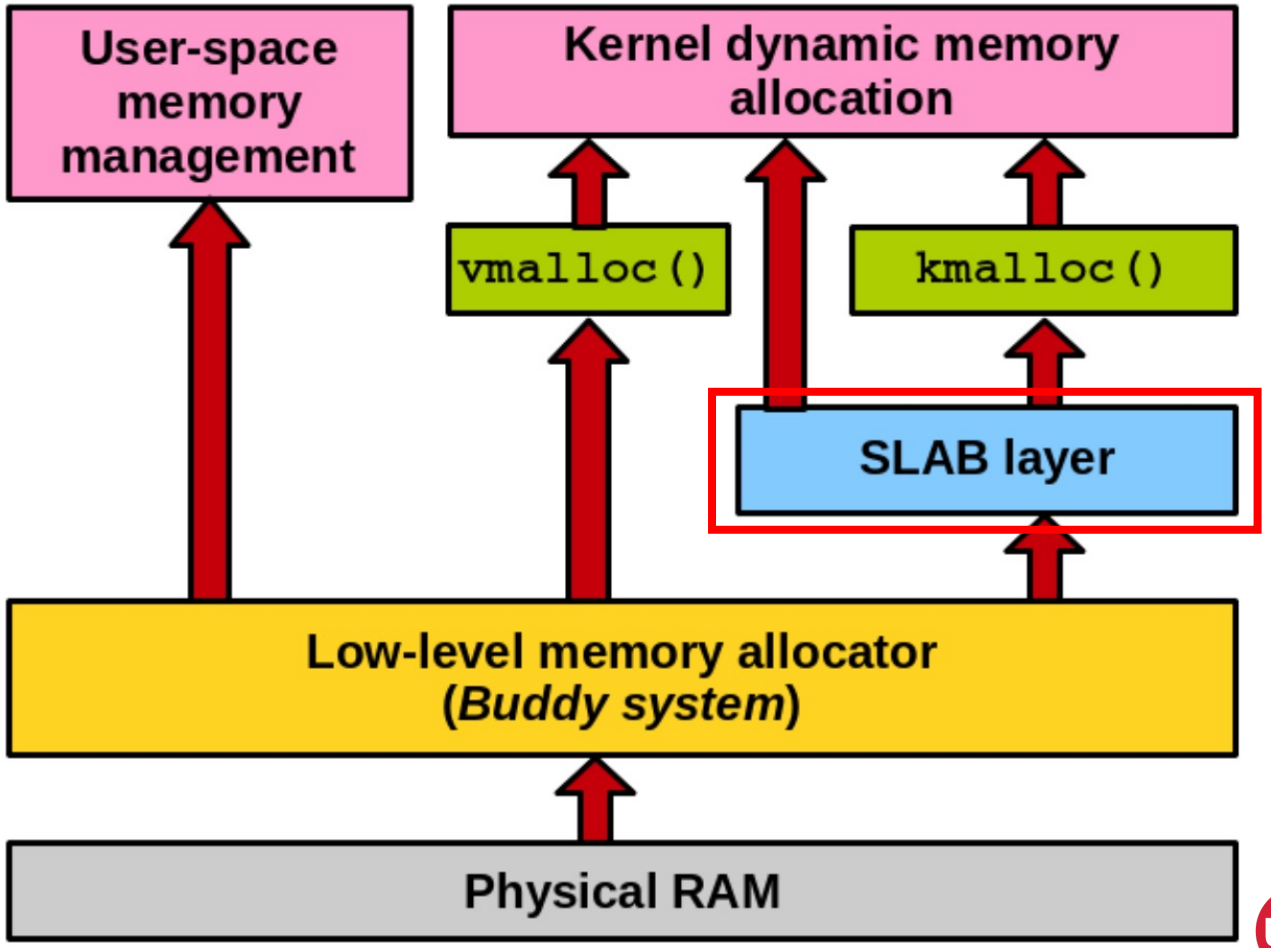
Recap: Slab allocator APIs

```c
/**
 * Create a cache for a data structure type
 */
struct kmem_cache *kmem_cache_create(
    const char *name,     /* Name of the cache */
    size_t size,          /* Size of objects */
    size_t align,         /* Offset of the first element
                             within pages */
    unsigned long flags, /* Options */
    void (*ctor)(void *) /* Constructor */
);

/**
 * Destroy the cache
 * - Should be only called when all slabs in the cache are empty
 * - Should not access the cache during the destruction
 */
void kmem_cache_destroy(struct kmem_cache *cachep);
```

Recap: Slab allocator APIs

```
/**
 * Allocate an object from the cache
 */
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);

/**
 * Free an object allocated from a cache
 */
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

Recap: Hierarchy of memory allocators



Does `kmalloc()` use slab?

include/linux/slab.h

- kmalloc(size_t size, gfp_t flags) → __kmalloc(size, flags);

mm/slab_common.c

- __kmalloc() → __do_kmalloc_node(size, flags, NUMA_NO_NODE, _RET_IP_);

→

```
...
s = kmalloc_slab(size, flags);
...
```

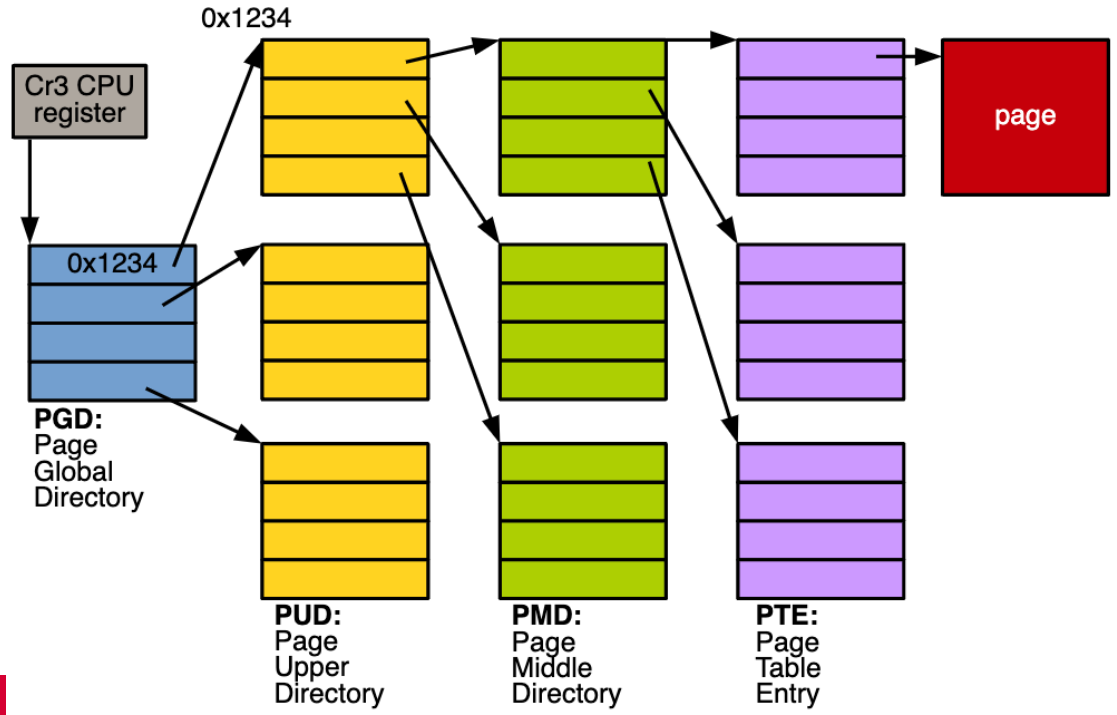
https://ruffell.nz/programming/writeups/2019/02/15/looking-at-kmalloc-and-the-slub-memory-allocator.html

Does `kmalloc()` use slab?

```
└─$ sudo cat /proc/slabinfo | grep kmalloc
kmalloc-cg-8k         32     32   8192    4    8 : tunables    0    0    0 : slabdata      8      8      0
kmalloc-cg-4k        160    160   4096    8    8 : tunables    0    0    0 : slabdata     20     20      0
kmalloc-cg-2k        539    608   2048   16    8 : tunables    0    0    0 : slabdata     38     38      0
 ... ...
kmalloc-1k          2421   2464   1024   32    8 : tunables    0    0    0 : slabdata     77     77      0
kmalloc-512        26411  26624    512   32    4 : tunables    0    0    0 : slabdata    832    832      0
kmalloc-256         7986   8032    256   32    2 : tunables    0    0    0 : slabdata    251    251      0
kmalloc-192        10710  10710    192   21    1 : tunables    0    0    0 : slabdata    510    510      0
kmalloc-128         1792   1792    128   32    1 : tunables    0    0    0 : slabdata     56     56      0
kmalloc-96          3648   5166     96   42    1 : tunables    0    0    0 : slabdata    123    123      0
kmalloc-64         14809  15168     64   64    1 : tunables    0    0    0 : slabdata    237    237      0
kmalloc-32         12658  12672     32  128    1 : tunables    0    0    0 : slabdata     99     99      0
kmalloc-16         15734  15872     16  256    1 : tunables    0    0    0 : slabdata     62     62      0
kmalloc-8          10227  10240      8  512    1 : tunables    0    0    0 : slabdata     20     20      0
```

Recap: Page tables

```
/* include/linux/mm_types.h */
struct mm_struct {
    struct vm_area_struct *mmap;          /* list of VMAs */
    struct rb_root        mm_rb;          /* rbtree of VMAs */
    pgd_t                 *pgd;           /* page global directory */
    /* ... */
};
```



Recap: address space and VM areas

Address space in Linux kernel: `struct mm_struct`

```
└─$ sudo cat /proc/1/maps     # or sudo pmap 1
55ff919f7000-55ff91a2d000 r--p 00000000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91a2d000-55ff91b0d000 r-xp 00036000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b0d000-55ff91b6c000 r--p 00116000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b6c000-55ff91bbb000 r--p 00174000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbb000-55ff91bbc000 rw-p 001c3000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbc000-55ff91bbd000 rw-p 00000000 00:00 0
55ff93061000-55ff93383000 rw-p 00000000 00:00 0                          [heap]
```

Virtual memory area (VMA)

- Each line corresponds to one VMA

Recap: address space and VM areas

Address space in Linux kernel: `struct mm_struct`

```
└─$ sudo cat /proc/1/maps
55ff919f7000-55ff91a2d000
55ff91a2d000-55ff91b0d000
55ff91b0d000-55ff91b6c000
55ff91b6c000-55ff91bbb000
55ff91bbb000-55ff
55ff91bbc000-55ff
55ff93061000-55ff
```

```
/* include/linux/mm_types.h */
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    struct rb_root        mm_rb;        /* rbtree of VMAs */
    pgd_t                 *pgd;         /* page global directory */
```

```
/* include/linux/mm_types.h */
struct vm_area_struct {
    struct                mm_struct *vm_mm;  /* associated address space */
    unsigned long         vm_start;          /* VMA start, inclusive */
    unsigned long         vm_end;            /* VMA end, exclusive */
    struct vm_area_struct *vm_next;          /* list of VMAs */
    struct vm_area_struct *vm_prev;          /* list of VMAs */
    pgprot_t              vm_page_prot;      /* access permissions */
    unsigned long         vm_flags;          /* flags */
    struct rb_node        vm_rb;             /* VMA node in the tree */
    struct list_head      anon_vma_chain;    /* list of anonymous mappings */
    struct anon_vma       *anon_vma;         /* anonmous vma object */
```

Virtual memor

- Each line c

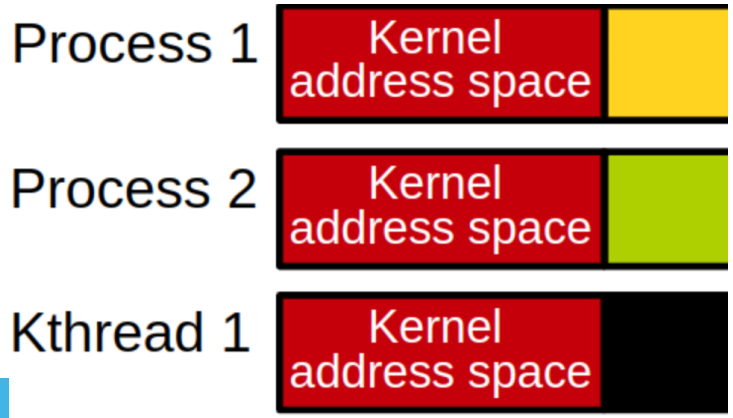
Recap: The `mm_struct`

Two user threads sharing the same address space have the `mm` field of their `task_struct` pointing to the same `mm_struct` object

- Linux threads: `clone()` with the `CLONE_VM` flag set

Kernel threads do not have a

- `mm` field of a kernel thread



`CLONE_VM` (since Linux 2.0)
If **`CLONE_VM`** is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with `mmap(2)` or `munmap(2)` by the child or calling process also affects the other process.

If **`CLONE_VM`** is not set, the child process runs in a separate copy of the memory space of the calling process at the time of the clone call. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with `fork(2)`.

VMA operations

vm_ops in vm_area_struct is a struct of function pointers to operate on a specific VMA

```
/* include/linux/mm.h */
struct vm_operations_struct {
    /* called when the area is added to an address space */
    void (*open)(struct vm_area_struct * area);

    /* called when the area is removed from an address space */
    void (*close)(struct vm_area_struct * area);

    /* invoked by the page fault handler when a page that is
     * not present in physical memory is accessed*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* invoked by the page fault handler when a previously read-only
     * page is made writable */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* ... */
}
```

Create an address interval

`do_mmap()` is used to create a new linear address interval

- Can result in the creation of a new VMAs
- Or a merge of the create area with an adjacent one when they have the same permissions

```
/*
 * The caller must hold down_write(&current->mm->mmap_sem).
 */
unsigned long do_mmap(struct file *file, unsigned long addr,
            unsigned long len, unsigned long prot,
            unsigned long flags, vm_flags_t vm_flags,
            unsigned long pgoff, unsigned long *populate,
            struct list_head *uf);
```

Create an address interval

`prot` specifies access permissions for the memory pages

Flag	Effect on the new interval
`PROT_READ`	Corresponds to `VM_READ`
`PROT_WRITE`	Corresponds to `VM_WRITE`
`PROT_EXEC`	Corresponds to `VM_EXEC`
`PROT_NONE`	Cannot access page

Create an address interval

`flags` specifies the rest of the VMA options

Flag	Effect on the new interval
`MAP_SHARED`	The mapping can be shared.
`MAP_PRIVATE`	The mapping cannot be shared.
`MAP_FIXED`	The new interval must start at the given address addr.
`MAP_ANONYMOUS`	The mapping is not file-backed, but is anonymous.
`MAP_GROWSDOWN`	Corresponds to `VM_GROWSDOWN`.

Create an address interval

On error `do_mmap()` returns a negative value

On success

- The kernel tries to merge the new interval with an adjacent one having same permissions
- Otherwise, create a new VMA
- Returns a pointer to the start of the mapped memory area

`do_mmap()` is exported to user-space through `mmap2()`

```
void *mmap2(void *addr, size_t length, int prot,
            int flags, int fd, off_t pgoffset);
```

Remove an address interval

Removing an address interval is done through `do_munmap()`

```
/* linux/include/linux/mm.h */
int do_munmap(struct mm_struct *, unsigned long, size_t);
```

Exported to user-space through `munmap()`

```
int munmap(void *addr, size_t len);
```

Further readings

LKD3: Chapter 15 The Process Address Space

Supporting bigger and heterogeneous memory efficiently

- AutoNUMA, Transparent Hugepage Support, Five-level page tables
- Heterogeneous memory management

Optimization for virtualization

- Kernel same-page merging (KSM)
- MMU notifier

Virtual File System

Xiaoguang Wang

The Virtual File System (VFS)

Abstract all the filesystem models supported by Linux

- Similar to an abstract base class in C++

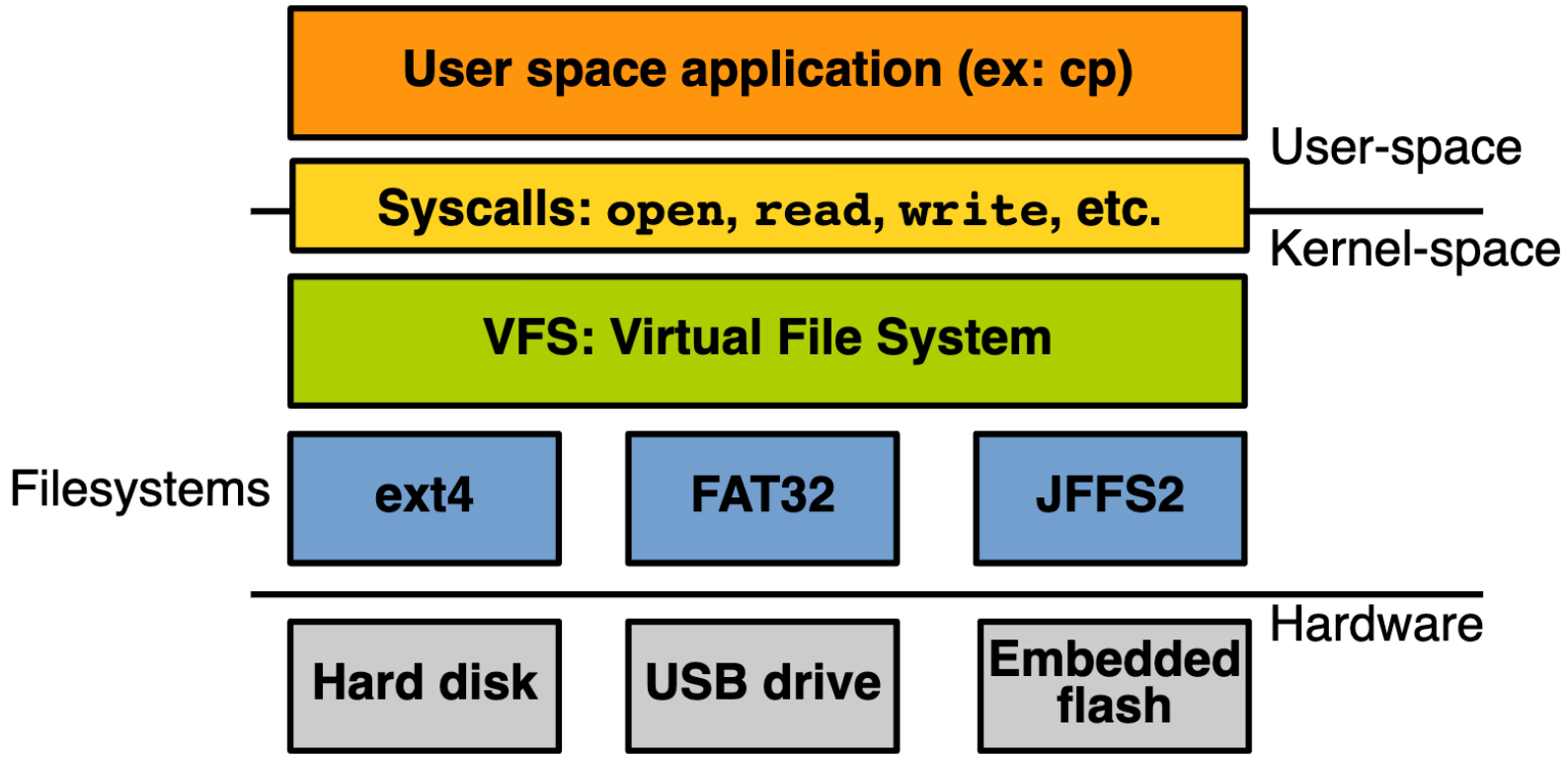
Allow them to **coexist**

- Example: a user can have a *USB drive formatted with FAT32* mounted at the same time as an *HDD rootfs with ext4*

Allow them to **cooperate**

- Example: a user can seamlessly *copy a file between the FAT32 and Ext4 partitions*

The Virtual File System (VFS)



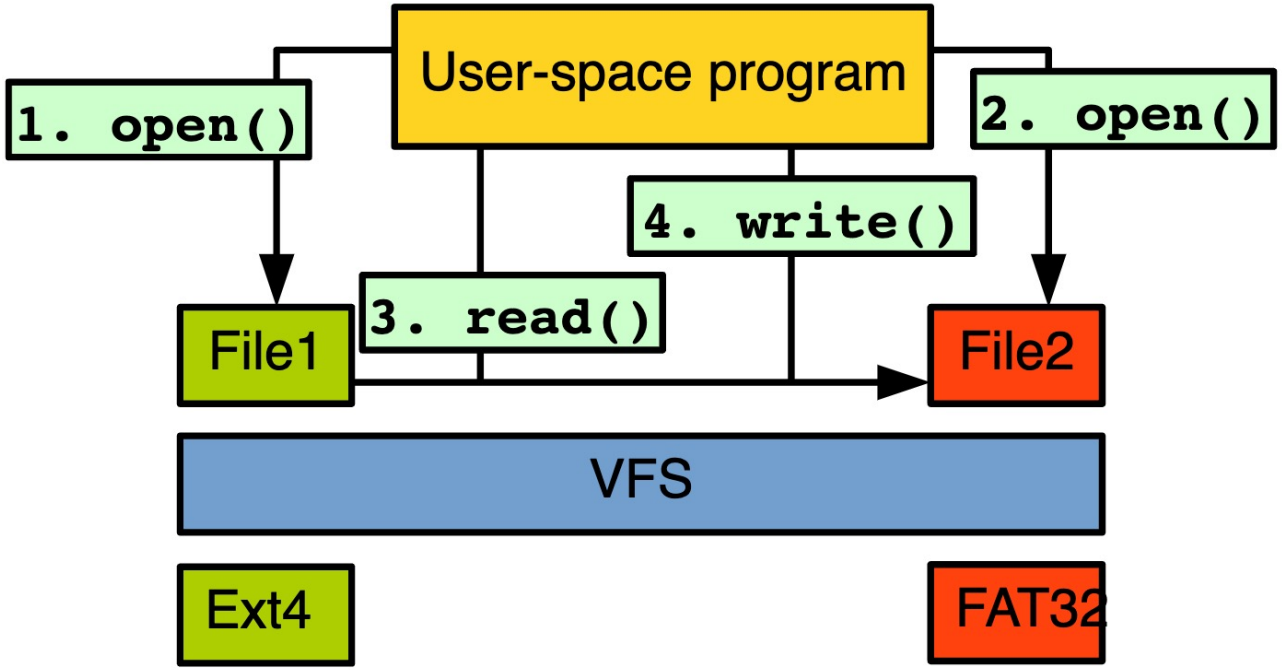
Common filesystem interface

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

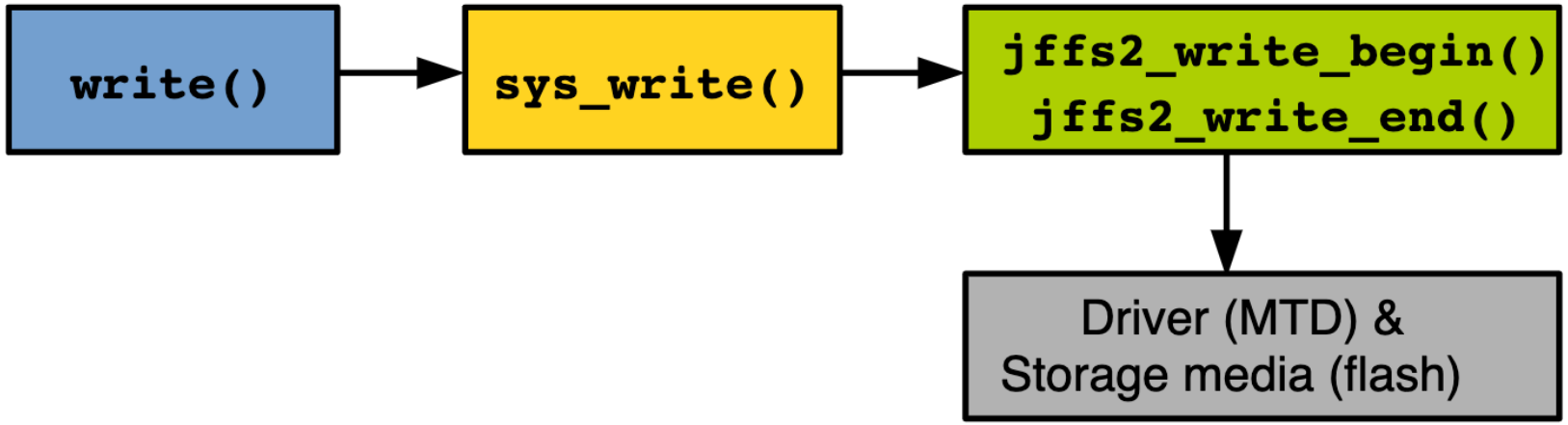
Common filesystem interface



Filesystem abstraction layer

VFS redirects user-space requests to the corresponding concrete filesystem

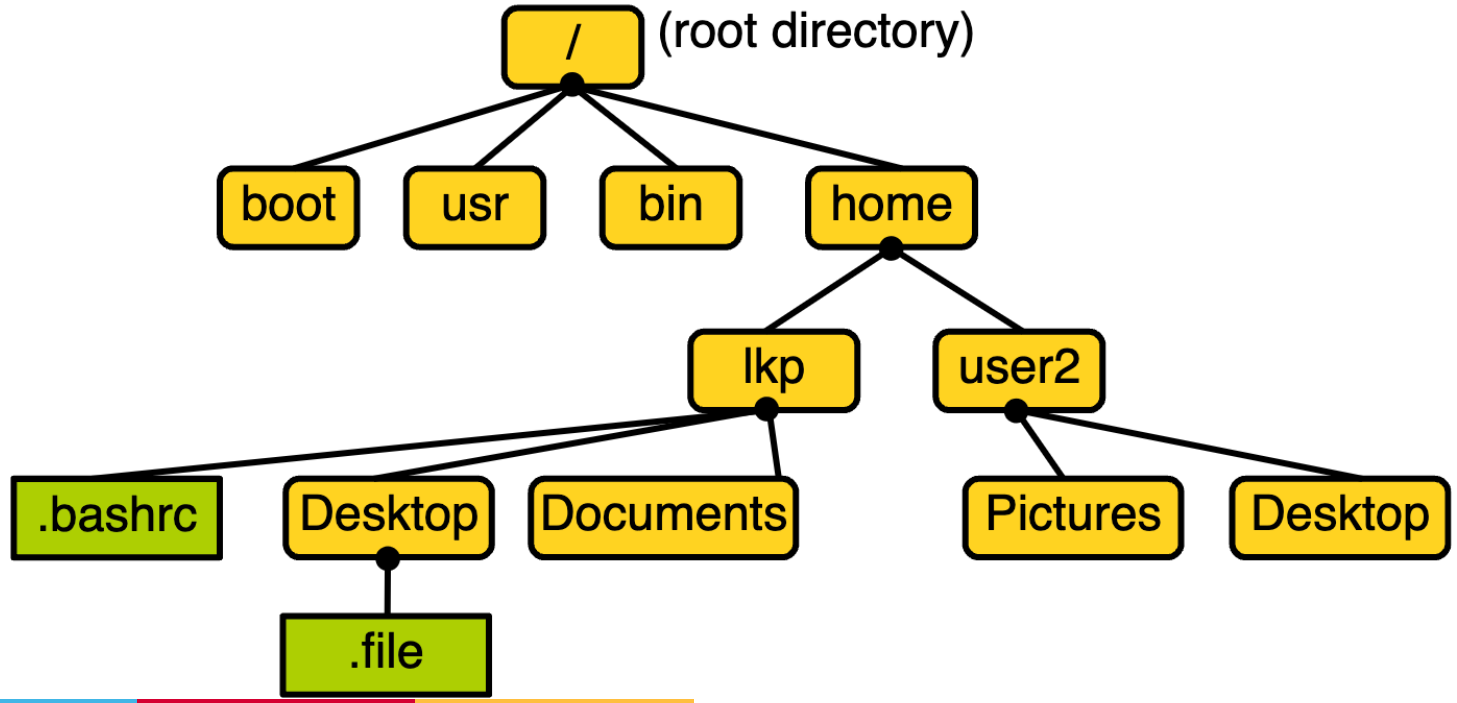
- "bottom" VFS interface (with the filesystem)
- Developing a new filesystem for Linux means conforming with the bottom interface



Unix filesystems

The term filesystem can refer to a filesystem type or a partition

Hierarchical tree of files organized into directories



Unix filesystems

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

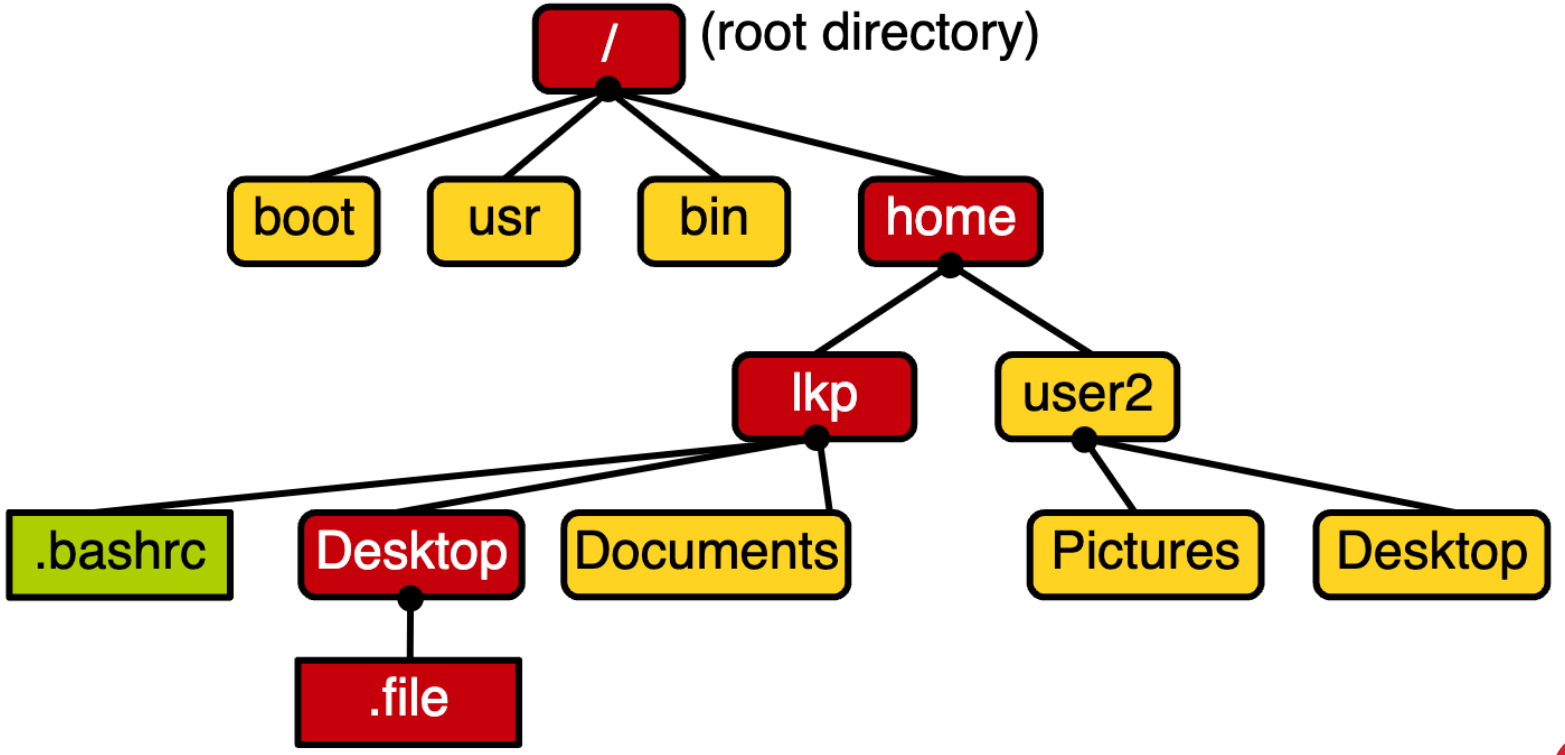
- Metadata: name, access permissions, modification date, etc.
- Separated from the file data into specific objects *inodes*, *dentries*

Directory: folder containing files or other directories (sub-directories)

- Sub-directories can be nested to create path: /home/lkp/Desktop/file

Unix filesystems

Path example: `/home/lkp/Desktop/.file`



VFS data structures

superblock object: represents a specific mounted filesystem

inode object: represents a specific file

dentry: contains file/directory name and hierarchical links defining the filesystem directory tree

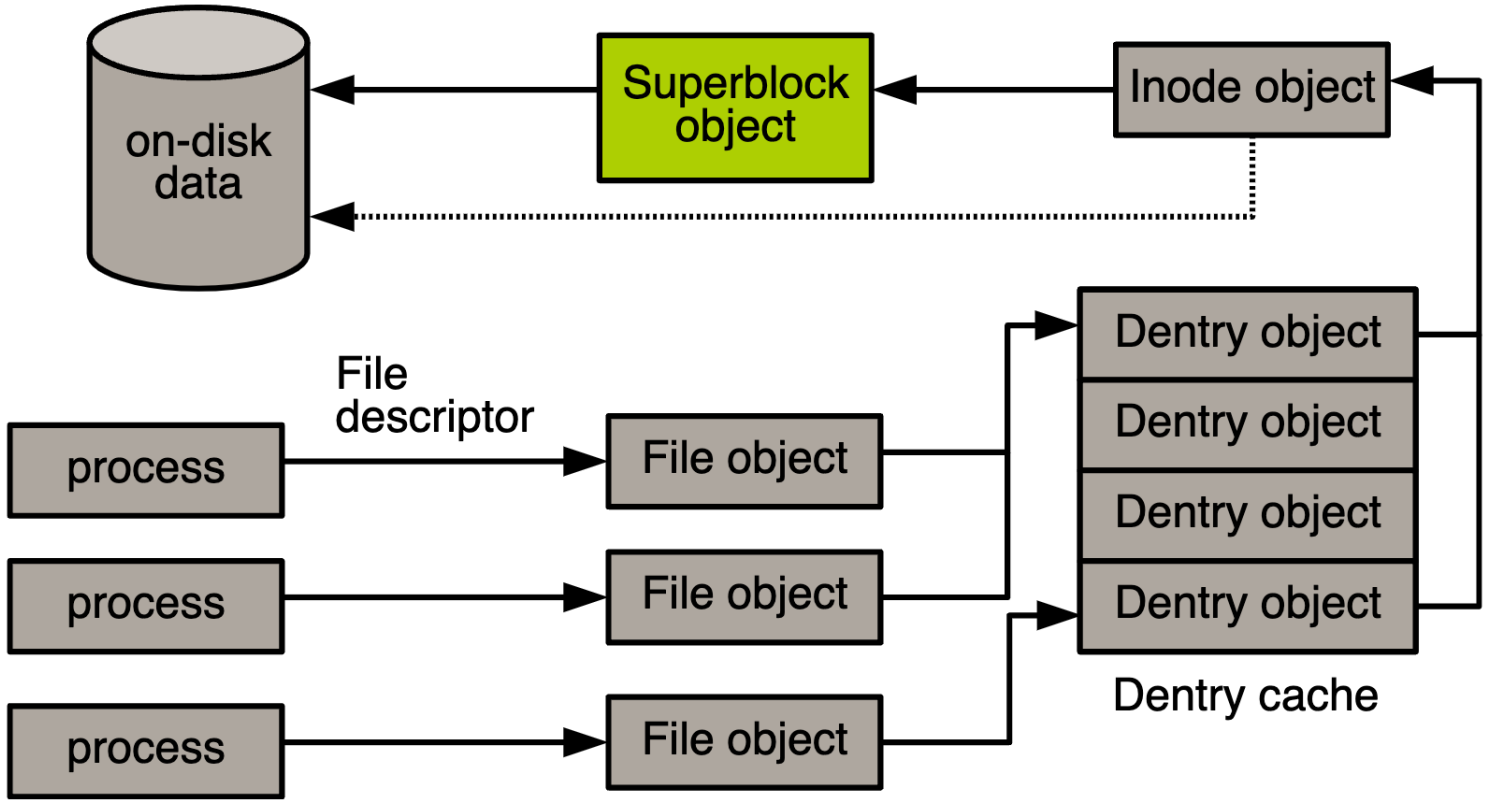
file: represents an opened file associated with a process

file_system_type: contains information about a file system type (ext4)

Associated **operations** ("bottom" VFS interface):

- super_operations, inode_operations, dentry_operations, file_operations

Superblock


on-disk data
Superblock object
Inode object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache
File descriptor
process
process
process
File object
File object
File object

Superblock

Contains global information about the filesystem (partition)

- **Filesystem type**: e.g., ext2, ext3, ext4, XFS, etc.
- **Filesystem size**: the total size of the file system.
- **Block size**: the size of the blocks used by the file system.
- **Inode size**: the size of each inode in the file system.
- **Number of inodes**: the total number of inodes in the file system.
- **Filesystem state**: whether the file system is clean or has errors.
- … …

Superblock

Created by the filesystem and given to VFS at mount time:

- Disk-based filesystem store it in a fixed location; generate when the filesystem is formatted

`struct super_block` defined in `include/linux/fs.h`

```
/* include/linux/fs.h */
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 */
    /* ... */
```

Superblock operations

`struct super_operations`

- Each field is a function pointer operating on a `struct super_block`
- Usage: `sb->s_op->alloc_inode(sb)`

```c
/* 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 *);
    /* ... */
}
```

Superblock operations: `inode`

```
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)

Superblock operations: `inode`

```
void write_inode(struct inode *inode, int wait)
```

- Writes the inode to disk, wait specifies if the write should be synchronous

```
void clear_inode(struct inode *inode)
```

- Releases the inode and clear any page containing related data

```
void drop_inode(struct inode *inode)
```

- Called by VFS when the last reference to the inode is dropped

Superblock operations: `superblock`

```
void put_super(struct super_block *sb)
```

- Called by VFS on unmount (holding `s_lock`)

```
void write_super(struct super_block *sb)
```

- Update the on-disk superblock, caller must hold `s_lock`

Superblock operations: `filesystem`

```
int sync_fs(struct super_block *sb, int wait)
```

- *Synchronize filesystem metadata* with on-disk filesystem, *wait* specifies if the operation should be synchronous

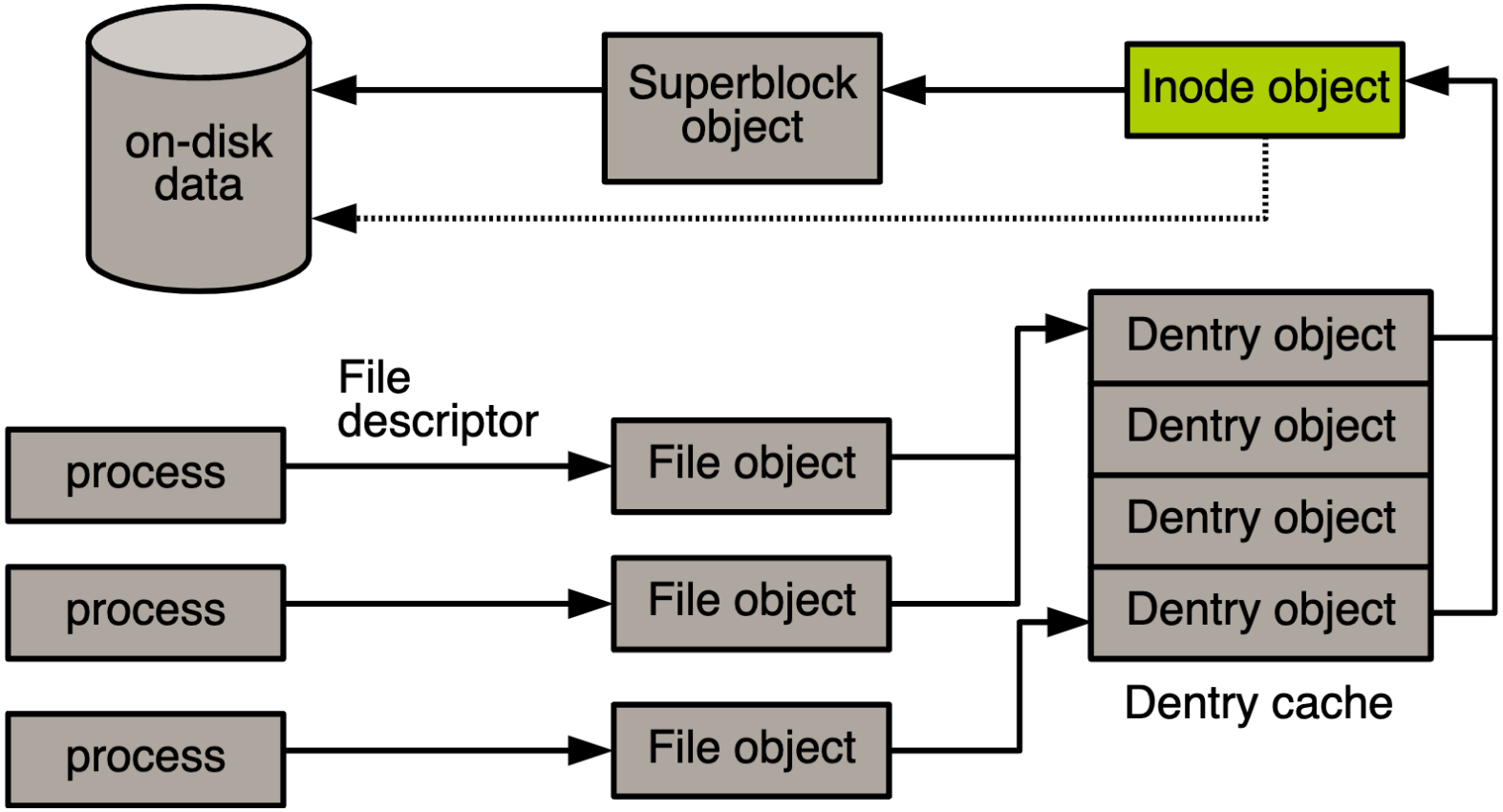
```
void unlockfs(struct super_block *sb)
```

- Unlocks the filesystem locked by `write_super_lockfs()`

```
int statfs(struct super_block *sb, struct statfs *statfs)
```

- Obtain filesystem statistics

inode


on-disk data
Superblock object
Inode object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache
File descriptor
process
process
process
File object
File object
File object

inode

`inode` (index node) is a data structure to represent a file or a directory on the file system

- Every file or directory is associated with an `inode`
- Contains metadata about the file or directory
  - File type, size, permissions, owner/group, timestamp, etc.
- Information about how to manipulate the file/directory

When a file is created, the file system allocates an `inode` to represent that file.

inode

```
/* include/linux/fs.h */
struct inode {
    struct hlist_node       i_hash;         /** hash list **/
    struct list_head        i_lru;          /* inode LRU list*/
    struct list_head        i_sb_list;      /** inode list in superblock **/
    struct list_head        i_dentry;       /** list of dentries **/
    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 */
```

inode

```
    /* ... */
    const struct inode_operations *i_op;    /** inode operations **/
    struct super_block    *i_sb;            /** associated superblock **/
    struct address_space  *i_mapping;       /** associated page cache **/
    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 */
    /* ... */
};
```

inode operations

```
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);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
            struct inode *, struct dentry *, unsigned int);
    /* ... */
};
```

inode operations

```
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)

inode operations

```
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

inode operations

```
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

```
int rmdir(struct inode *dir,struct dentry *dentry)
```

- Removes a directory `dentry` from `dir`

inode operations

```
int mknod(struct inode *dir, struct dentry *dentry, int mode,
dev_t rev)
```

- Creates a special file (device file, pipe, socket)

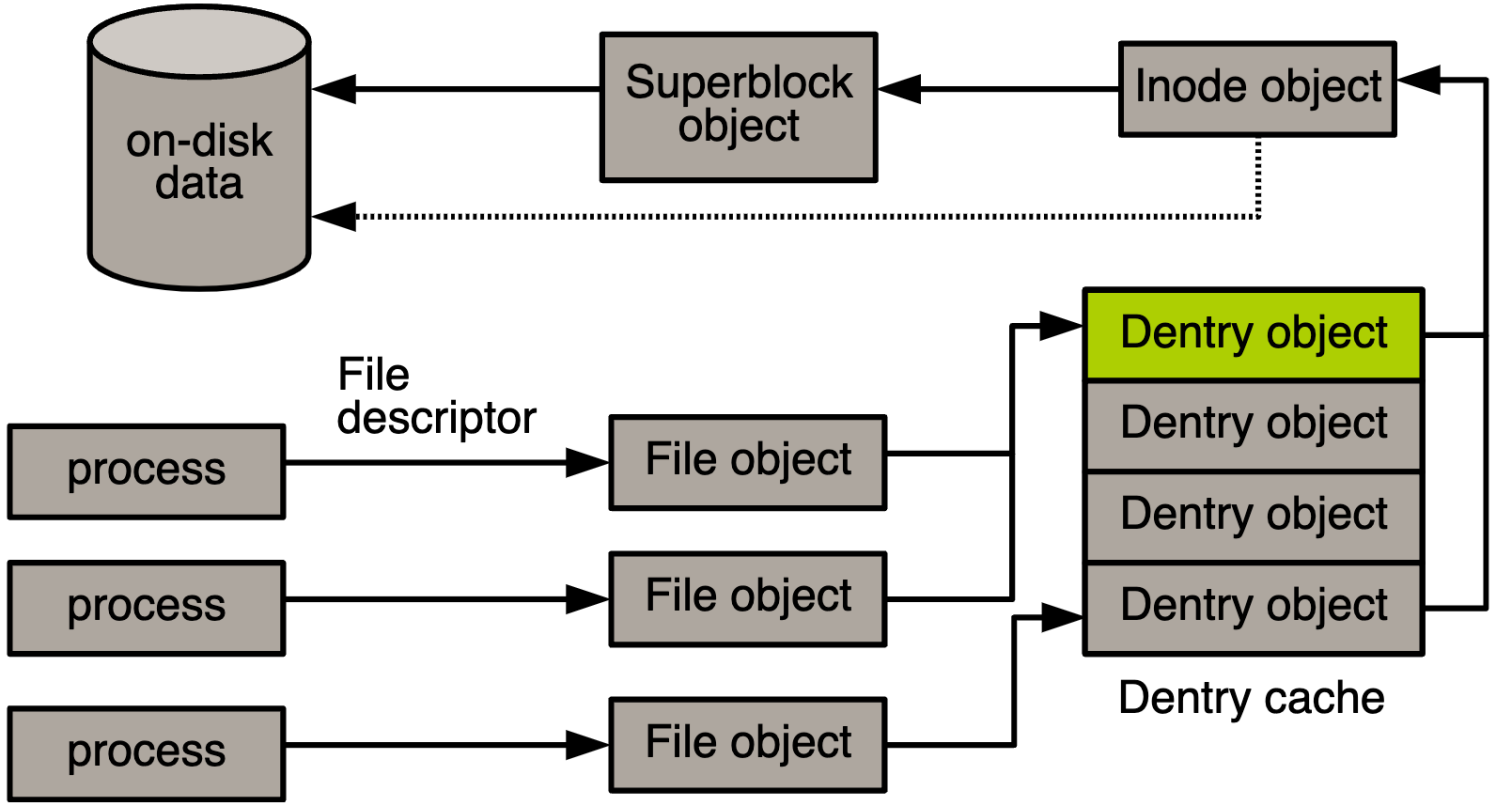
```
int rename(struct struct inode *old_dir, struct dentry
*old_dentry, struct inode *new_dir, struct dentry *new_dentry)
```

- Rename a file

Quiz: which is (are) wrong

1. VFS is an abstraction layer that provides a unified interface for different actual file systems.
2. Both inode and superblock objects represent files and filesystem information on the disk.
3. The superblock contains metadata about a mounted file system, and these operations define how the kernel interacts with and manages the file system's superblock.
4. Each file system defines its own set of inode operations to handle various file-related activities.
5. The user-space process can directly operate on file's inode.
6. The inode object represents the file's actual data.
7. "`ls -i`" will print the inode number of the file on Linux

dentry (or directory entry)



dentry

Associated with a file or a directory to:

- Store the file/directory name
- Store its location in the directory
- Perform directory specific operations, e.g., pathname lookup

`/home/lkp/test.txt`

- One dentry associated with each of: ‘/’, ‘home’, ‘lkp’, and ‘test.txt’

Constructed on the fly as files and directories are accessed

- Cache of disk representation

dentry

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

- e.g., `open()` on a file that does not exists
- Kept around for caching; can be discarded

dentry

```
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 */
    /* ... */
};
```

ER SCIENCE

dentry cache

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

dentry operations

```
/* 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;
```

dentry operations

```
int d_hash(struct dentry *dentry, struct qstr *name)
```

- Create a hash value for a dentry to insert in the dcache

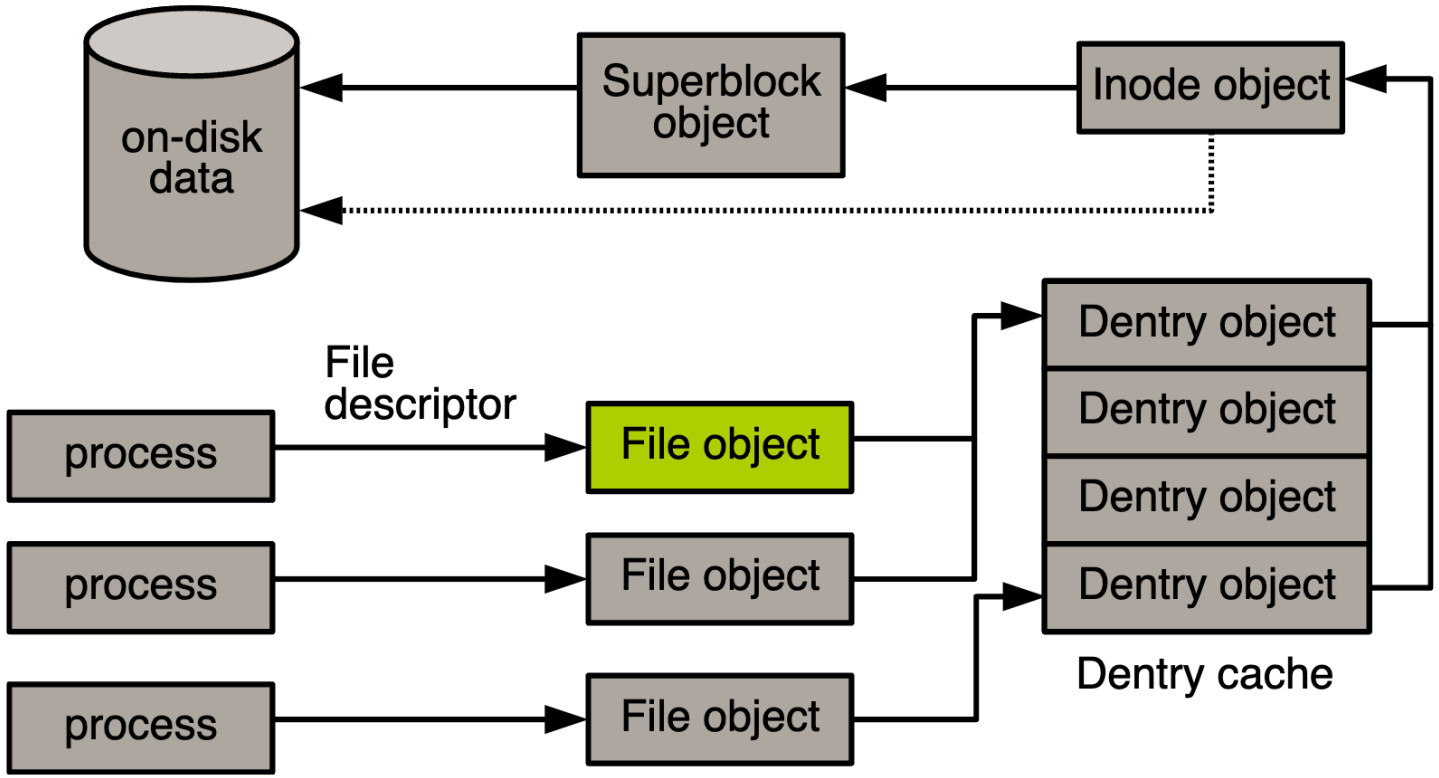
```
int d_compare(struct dentry *dentry, struct qstr *name1,
struct qstr *name2)
```

- 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

File object


on-disk data
Superblock object
Inode object
File descriptor
process
process
process
File object
File object
File object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache

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

File object

```
/* include/linux/fs.h */
struct file {
    struct path             f_path;         /* contains the dentry */
    struct file_operations *f_op;           /** operations **/
    spinlock_t              f_lock;         /* lock */
    atomic_t                f_count;        /* usage count */
    unsigned int            f_flags;        /* open flags */
    mode_t                  f_mode;         /* file access mode */
    loff_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
                                             ** == inode->i_mapping **/

    /* ... */
};
```

File operations

```c
/* include/linux/fs.h */
struct file_operations {
    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 *);
    /* ... */
};
```

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

File operations

```
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, loff_t offset)
```

- Asynchronous write

```
int readdir(struct file *file, void *dirent, filldir_t filldir)
```

- Read the next directory in a directory listing

File operations

```
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

```
int open(struct inode *inode, struct file *file)
```

- Opens a file

Summary

Key data structures

- `struct file_system_type`: file system (e.g., ext4)
- `struct super_block`: mounted file system instance (i.e., partition)
- `struct dentry`: path name
- `struct inode`: file metadata
- `struct file`: open file descriptor

Further readings

LKD3: Chapter 13 The Virtual Filesystem

Performance and protection in the ZoFS user-space NVM file system, SOSP'19

CrossFS: A Cross-layered Direct-Access File System, OSDI'20

Next lecture

Page Cache and Page Fault