Virtual File System

Changwoo Min

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
    - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management & address space

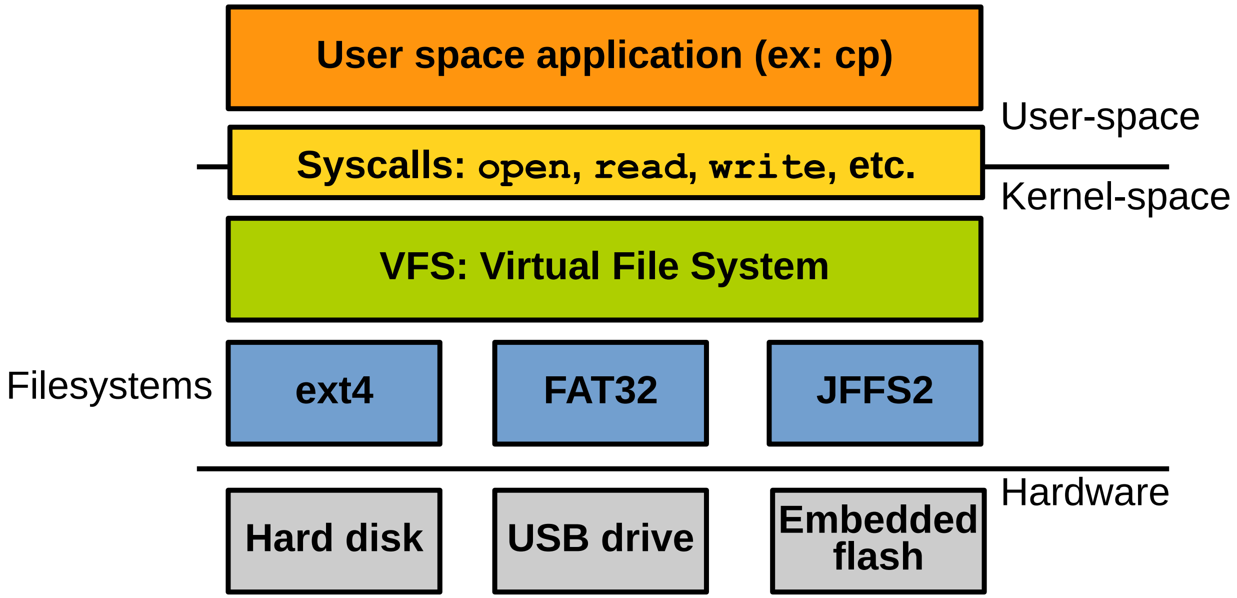
Today: virtual file system

- Introduction
- VFS data structures
- Filesystem data structures
- Process data structures

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 a HDD rootfs with ext4
- Allow them to *cooperate*
  - Example: a user can seamlessly copy a file between the FAT32 and Ext4 partitions

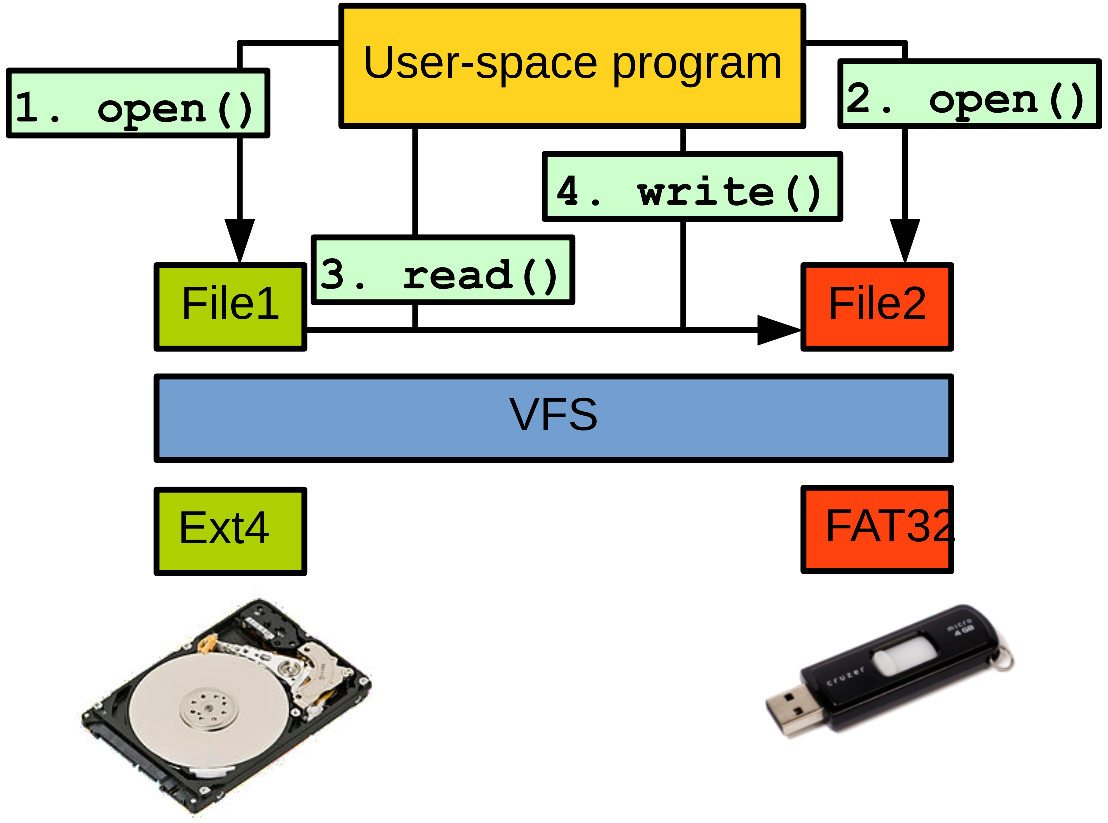
The Virtual File System (VFS)



Common filesystem interface

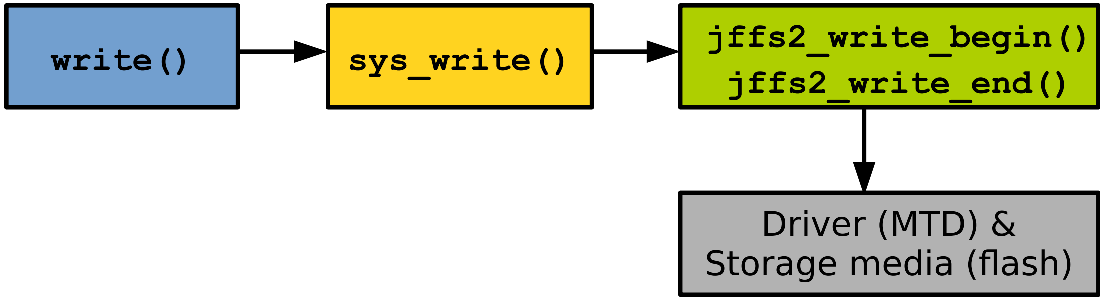
- VFS allows user-space to access files *independently* of the concrete filesystem they are stored on with a *common interface*
  - Standard system calls: `open()`, `read()`, `write()`, `lseek()`, etc.
  - “top” VFS interface (with user-space)
- Interface can work transparently between filesystems

Common filesystem interface



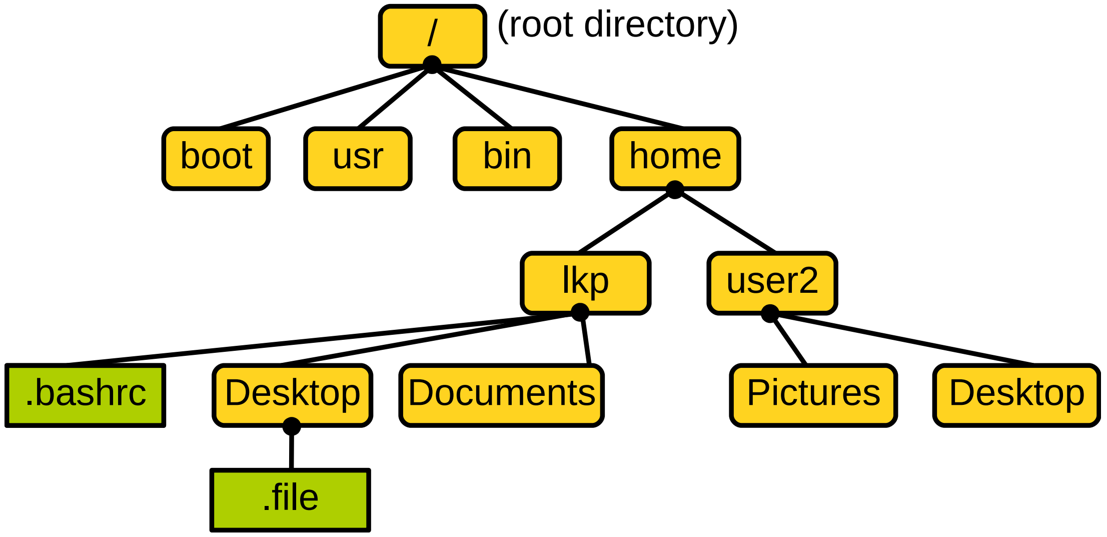
Filesystem abstraction layer

- 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



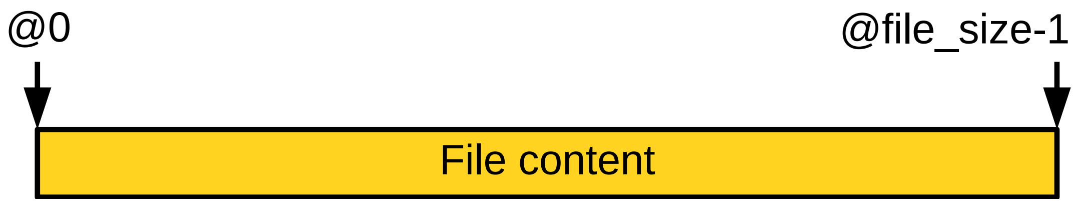
Unix filesystems

- 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*

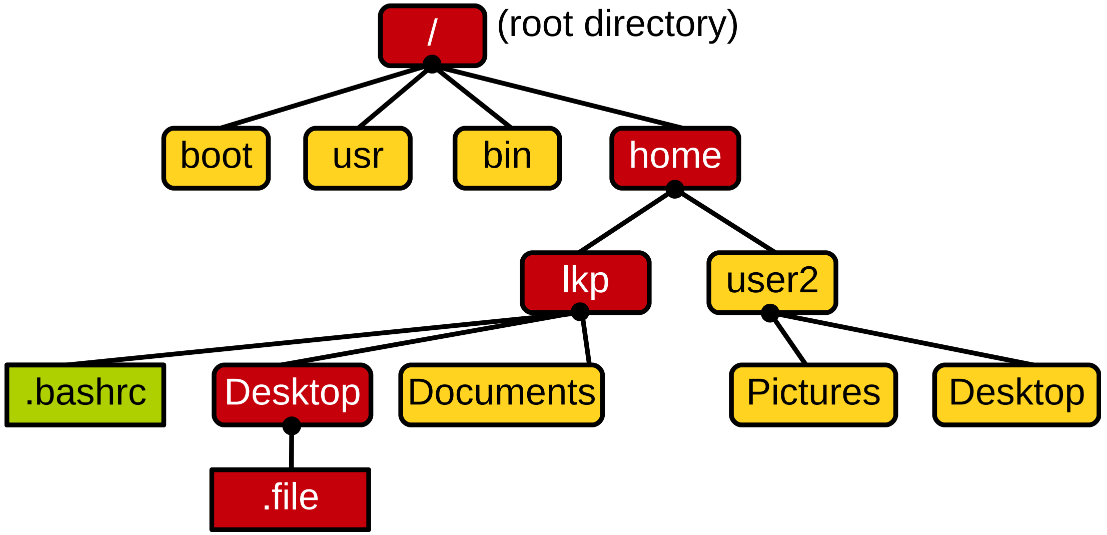


- **Directory:** folder containing files or other directories (sub-directories)
  - Sub-directories can be nested to create path:

    `/home/lkp/Desktop/file`

Unix filesystems

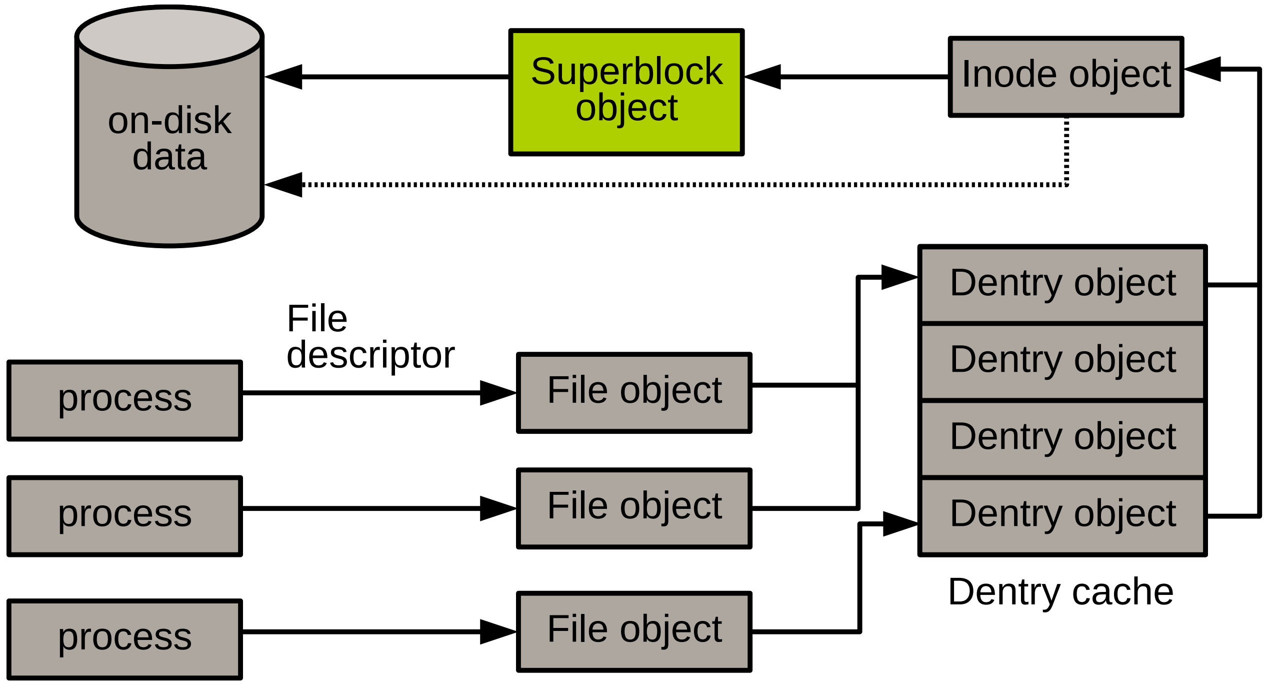
- Path example: `/home/lkp/Desktop/.file`



VFS data structures

- **dentry:** contains file/directory name and hierarchical links defining the filesystem directory tree
- **inode:** contains file/directory metadata
- **file:** contains information about a file opened by a process
- **superblock:** contains general information about the partition
- **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)
- 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`

```
/* linux/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

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

Superblock operations

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

```
/* linux/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);
    /* ... */
}
```

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 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()`

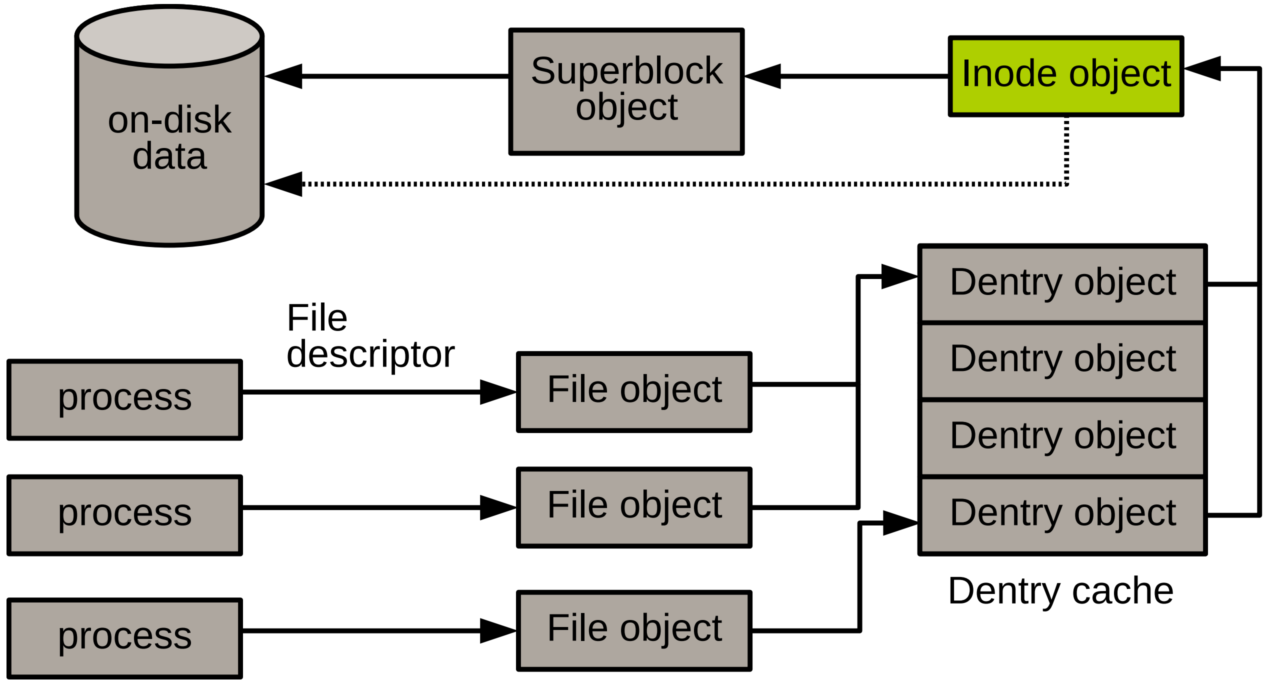
Superblock operations: `filesystem`

- `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`

Superblock operations: `filesystem`

- `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()`)
- **Q: where is the function to mount a file system?**
  - `mount_bdev()` in `fs/super.c`

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

- 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

inode

```
/* linux/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 */
    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 (contents or attributes) **/
    unsigned int            i_blkbits;      /* block size in bits */
    /* ... */
```

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
    - **Q: how does it return a new inode?**
- `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`

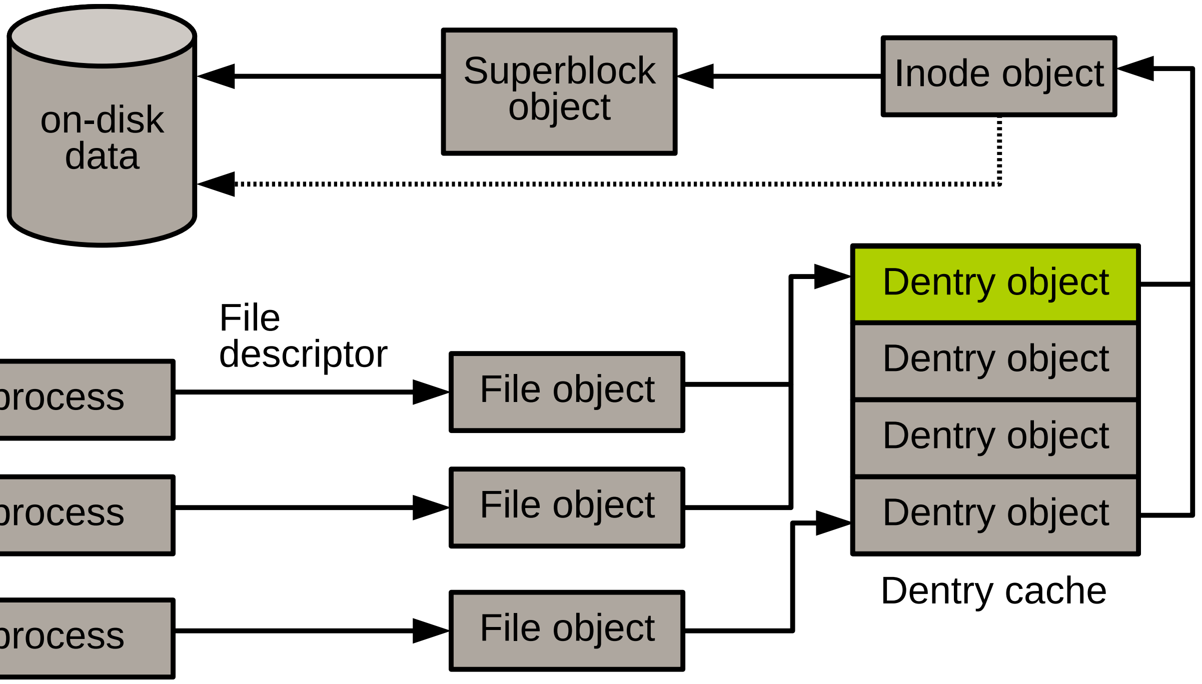
inode operations

- `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)`
    - Moves a file

dentry (or directory entry)



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

dentry

- Associated with a file or a directory to:
  - Store the file/directory `name`
  - Store its *location in the directory*
  - Perform directory specific operations, for example 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

dentry

- `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
- Dentries are constructed on demand and **kept in RAM for quick future pathname lookups**
    - **dentry cache** or **dcache**
- **Q: Why does Linux cache negative dentries?**

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 cache

- 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

dentry operations

```
/* linux/include/linux/dcache.h */
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, struct qstr *);
    int (*d_compare)(const struct dentry *,
            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_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

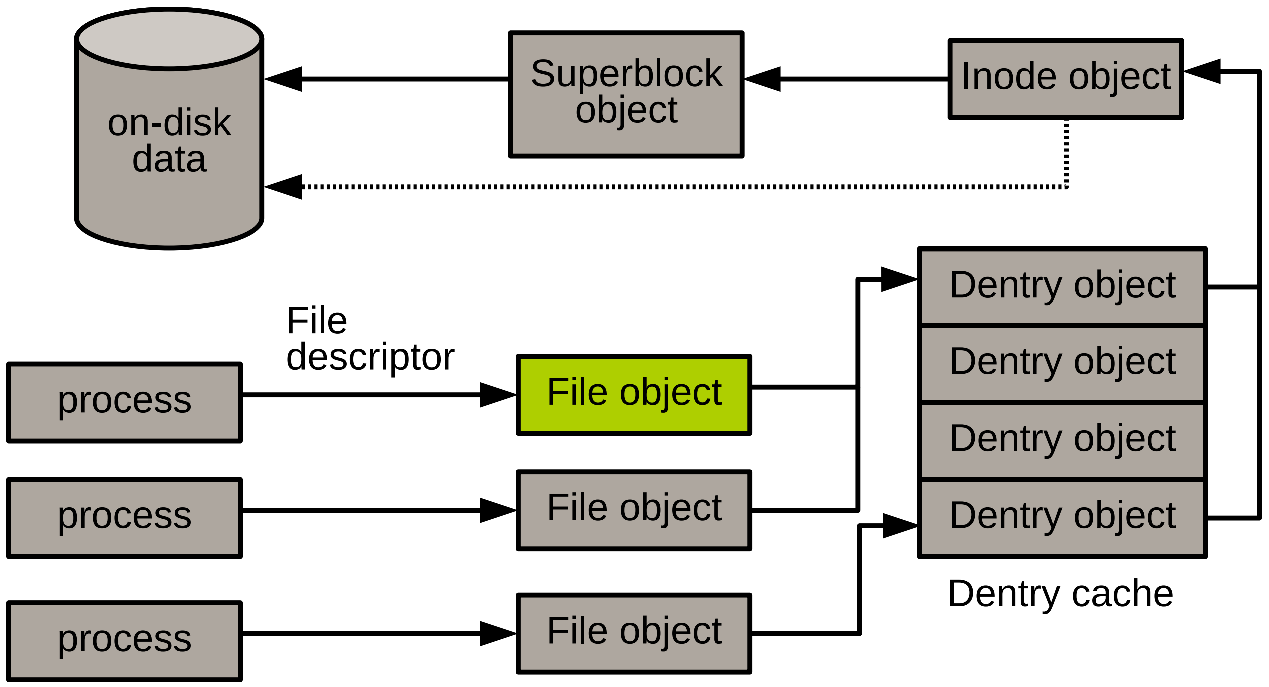
dentry operations

- `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`

dentry operations

- `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()`

File object


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

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

```
/* linux/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

```
/* linux/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

- `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

File operations

- `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
- `int flush(struct file *file)`
    - Called by VFS when the reference count of an open file decreases

File operations

- `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

File operations

- `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)

File operations

- `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
- `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

File operations

- `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

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 data structures

```
struct file_system_type {
    const char *name;     /** name: e.g., ext4 **/
    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 data structures

- When a filesystem is mounted, a `vfsmount` structure is created
    - Represent a specific instance of the filesystem: a mount point

```
/* linux/include/linux/fs.h */
struct vfsmount {
    struct dentry *mnt_root;    /* root of the mounted tree */
    struct super_block *mnt_sb; /* pointer to superblock */
    int mnt_flags;
};
```

Process data structure

- `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`

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
    - `struct address_space` : per-inode page cache

Summary

- Three key caches
  - dentry cache: `dentry_hashtable`, `dentry->d_hash`, `dentry->d_hash`
  - inode cache: `inode_hashtable`, `inode->i_hash`
  - page cache: `inode->i_mapping`

Further readings

- SFS: Random Write Considered Harmful in Solid State Drives, FAST12
- NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, FAST16
- Performance and protection in the ZoFS user-space NVM file system, SOSP19
- CrossFS: A Cross-layered Direct-Access File System, OSDI20

Next action

- 11/06: (4xxx)Project 4-1
- 11/10: Reading Assignment: lwCS

Next lecture

- Page Cache and Page Fault