Process Address Space

Have a look

- Before getting into **Process Address Space**, please take a look at
    - LK_Bird's Eye View sessions,

      S3 Virtual space,Privilege Level, System Call, Pagetable, Mapping.
      S4 MMU,Early Kernel,Add Types,Low High Memory.

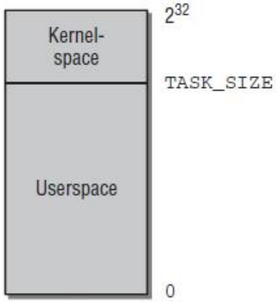
Process Address Space

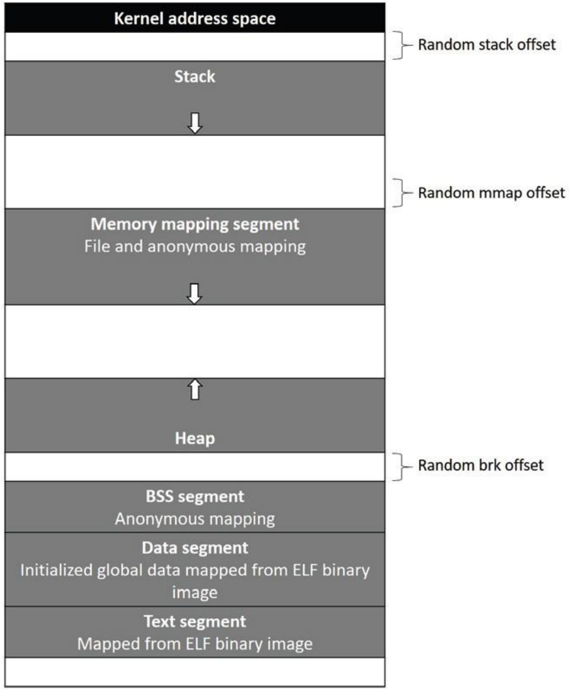
- Process memory segments,
  - **Random offsets** between the different sections.
  - **Stack** : A new frame is created on top of the stack upon entry into a called function, and is destroyed when the current function returns.

    i.e x86-32 by default **2MB**, Kernel usually offers an api to user-space to set process stack size attribute.

    In case process needs to expand its stack, such expansion is handled by the virtual memory manager through page faults: when the process attempts to touch an unmapped address at the top of the stack, the system triggers a page fault, which is handled by the kernel to extend the stack as long as the utilization within ***RLIMIT_STACK***.
  - **Memory Mapping Segment** (**Mmap**) : used for mapping file data from page cache or anonymous pages into process address space, also for mapping shared objects (**.SO**) or dynamic libraries.

    User-mode processes can initiate new mappings through the ***mmap()*** API.





Process Address Space

- Process memory segments, *Cont'd*
  - **Heap** : used for dynamic memory allocation that allows a process to store runtime data, the kernel provides ***brk()*** through user-mode processes to expand or shrink the heap at runtime.

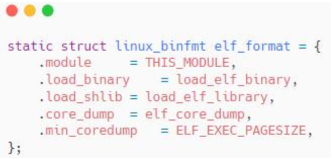
    i.e **GNU glibc** implements heap management that offers the ***malloc()*** family of functions for allocations.

  - **BSS, Data, and Text** : related to the binary image of the process,
    BSS : is set up through anonymous memory mapping.

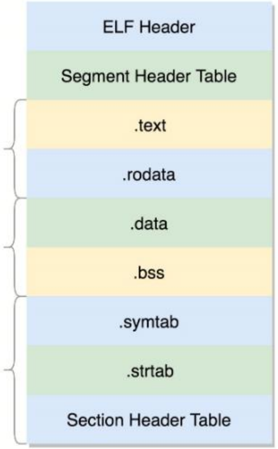
    Data : this mapping is created of type **private** memory mapping, which ensures that changes to data variables' memory are not reflected on the disk file.

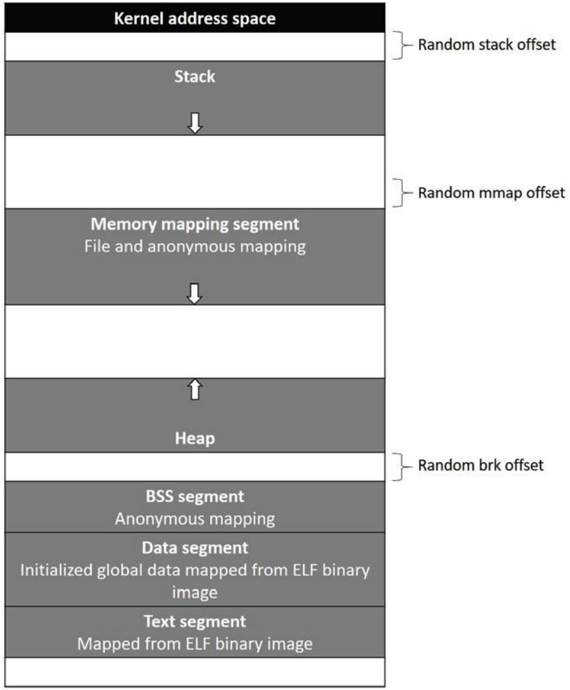
    Text : Mapped from .elf binary and the mapping here is cosidered as **read-only** mapping, resulting in a segmentation fault to be triggered on an attempt to write into this segment.

/fs/binfmt_elf.c



```
static struct linux_binfmt elf_format = {
    .module      = THIS_MODULE,
    .load_binary     = load_elf_binary,
    .load_shlib = load_elf_library,
    .core_dump  = elf_core_dump,
    .min_coredump    = ELF_EXEC_PAGESIZE,
};
```

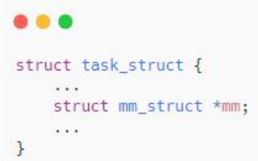




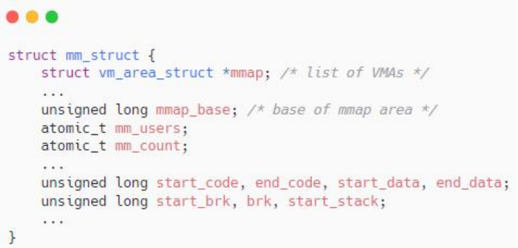
Process Address Space

- Kernel maintains all information on process memory segments and the corresponding translation table in ***struct mm_struct.***
- ***struct mm_struct***
  - Segments,
    mm_base : Start of the mmap segment in the VAS.
    mm_users : Holds the count of LWPs that share this memory descriptor.
    mm_count : number of processes currently using this descriptor, and the VM subsystem ensures that a structure is only released when ***mm_count = 0***.
    start_code, end_code, start_data, end_data : of virtual addresses for the code/data block mapped from the program's binary file.
    start_brk, brk : start and current end addresses of the heap segment.

  - Virtual Memory Areas,
    mmap : pointer to ***vm_area_struct*** points to a list of VMAs of the task.
    A **VM** area is any part of the **process virtual memory** space that has a special rule.

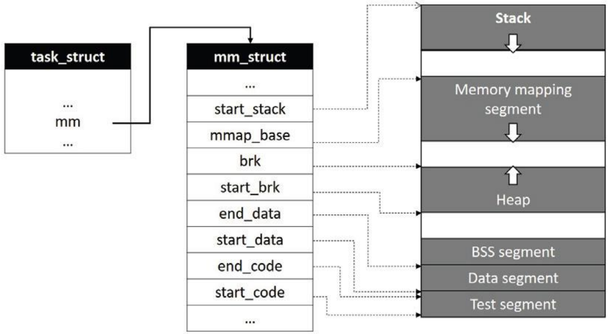


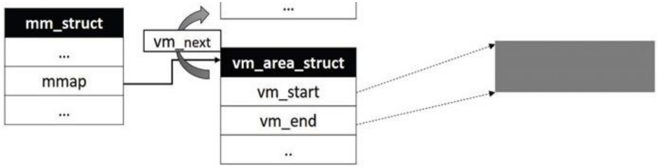
```
struct task_struct {
    ...
    struct mm_struct *mm;
    ...
}
```



```
struct mm_struct {
    struct vm_area_struct *mmap; /* list of VMAs */
    ...
    unsigned long mmap_base; /* base of mmap area */
    atomic_t mm_users;
    atomic_t mm_count;
    ...
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    ...
}
```

/include/linux/mm_types.h

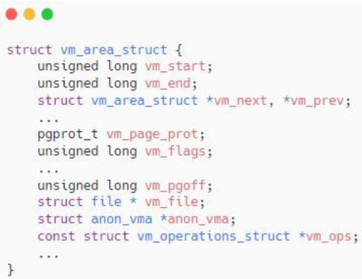




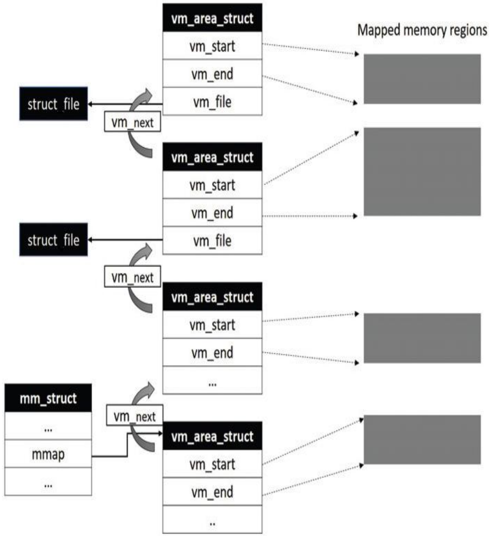
Process Address Space

- *vm_area_struct,*
  - *vm_start, vm_end* : start, end virtual address of the region.
  - *vm_next, vm_prev* : linked list of VM areas per task.
  - *vm_page_prot* : contains access permissions for the pages in the region r/w/x.
  - *vm_flags* : specify the type and change the behavior of the mapping.
  - *vm_file* : File we map to, can be NULL.
  - *anon_vma* : the VM subsystem groups all VMA instances of the process that represent anonymous memory regions into a list ***struct anon_vma.***
    So ***anon_vma***, access to all of the process VMAs that map anonymous pages.
  - *vm_ops* : contains operations performed on the current VMA.

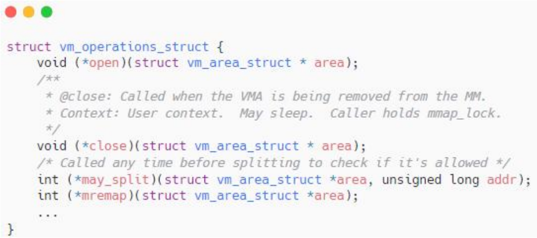
/include/linux/mm_types.h



```
struct vm_area_struct {
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next, *vm_prev;
    ...
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    ...
    unsigned long vm_pgoff;
    struct file * vm_file;
    struct anon_vma *anon_vma;
    const struct vm_operations_struct *vm_ops;
    ...
}
```



/include/linux/mm.h

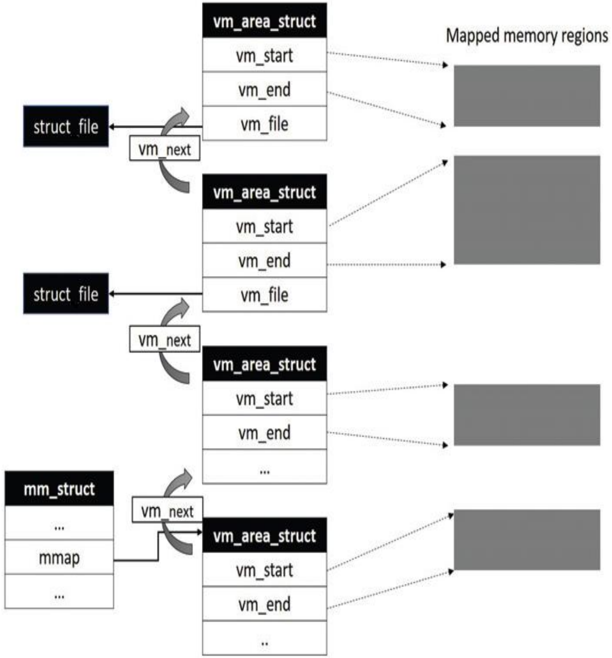


```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    /**
     * @close: Called when the VMA is being removed from the MM.
     * Context: User context.  May sleep.  Caller holds mmap_lock.
     */
    void (*close)(struct vm_area_struct * area);
    /* Called any time before splitting to check if it's allowed */
    int (*may_split)(struct vm_area_struct *area, unsigned long addr);
    int (*mremap)(struct vm_area_struct *area);
    ...
}
```

Process Address Space

- Big picture
  - Some memory regions mapped into the address space are file-backed (code regions form the application binary file, shared library, shared memory mappings, and so on).
  - Regions of the virtual address space such as heap, stack, and mmap are allocated through anonymous memory mappings ***anon_vma.***
  - The page cache tracks mappings to file regions by various user-mode process through an ***address_space***.
  - when a process forks a child, all anonymous pages of the caller address space are shared with the child process under copy-on-write (COW).

    This causes new VMAs to be created (for the child) that represent the same anonymous memory regions of the parent.



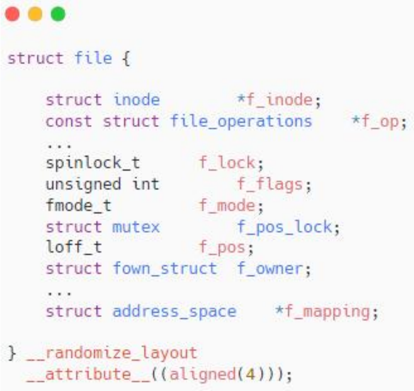
Process Address Space

- Big picture, *Cont'd*
  - ***file*** structure that ***vm_file*** refers to, represents an open file associated with a process, so it's considered local to a given process.
  - ***struct inode*** is a file-specific data structure.

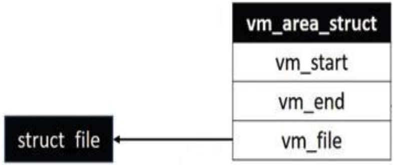
    This allows multiple processes to access the same file without directly interfering with the other processes.

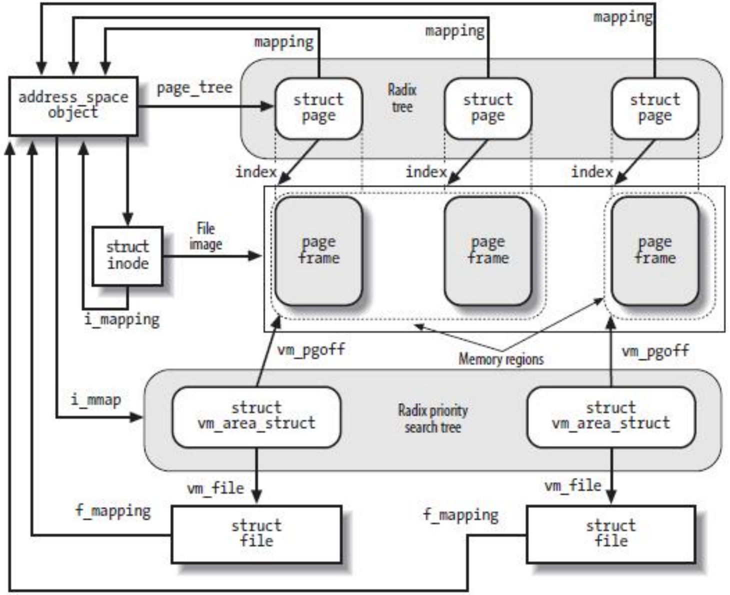
    *that's why, we see more than one **struct file** may refer to one **struct address_space**.*

/include/linux/fs.h



```
struct file {

    struct inode          *f_inode;
    const struct file_operations    *f_op;
    ...
    spinlock_t        f_lock;
    unsigned int          f_flags;
    fmode_t           f_mode;
    struct mutex          f_pos_lock;
    loff_t            f_pos;
    struct fown_struct  f_owner;
    ...
    struct address_space    *f_mapping;

} __randomize_layout
  __attribute__((aligned(4)));
```





Process Address Space

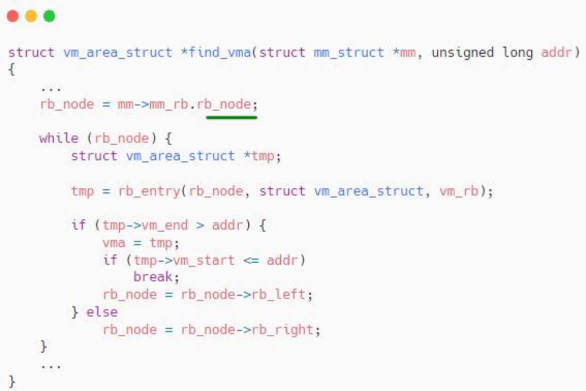
- Managing virtual memory
  - *find_vma()* : locates the first region in the VMA list that satisfies the condition for a given address, such that (addr < vm_area_struct->vm_end).

    As a workaround, the VM subsystem maintains a red-black tree for efficient access of ***vm_area_struct*** regions.

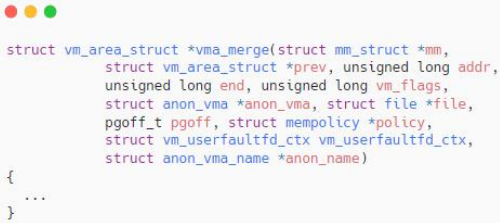
  - *vma_merge()* : When a new VMA is mapped immediately before or after an existing VMA with identical access attributes and data from a file-backed memory region, it is more optimal to merge them into a single VMA structure.

  - *do_mmap()* : this functionality is exported to user-space via the ***mmap()*** system call or second variant ***mmap2()***.

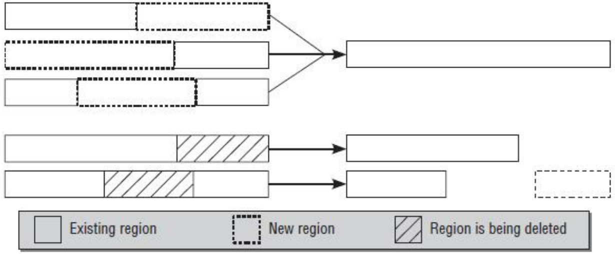
/mm/mmap.c



```
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    ...
    rb_node = mm->mm_rb.rb_node;

    while (rb_node) {
        struct vm_area_struct *tmp;

        tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);

        if (tmp->vm_end > addr) {
            vma = tmp;
            if (tmp->vm_start <= addr)
                break;
            rb_node = rb_node->rb_left;
        } else
            rb_node = rb_node->rb_right;
    }
    ...
}
```



```
struct vm_area_struct *vma_merge(struct mm_struct *mm,
            struct vm_area_struct *prev, unsigned long addr,
            unsigned long end, unsigned long vm_flags,
            struct anon_vma *anon_vma, struct file *file,
            pgoff_t pgoff, struct mempolicy *policy,
            struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
            struct anon_vma_name *anon_name)
{
  ...
}
```



Process Address Space

- Managing virtual memory, *Cont'd*
  - ***do_mmap()**, Cont'd*

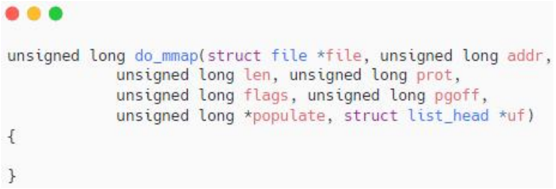
    creates a new linear address interval to a process's address space, if the created address interval is adjacent to an existing address interval and share the same permissions, the two intervals are merged into one.
    If not possible, a new VMA (***vm_area_struct***) is created.

    it maps the file specified by ***file*** at offset ***offset*** for length ***len*** (file-backed mapping).
    if ***file (NULL)*** : anonymous mapping.

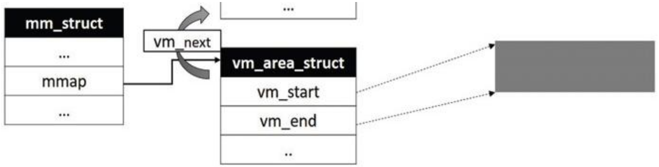
    prot : the access permissions for pages in the memory area.
    flags : specify the type and change the behavior of the mapping.

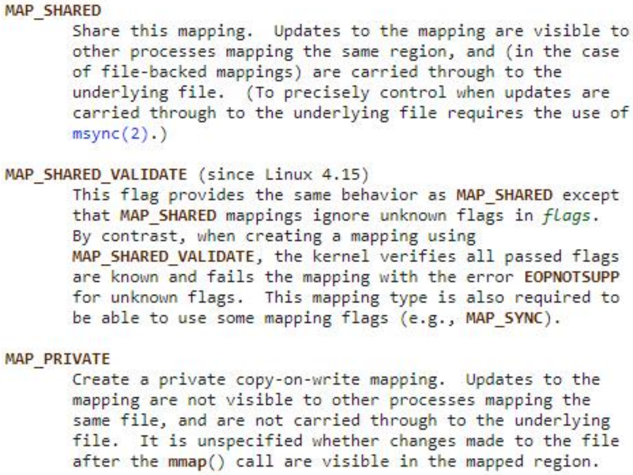
    https://man7.org/linux/man-pages/man2/mmap.2.html
    for all prot/flags.

/mm/mmap.c



```
unsigned long do_mmap(struct file *file, unsigned long addr,
                unsigned long len, unsigned long prot,
                unsigned long flags, unsigned long pgoff,
                unsigned long *populate, struct list_head *uf)
{

}
```





```
MAP_SHARED
          Share this mapping.  Updates to the mapping are visible to
          other processes mapping the same region, and (in the case
          of file-backed mappings) are carried through to the
          underlying file.  (To precisely control when updates are
          carried through to the underlying file requires the use of
          msync(2).)

MAP_SHARED_VALIDATE (since Linux 4.15)
          This flag provides the same behavior as MAP_SHARED except
          that MAP_SHARED mappings ignore unknown flags in flags.
          By contrast, when creating a mapping using
          MAP_SHARED_VALIDATE, the kernel verifies all passed flags
          are known and fails the mapping with the error EOPNOTSUPP
          for unknown flags.  This mapping type is also required to
          be able to use some mapping flags (e.g., MAP_SYNC).

MAP_PRIVATE
          Create a private copy-on-write mapping.  Updates to the
          mapping are not visible to other processes mapping the
          same file, and are not carried through to the underlying
          file.  It is unspecified whether changes made to the file
          after the mmap() call are visible in the mapped region.
```

Process Address Space

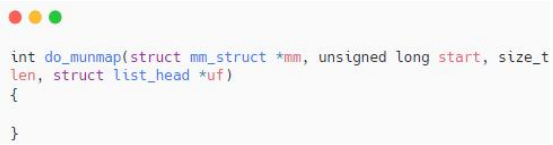
- Managing virtual memory, *Cont'd*
  - ***do_munmap()*** : removes an address interval from a specified process address space.

    mm : which process.
    start, len : specifies the interval.

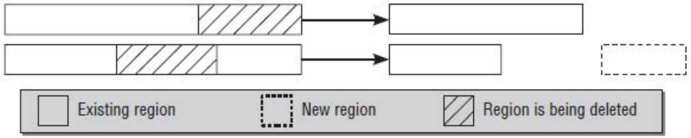
    The ***munmap()*** system call is exported to user-space to enable processes to remove address intervals from their address space.

    user-space :
    int munmap(void *start, size_t length)

/mm/mmap.c

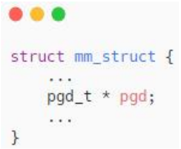


```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t
len, struct list_head *uf)
{

}
```

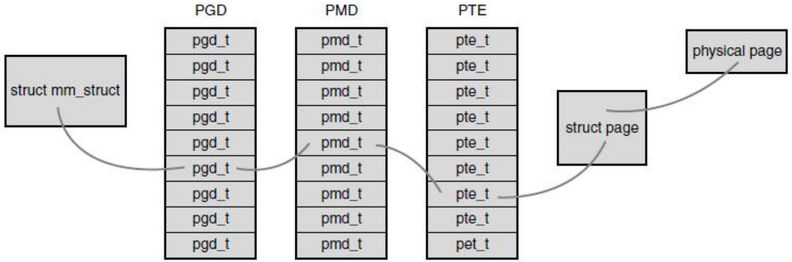


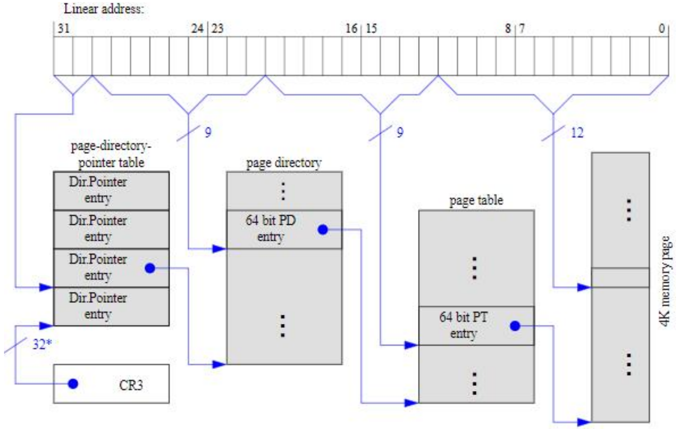
Process Address Space

- Page Tables
  - After mapping a file/anon to the virtual address space of process, **page table** is used to virtual-to-physical mappings.
  - Most architectures offer, three-level page table,
    PGD : page global directory.
    PMD : page middle directory.
    PTE : page table entry.
  - Each process has its own page tables (threads share them, of course).
  - Looking up all these addresses in memory can be done so quickly by a translation lookaside buffer (TLB) HW cache.
  - Page table for x86-64, the process page table is loaded by copying ***mm_struct->pgd*** into the ***cr3*** register which has the side effect of flushing the HW TLB.



```
struct mm_struct {
    ...
    pgd_t * pgd;
    ...
}
```





Process Address Space

- Page Tables, *Cont'd*
  - The following functions must be implemented by all **architectures** to enable memory management code to create and destroy page tables.
    mk_pte(page,prot) : Creates a pte entry; a page instance and the desired page access permissions must be passed as parameters.

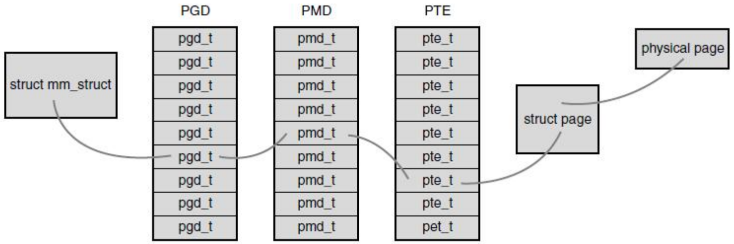
    pte_page(pte) : yields the address of the page instance belonging to the page described by the page table entry.

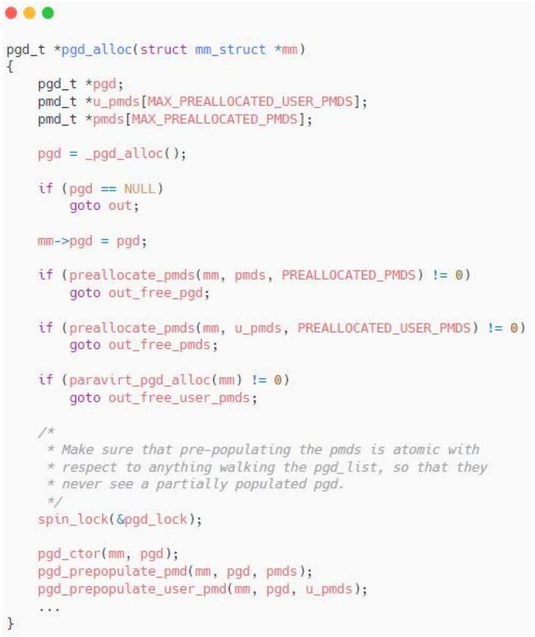
    pgd_alloc(), pmd_alloc(), pte_alloc() : Reserve and initialize memory to hold a complete page table (not just a single entry).

    pgd_free(), ***pmd_free()***, ***pte_free()*** : Free the memory occupied by the page table.

    set_pgd(), ***set_pmd()***, ***set_pte()*** : Set the value of an entry in a page table.

    i.e. take a look at */arch/x86/mm/pgtable.c*





```
pgd_t *pgd_alloc(struct mm_struct *mm)
{
    pgd_t *pgd;
    pmd_t *u_pmds[MAX_PREALLOCATED_USER_PMDS];
    pmd_t *pmds[MAX_PREALLOCATED_PMDS];

    pgd = _pgd_alloc();

    if (pgd == NULL)
        goto out;

    mm->pgd = pgd;

    if (preallocate_pmds(mm, pmds, PREALLOCATED_PMDS) != 0)
        goto out_free_pgd;

    if (preallocate_pmds(mm, u_pmds, PREALLOCATED_USER_PMDS) != 0)
        goto out_free_pmds;

    if (paravirt_pgd_alloc(mm) != 0)
        goto out_free_user_pmds;

    /*
     * Make sure that pre-populating the pmds is atomic with
     * respect to anything walking the pgd_list, so that they
     * never see a partially populated pgd.
     */
    spin_lock(&pgd_lock);

    pgd_ctor(mm, pgd);
    pgd_prepopulate_pmd(mm, pgd, pmds);
    pgd_prepopulate_user_pmd(mm, pgd, u_pmds);
    ...
}
```

Filesystems & Virtual Filesystems

Have a look

- Before getting into **Filesystems** & **Virtual Filesystems**, please take a look at
  - *Prof. Ahmed Elarabawy Course.*

    Course 102: Lecture 5: File Handling Internals
    Course 102: Lecture 26: FileSystems in Linux (Part 1)
    Course 102: Lecture 27: FileSystems in Linux (Part 2)
    Course 102: Lecture 28: Virtual FileSystems

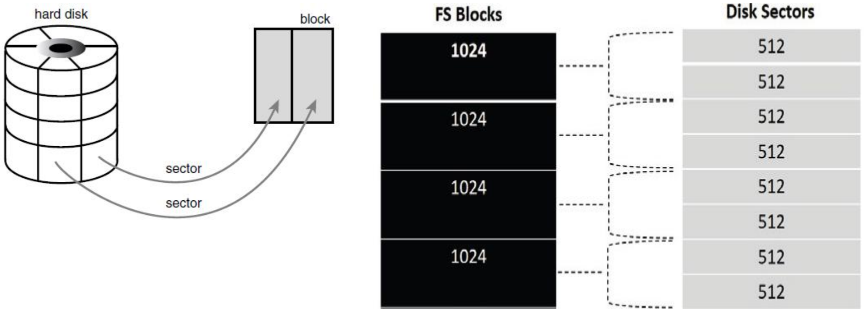
Filesystems

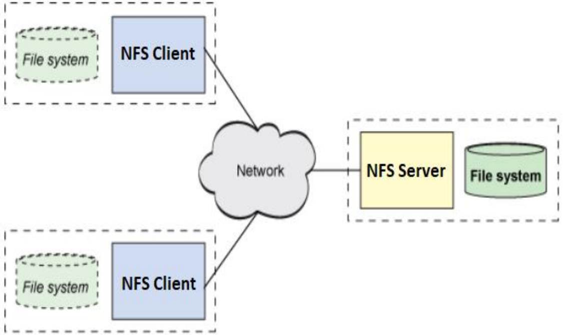
- Filesystem Types
  - **Disk-based** filesystems : i.e. Ext2/3, Reiserfs, FAT. From the filesystem point of view, the underlying devices are nothing more than a list of storage **blocks** for which an appropriate organization scheme must be adopted.**block** may be one sector or multiples of **sector** size.

  - **Virtual filesystems** : Generated in the kernel itself, i.e. *proc* FS, it requires no storage space on any HW device. Instead, the kernel creates a **hierarchical** file structure whose entries contain information on a particular part of the system.

  - **Network** filesystems : It's between disk-based and virtual filesystems, all operations on **files** in this filesystem are carried out over a **network** connection. Nevertheless, the kernel needs information on the size of files, their position within the **directory hierarchy.**

    So as a result of the VFS layer, userspace processes see no difference between a local filesystem and a filesystem available only via a network.





Filesystems

- Metadata
    - Inode

      contain metadata info to a file such as filename, type of file, last access timestamp, owner, access privileges, last modification timestamp, creation time, size of file data, and references to disk blocks containing file data.

      Filesystems reserve a few disk **blocks** for storing inode instances and the rest for storing corresponding file data.

      The on-disk list of all **inodes** held in these **blocks** called the **inode table**.

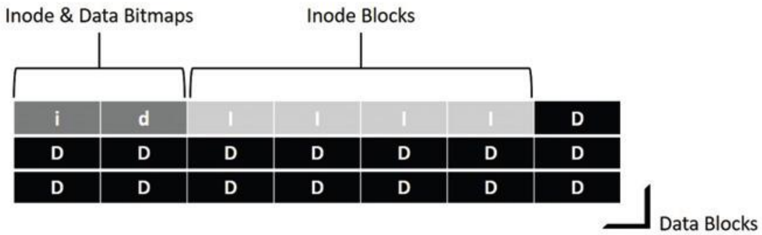
      Filesystems tracks the status of the inode and data blocks through ***bitmaps*** (free inodes and data blocks).

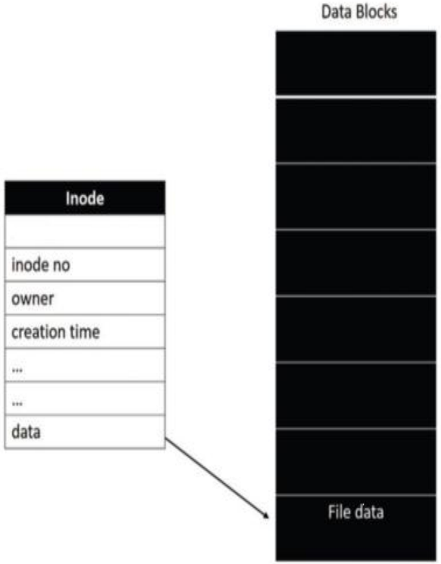
    - Data block map

      inode should record the locations of data blocks in which corresponding file data is stored.

      it uses differnet ways in order to point to the data blocks of the file.

Disk Storage Layout





Filesystems

- Metadata, *Con'td*
  - Data block map, *Cont'd*

    - **Direct Pointers** : The number of such direct pointers would depend on filesystem design.

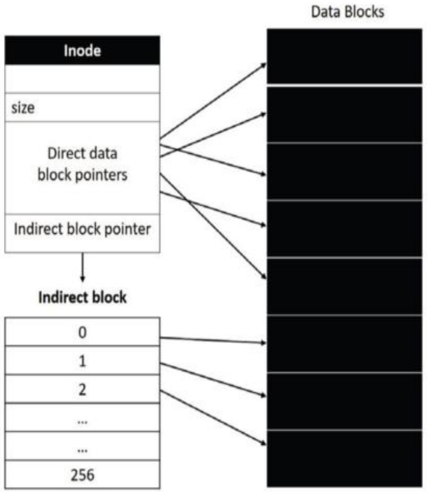
    This method is productive for small files which span a few data blocks.

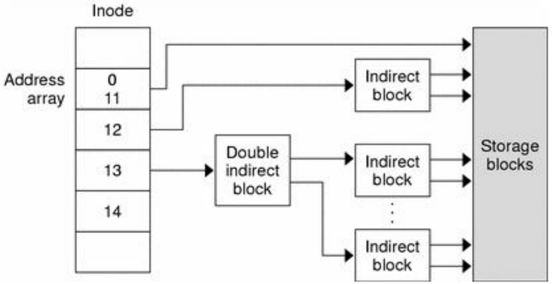
    - **Indirect Pointers** : refers to a block containing direct pointers to data blocks of the file.

    To support more larger files, **double-indirect** pointer, which refers to a block containing indirect pointers with each entry referring to a block containing direct pointers.

    This technique can be extended with a **triple-indirection** pointer, resulting in even more metadata to be managed by filesystems.

    - **Extent Lists** : You can check this article https://www.linux.org/threads/intro-to-extents.8625/





Filesystems

- Metadata, *Con'td*
  - Directories : Filesystems consider it as a special file. ***type*** field, which is marked as directory.

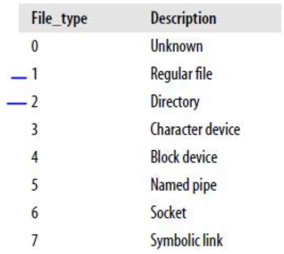
    Each directory is assigned data blocks where it holds information about files and subdirectories it contains represented as (name : inode).

    File name length is defined by the filesystem's naming policy.

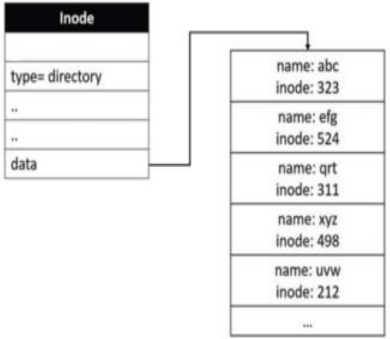
    Example of **Ext2** filesystem directory structure stored in disk blocks.

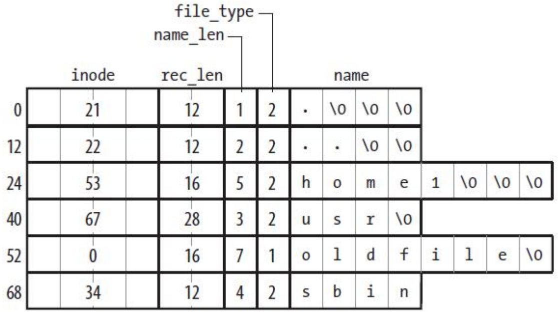
  - Superblock : filesystems also need to maintain metadata with respect to disk volume as a whole, such as size of the volume, total block count, current state of filesystem, count of inode blocks, count of inodes, count of data blocks,...

    During initialization of filesystem on disk volume, the superblock is organized at start of disk storage.



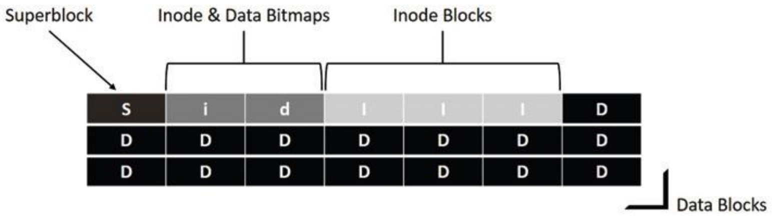
File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link





	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

Disk Storage Layout



Filesystems

- Each Filesystem (Ext2,...) should implement some basic functionalties of VFS abstraction layer related to (*inode, super block, dir, file*), will discuss them later in VFS layer.

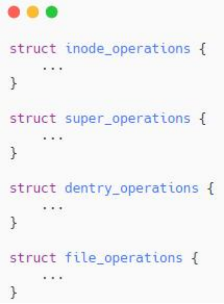
- The VFS layer builds called rootfs, under which all filesystems can enumerate their directories and files.

- General Filesystems Operations
  - Mount : operation of enumerating an on-disk superblock and metadata into memory for the filesystem's use.

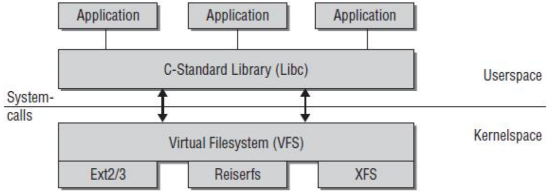
    It creates in-memory data structures that describe file metadata and present the host operating system with a view of the directory and file layout in the volume.

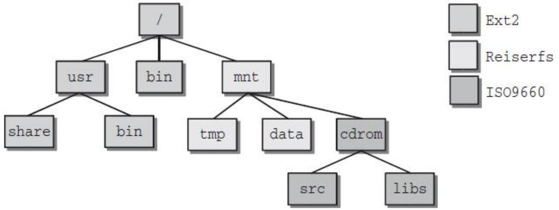
  - Unmount : operation of flushing the **in-memory** state of filesystem data structures back **to disk**.

    The ***superblock*** contains the state of the filesystem; it indicates whether the volume is **consistent** or **dirty**.

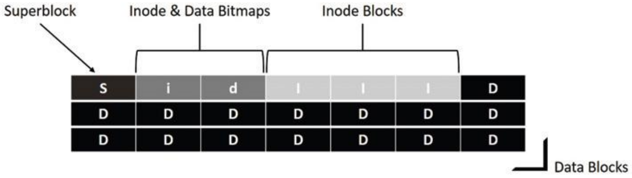


```
struct inode_operations {
    ...
}

struct super_operations {
    ...
}

struct dentry_operations {
    ...
}

struct file_operations {
    ...
}
```





Disk Storage Layout



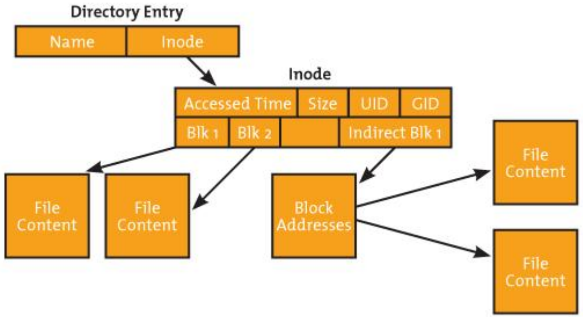
Filesystems

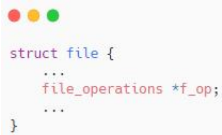
- General Filesystems Operations, *Cont'd*
  - File creation and deletion
    Creation : Instantiation of a new inode with appropriate attributes (***filename***, ***directory*** under which file is to be created, ***access permissions*** for users,...).

    Deletion : release its data blocks to the list of free data blocks, and inode to list of free inodes.
    But before doing this, checks the file's reference count to determine the number of **processes** currently using the file.

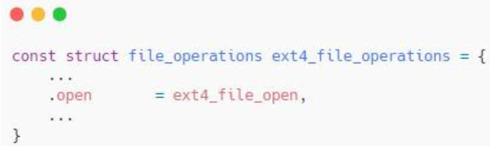
  - File open and close
    Open : Once a process triggers ***open*** operation it invokes the ***open()*** of the filesystem, that traverse until gets ***inode*** number of the specified file, then instantiates a ***file*** structure related to the requestd **process.**

    Close : the ***file*** structure is destroyed and the file's reference count is decremented.
    The caller **process** will no longer be able to initiate any other file operation until it can open the file all over again.





```
struct file {
    ...
    file_operations *f_op;
    ...
}
```



```
const struct file_operations ext4_file_operations = {
    ...
    .open       = ext4_file_open,
    ...
}
```

/fs/ext4/file.c

Filesystems

- General Filesystems Operations, *Cont'd*
  - File read and write

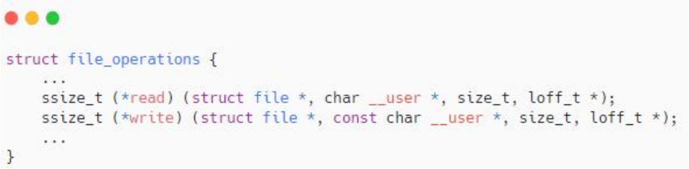
    Read : Once a process triggers ***read*** operation, filesystem's ***read()*** routine is invoked.

    Operations begin with a lookup into the file's **data block** map to locate the appropriate data disk **sector** to be read, then allocates a ***page*** from the **page cache** and schedules disk I/O (**Block Layer**).

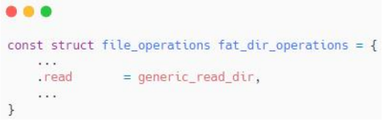
    On completion of I/O transfer, the filesystem moves requested data into the application's buffer and updates the file offset position in the process's ***file*** structure.

    Write : retrieves data passed from user buffer and writes it into the appropriate offset of file buffer in the page cache, and marks the page with the ***PG_dirty*** flag.

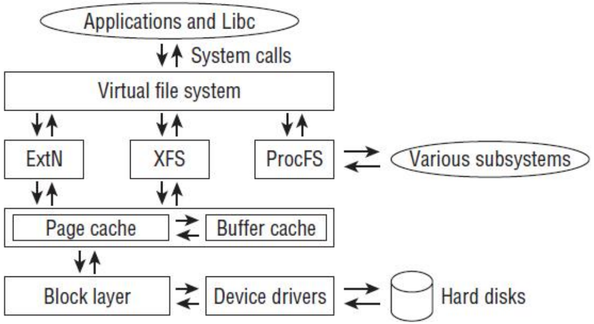
/include/linux/fs.h

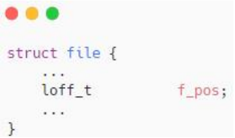


```
struct file_operations {
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
}
```



```
const struct file_operations fat_dir_operations = {
    ...
    .read        = generic_read_dir,
    ...
}
```





```
struct file {
    ...
    loff_t          f_pos;
    ...
}
```

Filesystems

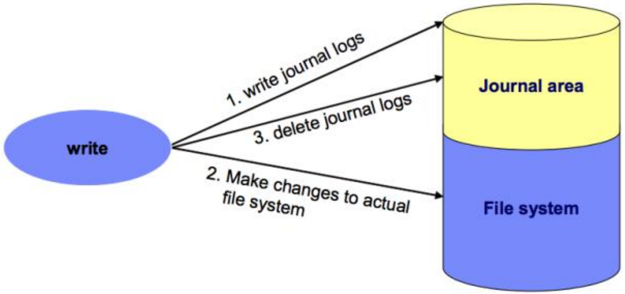
- General Filesystems Operations, *Cont'd*
  - Filesystem consistency and crash recovery
    Possibility to occur (power down, OS crash,...), causing interruption of a partially committed critical update.

    This results in corruption of on-disk structures and leaves the filesystem in an inconsistent state.

    Journaling : is a technique implemented by most modern filesystem for quick and reliable crash recovery.

    Journaling idea is to prepare a log (note) listing out changes to be committed to the on-disk image of the filesystem, and writing the log to a special disk block called a **journal block**, before beginning the actual update operation.

    So, the filesystem can easily detect inconsistencies and fix them by looking through information recorded in the log.

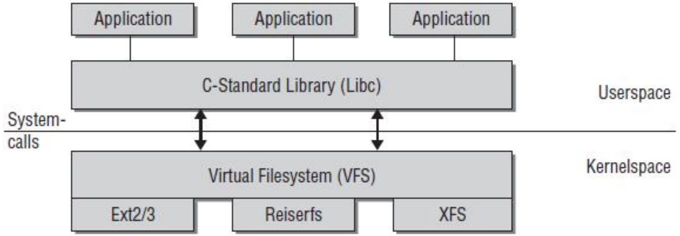


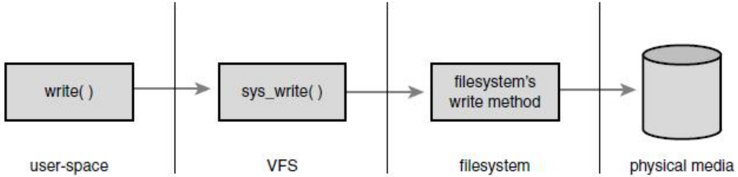
Virtual Filesystem (VFS)

- VFS is the subsystem of the kernel that implements the file and filesystem-related interfaces (Common Filesystem Interface) provided to user-space programs.

- VFS enables system calls such as ***open()***, ***read()***, and ***write()*** to work regardless of the filesystem or underlying physical medium (Filesystem abstraction).

- Flow of data from user-space,
  issuing a ***write()*** call, through the VFS's generic system call, into the filesystem's specific write method, and finally
  arriving at the physical media.





Virtual Filesystem (VFS)

- VFS Objects & Data structures
  - Primary objects

    inode : represents a specific file in the system.

    dentry : represents a directory entry, which is a single component of a path.

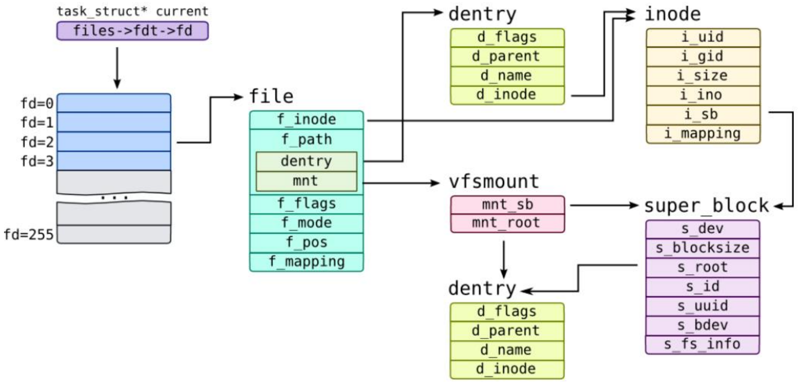
    i.e. Pathname Lookup

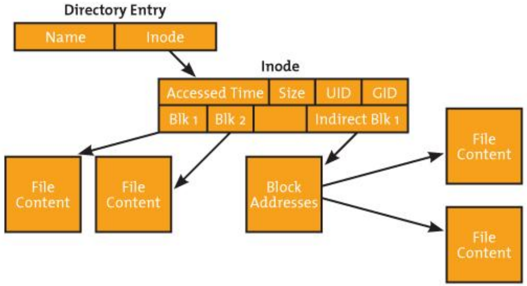
    when looking up the ***/tmp*** pathname, the kernel creates a dentry object for the ***/*** root directory, a second dentry object for the ***tmp*** entry of the root directory,

    Lookup examines the entry matching the first name to derive the corresponding **inode**.

    Then the directory file that has that inode is read from disk and the entry matching the second name is examined to derive the corresponding **inode**. This procedure is repeated for each name included in the path.

    The **dentry cache** considerably speeds up the procedure, because it keeps the most recently used dentry **objects** in memory.





Virtual Filesystem (VFS)

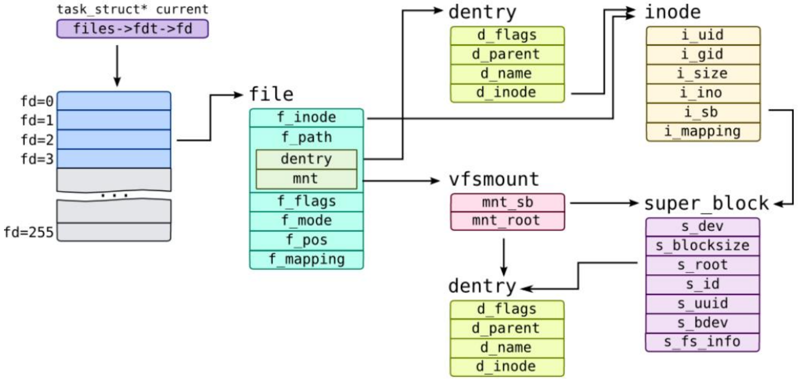
- VFS Objects & Data structures, *Cont'd*
  - Primary objects, *Cont'd*
    super_block : info describe the disk volume as a whole which is related to a specific mounted filesystem.

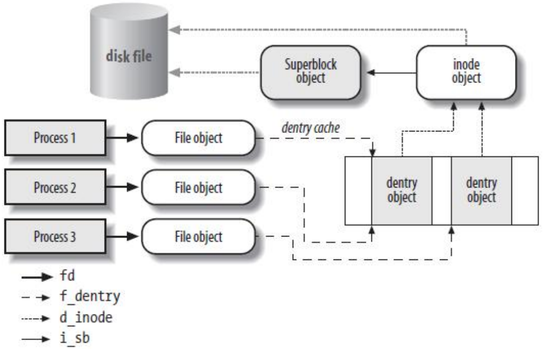
    ***file** :* represents open file associated with a process.
  - Operations objects : pointers injected in primary objects.
    inode_operations* :** methods, the kernel invokes on a specific filesystem ***create() and ***link().***

    dentry_operations : invoked on a specific directory entry, such as ***d_compare()*** and ***d_delete().***

    super_operations : invoked on a specific filesystem, such as ***write_inode()*** and ***sync_fs().***

    file_operations : methods that a process can invoke on an open file, such as ***read()*** and ***write().***





Virtual Filesystem (VFS)

- VFS Structures Details
  - *super_block* : Each FS needs to create an object of *super_block* to fill in its superblock details during **mount**.

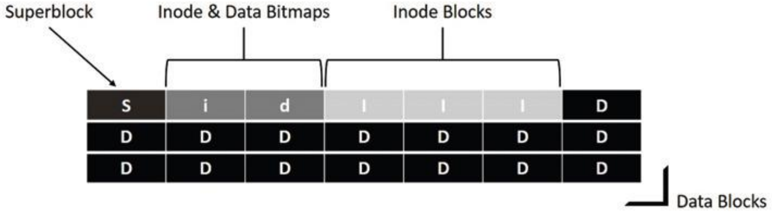
    s_list : list of mounted superblocks.

    s_dev : device ID.

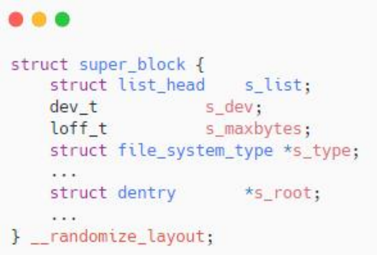
    s_maxbytes : max file size.

    s_type : points to *file_system_type* type.

    s_root : points to the dentry object of the filesystem's root directory.



/include/linux/fs.h



```
struct super_block {
    struct list_head    s_list;
    dev_t               s_dev;
    loff_t              s_maxbytes;
    struct file_system_type *s_type;
    ...
    struct dentry       *s_root;
    ...
} __randomize_layout;
```

Virtual Filesystem (VFS)

- VFS Structures Details, *Cont'd*
  - ***super_operations***
    alloc_inode() : create and allocate space for the new ***inode*** object and initialize it under the superblock.

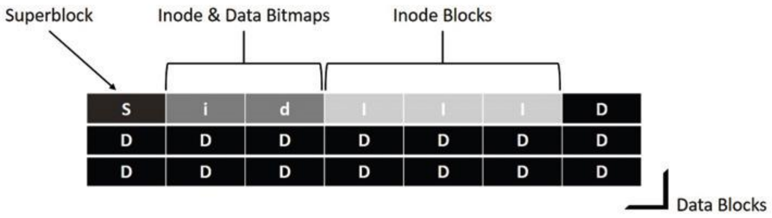
    write_inode() : write an inode on to the disk.

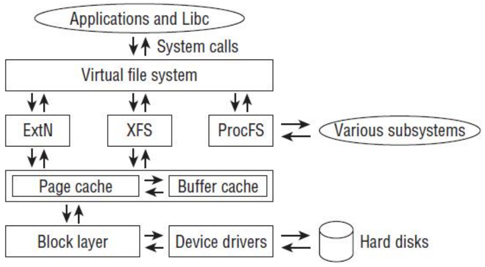
    put_super() : when VFS needs to free the superblock.

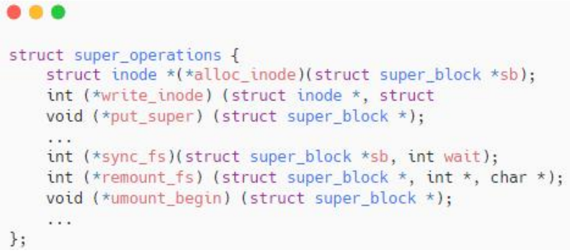
    sync_fs() : invoked to synchronize filesystem data with that of the underlying block device.

    remount_fs() : when the filesystem needs to be remounted,
    This is commonly used to change the mount flags for a filesystem, especially to make a readonly filesystem writeable.

    unmount_begin() : when the VFS is unmounting a filesystem.







```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    int (*write_inode) (struct inode *, struct
    void (*put_super) (struct super_block *);
    ...
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*umount_begin) (struct super_block *);
    ...
};
```

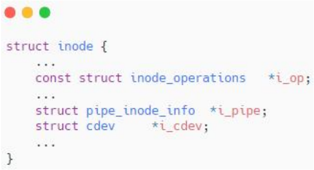
/include/linux/fs.h

Virtual Filesystem (VFS)

- VFS Structures Details, *Cont'd*
  - *inode* : contains all info to manipulate a file/directory (normal file, special files such as device files or pipes).
  - *inode_operations*
    *struct dentry * lookup(struct inode *dir, struct dentry *dentry, int flags)* : searches a directory for a filename specified in the given dentry.

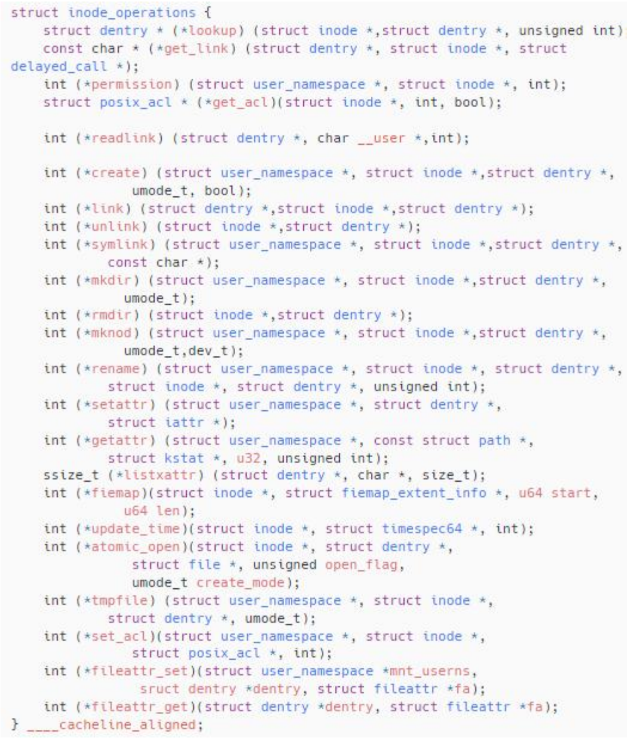
    Assume, dir *(/tmp)* , and lookup for a filename(*f1*) while the kernel created on-the-fly ***denty*** and fill it with filename(*f1*) to ease the search, So the kernel checks whether the ***dentry*** that coupled with the filename is one of the directory components.

    *int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)* : create a hard link of the file ***old_dentry*** in the directory ***dir*** with the new filename ***dentry***.



```
struct inode {
    ...
    const struct inode_operations   *i_op;
    ...
    struct pipe_inode_info  *i_pipe;
    struct cdev     *i_cdev;
    ...
}
```

/include/linux/fs.h



```
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int)
    const char * (*get_link) (struct dentry *, struct inode *, struct
delayed_call *);
    int (*permission) (struct user_namespace *, struct inode *, int);
    struct posix_acl * (*get_acl)(struct inode *, int, bool);

    int (*readlink) (struct dentry *, char __user *,int);

    int (*create) (struct user_namespace *, struct inode *,struct dentry *,
              umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct user_namespace *, struct inode *,struct dentry *,
            const char *);
    int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *,
             umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct user_namespace *, struct inode *,struct dentry *,
             umode_t,dev_t);
    int (*rename) (struct user_namespace *, struct inode *, struct dentry *,
            struct inode *, struct dentry *, unsigned int);
    int (*setattr) (struct user_namespace *, struct dentry *,
            struct iattr *);
    int (*getattr) (struct user_namespace *, const struct path *,
            struct kstat *, u32, unsigned int);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
             u64 len);
    int (*update_time)(struct inode *, struct timespec64 *, int);
    int (*atomic_open)(struct inode *, struct dentry *,
              struct file *, unsigned open_flag,
              umode_t create_mode);
    int (*tmpfile) (struct user_namespace *, struct inode *,
            struct dentry *, umode_t);
    int (*set_acl)(struct user_namespace *, struct inode *,
              struct posix_acl *, int);
    int (*fileattr_set)(struct user_namespace *mnt_userns,
               sruct dentry *dentry, struct fileattr *fa);
    int (*fileattr_get)(struct dentry *dentry, struct fileattr *fa);
} ____cacheline_aligned;
```

Virtual Filesystem (VFS)

- VFS Structures Details, *Cont'd*
  - ***inode_operations, Con'td***

    int symlink(struct user_namespace *mnt_userns, struct inode *dir, struct dentry *dentry, const char *symname) :

    create a symbolic link named ***symname*** to the file represented by ***dentry*** in the directory ***dir***.

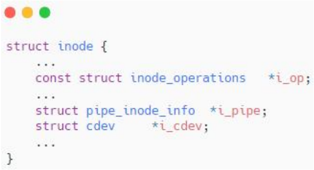
    int mkdir(struct user_namespace *mnt_userns, struct inode *dir, struct dentry *dentry, umode_t mode) :

    create a new directory.

    int mknod(struct user_namespace *mnt_userns, struct inode *dir, struct dentry *dentry, umode_t mode, dev_t rdev) :

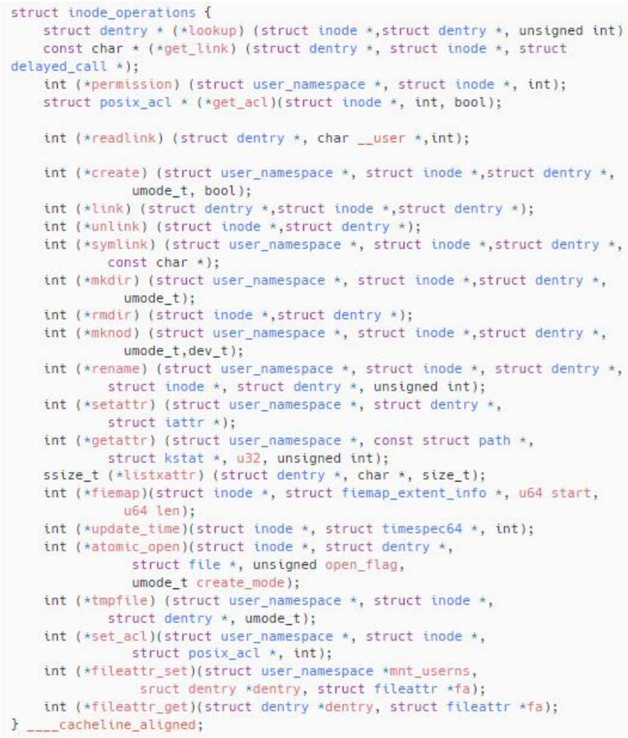
    create a special file (device file, pipe, or socket).

    The file is referenced by the device ***rdev*** and the name ***dentry*** in the directory ***dir***.



```
struct inode {
    ...
    const struct inode_operations   *i_op;
    ...
    struct pipe_inode_info  *i_pipe;
    struct cdev     *i_cdev;
    ...
}
```

/include/linux/fs.h



```
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int)
    const char * (*get_link) (struct dentry *, struct inode *, struct
delayed_call *);
    int (*permission) (struct user_namespace *, struct inode *, int);
    struct posix_acl * (*get_acl)(struct inode *, int, bool);

    int (*readlink) (struct dentry *, char __user *,int);

    int (*create) (struct user_namespace *, struct inode *,struct dentry *,
            umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct user_namespace *, struct inode *,struct dentry *,
          const char *);
    int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *,
           umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct user_namespace *, struct inode *,struct dentry *,
           umode_t,dev_t);
    int (*rename) (struct user_namespace *, struct inode *, struct dentry *,
          struct inode *, struct dentry *, unsigned int);
    int (*setattr) (struct user_namespace *, struct dentry *,
          struct iattr *);
    int (*getattr) (struct user_namespace *, const struct path *,
          struct kstat *, u32, unsigned int);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
           u64 len);
    int (*update_time)(struct inode *, struct timespec64 *, int);
    int (*atomic_open)(struct inode *, struct dentry *,
            struct file *, unsigned open_flag,
            umode_t create_mode);
    int (*tmpfile) (struct user_namespace *, struct inode *,
          struct dentry *, umode_t);
    int (*set_acl)(struct user_namespace *, struct inode *,
            struct posix_acl *, int);
    int (*fileattr_set)(struct user_namespace *mnt_userns,
             sruct dentry *dentry, struct fileattr *fa);
    int (*fileattr_get)(struct dentry *dentry, struct fileattr *fa);
} ____cacheline_aligned;
```

Virtual Filesystem (VFS)

- VFS Structures Details, *Cont'd*
  - *dentry*

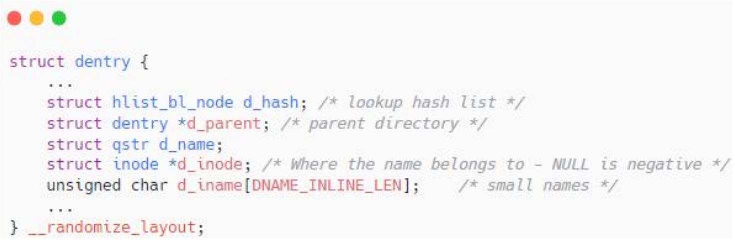
    Dentry State : there are 3 states for dentry object,
    1) **Used** : used, and points to a valid inode.
    2) **Unused** : not used , and points to a valid inode.
    3) **Negative** : dentry is not associated with a valid inode.

    Dentry Cache : As we know, when requesting operation related to path, VFS resolving each element into a dentry object.All these dentry objects, the kernel caches them in a cache called ***dcache*** to make use of them in the upcomming operations related to paths.

    dentry cache consists of 3 parts :
    1) Lists of "**used**" dentries linked off their associated inode via the ***i_dentry*** field of the inode object.

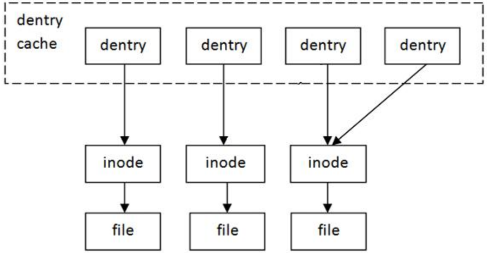
    2) A doubly linked "**least recently used**" list of unused and negative dentry objects.

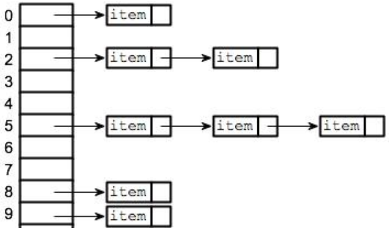
    3) A **hash table and hashing function** used to quickly resolve a given path into the associated dentry object by *d_lookup()*.

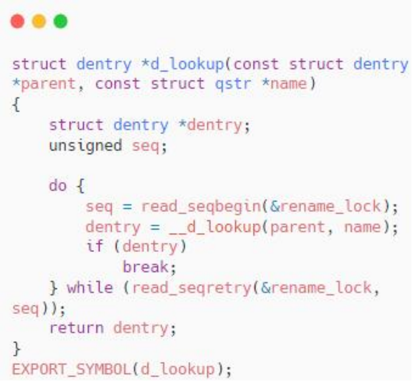


```
struct dentry {
    ...
    struct hlist_bl_node d_hash; /* lookup hash list */
    struct dentry *d_parent; /* parent directory */
    struct qstr d_name;
    struct inode *d_inode; /* Where the name belongs to - NULL is negative */
    unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */
    ...
} __randomize_layout;
```

/include/linux/dcache.h







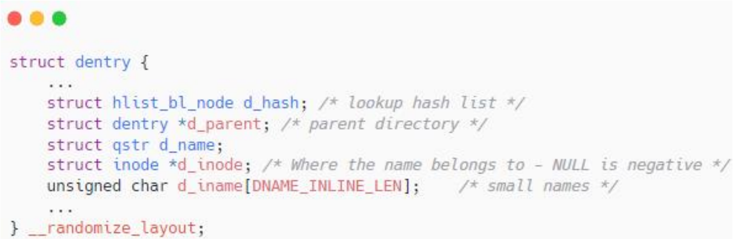
```
struct dentry *d_lookup(const struct dentry
*parent, const struct qstr *name)
{
    struct dentry *dentry;
    unsigned seq;

    do {
        seq = read_seqbegin(&rename_lock);
        dentry = __d_lookup(parent, name);
        if (dentry)
            break;
    } while (read_seqretry(&rename_lock,
seq));
    return dentry;
}
EXPORT_SYMBOL(d_lookup);
```

Virtual Filesystem (VFS)

- VFS Structures Details, *Cont'd*
  - ***dentry, Cont'd***
    The most important elements ***d_parent, d_name, d_inode, d_op***.
  - ***dentry_operations***
    d_revalidate() : Invoked when VFS needs to revalidate a dentry.Whenever a name lookup returns a dentry in the dcache, this is called.

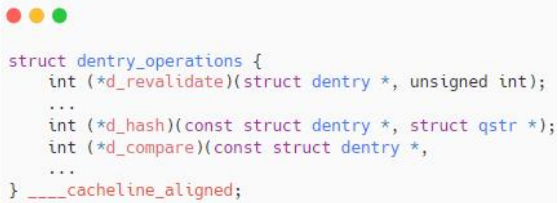
    d_compare() : Invoked to compare the filenames of two dentry instances. It compares a dentry name with a given name.

    d_hash() : Invoked when VFS adds a dentry to the hash table.



```
struct dentry {
    ...
    struct hlist_bl_node d_hash; /* lookup hash list */
    struct dentry *d_parent; /* parent directory */
    struct qstr d_name;
    struct inode *d_inode; /* Where the name belongs to - NULL is negative */
    unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */
    ...
} __randomize_layout;
```

/include/linux/dcache.h



```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    ...
    int (*d_hash)(const struct dentry *, struct qstr *);
    int (*d_compare)(const struct dentry *,
    ...
} ____cacheline_aligned;
```

Virtual Filesystem (VFS)

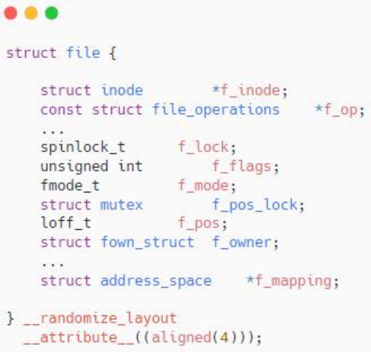
- VFS Structures Details, *Cont'd*
    - ***file, file_operations :***

      fsync() : Called by the fsync() system call to write all cached data for the file to disk.

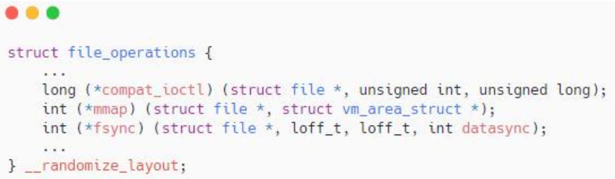
      compat_ioctl() : a portable variant of old ***ioctl()*** for use on 64-bit systems by 32-bit applications.

      what was ***ioctl()?***
      It sends a command and argument pair to a device. It is used when the file is an open device node.

      mmap() : Memory maps the given file onto the given address space and is called by the mmap() system call.



```
struct file {

    struct inode        *f_inode;
    const struct file_operations    *f_op;
    ...
    spinlock_t      f_lock;
    unsigned int        f_flags;
    fmode_t         f_mode;
    struct mutex        f_pos_lock;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    ...
    struct address_space    *f_mapping;

} __randomize_layout
  __attribute__((aligned(4)));
```

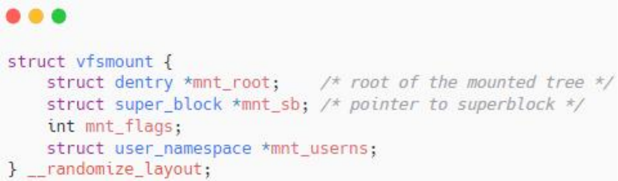


```
struct file_operations {
    ...
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    ...
} __randomize_layout;
```

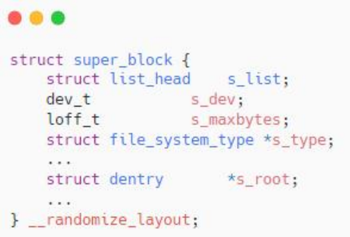
Virtual Filesystem (VFS)

- Data Structures Associated with Filesystems
  - ***file_system_type*** : describe a specific variant of a filesystem, such as ext3, ext4.
  - ***vfsmount*** : it’s the mounted filesystem descriptor.
    mnt_root : Points to the dentry of the root directory of this filesystem.

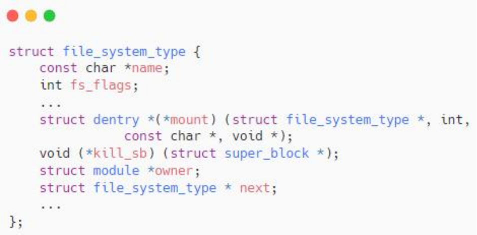
    mnt_sb : Points to the superblock object of this filesystem.



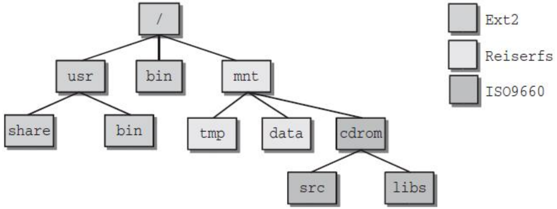
```
struct vfsmount {
    struct dentry *mnt_root;    /* root of the mounted tree */
    struct super_block *mnt_sb; /* pointer to superblock */
    int mnt_flags;
    struct user_namespace *mnt_userns;
} __randomize_layout;
```



```
struct super_block {
    struct list_head     s_list;
    dev_t                s_dev;
    loff_t               s_maxbytes;
    struct file_system_type *s_type;
    ...
    struct dentry        *s_root;
    ...
} __randomize_layout;
```



```
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;
    ...
};
```



Special Filesystems

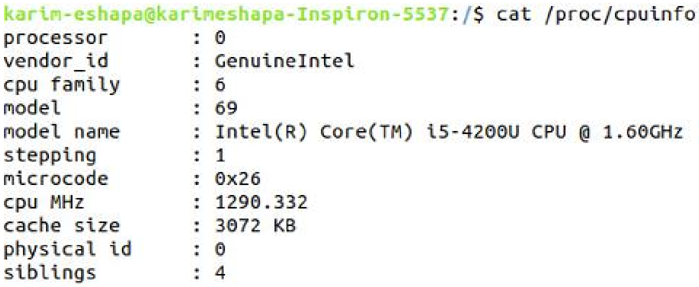
Special Filesystems

- Kernel implements special filesystems do not deal with persistent data, they do not consume disk blocks.

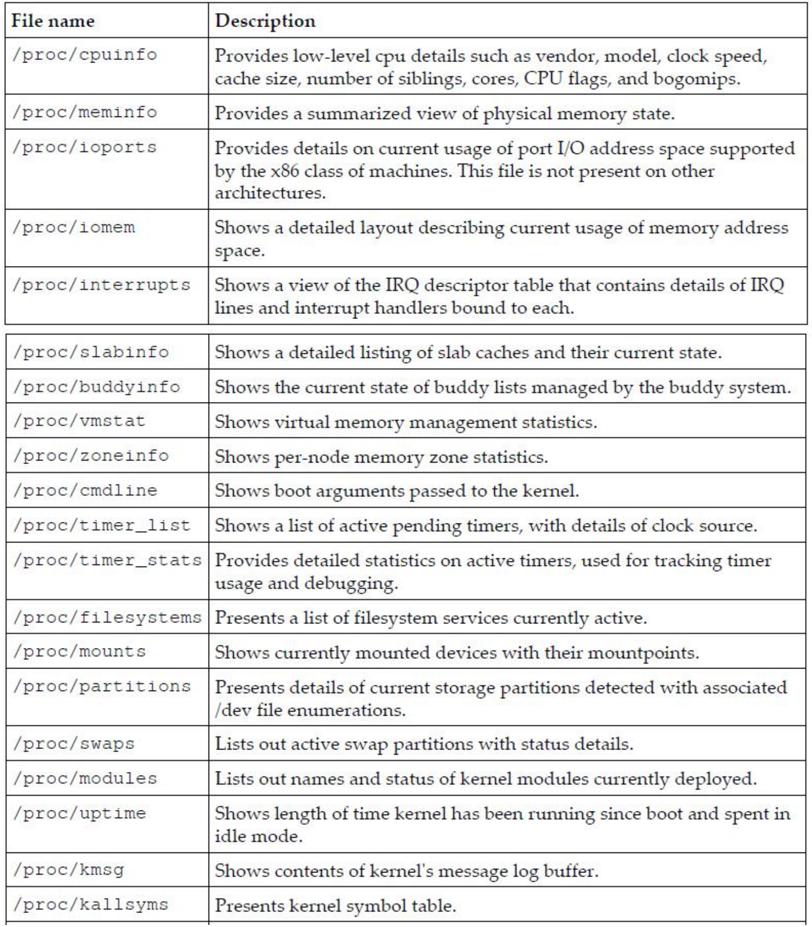
- These filesystems enables simplified application development, debugging, and easier error detection.

- Special FS
  - **Procfs** : Procfs is mounted to the ***/proc*** directory (mount point) of rootfs.
    Each file is an interface through which users can trigger associated operations.

    i.e. Read operation on a ***proc*** file invokes the associated read callback function bound to the file entry.



```
karim-eshapa@karimeshapa-Inspiron-5537:/$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 69
model name      : Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz
stepping        : 1
microcode       : 0x26
cpu MHz         : 1290.332
cache size      : 3072 KB
physical id     : 0
siblings        : 4
```



File name	Description
`/proc/cpuinfo`	Provides low-level cpu details such as vendor, model, clock speed, cache size, number of siblings, cores, CPU flags, and bogomips.
`/proc/meminfo`	Provides a summarized view of physical memory state.
`/proc/ioports`	Provides details on current usage of port I/O address space supported by the x86 class of machines. This file is not present on other architectures.
`/proc/iomem`	Shows a detailed layout describing current usage of memory address space.
`/proc/interrupts`	Shows a view of the IRQ descriptor table that contains details of IRQ lines and interrupt handlers bound to each.
`/proc/slabinfo`	Shows a detailed listing of slab caches and their current state.
`/proc/buddyinfo`	Shows the current state of buddy lists managed by the buddy system.
`/proc/vmstat`	Shows virtual memory management statistics.
`/proc/zoneinfo`	Shows per-node memory zone statistics.
`/proc/cmdline`	Shows boot arguments passed to the kernel.
`/proc/timer_list`	Shows a list of active pending timers, with details of clock source.
`/proc/timer_stats`	Provides detailed statistics on active timers, used for tracking timer usage and debugging.
`/proc/filesystems`	Presents a list of filesystem services currently active.
`/proc/mounts`	Shows currently mounted devices with their mountpoints.
`/proc/partitions`	Presents details of current storage partitions detected with associated /dev file enumerations.
`/proc/swaps`	Lists out active swap partitions with status details.
`/proc/modules`	Lists out names and status of kernel modules currently deployed.
`/proc/uptime`	Shows length of time kernel has been running since boot and spent in idle mode.
`/proc/kmsg`	Shows contents of kernel's message log buffer.
`/proc/kallsyms`	Presents kernel symbol table.

Special Filesystems

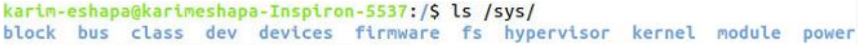
- Special FS, *Cont'd*
  - Sysfs : mounted to the ***/sys*** directory of the rootfs.
    - ***devices*** : present a unified list of devices currently enumerated and managed by respective driver.

    - ***bus*** : a listing of subdirs, each representing the physical bus type that has support registered in the kernel. Each bus type dir contains 2 subdirs :

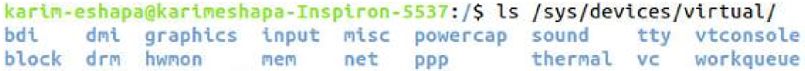
      1) devices : listing of devices currently discovered or bound to that bus type.
      Each file in the listing is a symbolic link to the device file in ***device's*** dir in the global device tree.

      2) drivers : contains dirs describing each device driver registered with the bus manager.

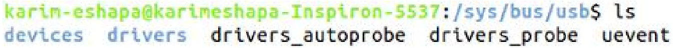
      Each driver directories lists attributes that show the current configuration of driver parameters, which can be modified, and symbolic links that point to the physical device directory that the driver is bound to.



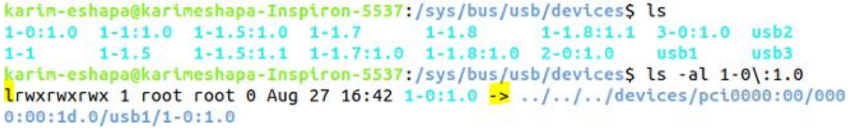
```
karim-eshapa@karimeshapa-Inspiron-5537:/$ ls /sys/
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module  power
```



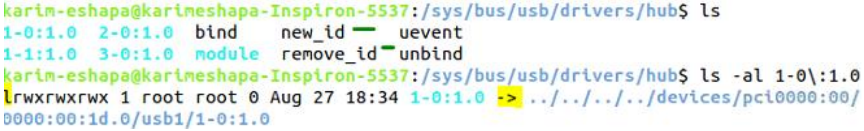
```
karim-eshapa@karimeshapa-Inspiron-5537:/$ ls /sys/devices/virtual/
bdi    dmi  graphics  input  misc  powercap  sound    tty  vtconsole
block  drm  hwmon     mem    net   ppp       thermal  vc   workqueue
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/bus/usb$ ls
devices  drivers  drivers_autoprobe  drivers_probe  uevent
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/bus/usb/devices$ ls
1-0:1.0  1-1:1.0  1-1.5:1.0  1-1.7      1-1.8      1-1.8:1.1  3-0:1.0  usb2
1-1      1-1.5    1-1.5:1.1  1-1.7:1.0  1-1.8:1.0  2-0:1.0    usb1     usb3
karim-eshapa@karimeshapa-Inspiron-5537:/sys/bus/usb/devices$ ls -al 1-0\:1.0
lrwxrwxrwx 1 root root 0 Aug 27 16:42 1-0:1.0 -> ../../../devices/pci0000:00/000
0:00:1d.0/usb1/1-0:1.0
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/bus/usb/drivers/hub$ ls
1-0:1.0  2-0:1.0  bind    new_id     uevent
1-1:1.0  3-0:1.0  module  remove_id  unbind
karim-eshapa@karimeshapa-Inspiron-5537:/sys/bus/usb/drivers/hub$ ls -al 1-0\:1.0
lrwxrwxrwx 1 root root 0 Aug 27 18:34 1-0:1.0 -> ../../../../devices/pci0000:00/
0000:00:1d.0/usb1/1-0:1.0
```

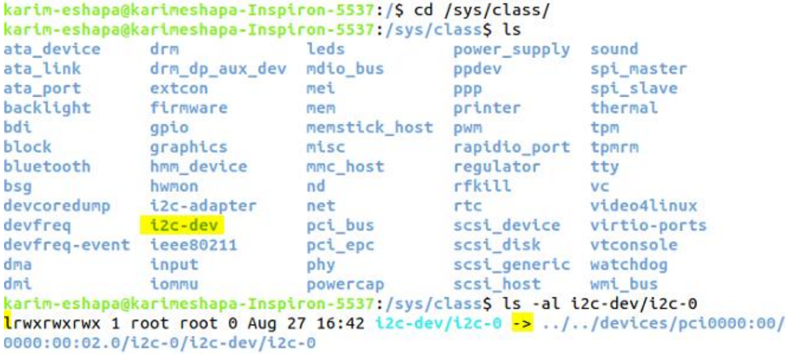
Special Filesystems

- Special FS, *Cont'd*
  - Sysfs, *Cont'd*

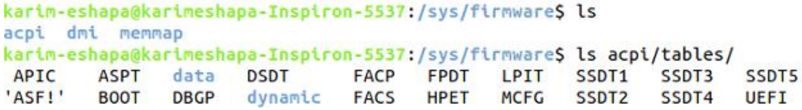
    - ***class*** : contains representations of device classes that are currently registered with the kernel.
    Each device class dir contains subdirs representing devices currently allocated and registered under this class.

    For most of the class device objects, their dirs contain symbolic links to the device dirs in the global ***devices*** hierarchy.

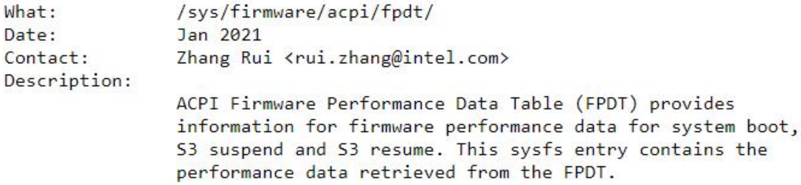
    - ***Firmware*** : contains interfaces for viewing and manipulating platform-specific firmware that is run during power on/reset, such as BIOS or UEFI on x86 and OpenFirmware for PPC platforms.



```
karim-eshapa@karimeshapa-Inspiron-5537:/$ cd /sys/class/
karim-eshapa@karimeshapa-Inspiron-5537:/sys/class$ ls
ata_device     drm              leds            power_supply   sound
ata_link       drm_dp_aux_dev   mdio_bus        ppdev          spi_master
ata_port       extcon           mei             ppp            spi_slave
backlight      firmware         mem             printer        thermal
bdi            gpio             memstick_host   pwm            tpm
block          graphics         misc            rapidio_port   tpmrm
bluetooth      hmm_device       mmc_host        regulator      tty
bsg            hwmon            nd              rfkill         vc
devcoredump    i2c-adapter      net             rtc            video4linux
devfreq        i2c-dev          pci_bus         scsi_device    virtio-ports
devfreq-event  ieee80211        pci_epc         scsi_disk      vtconsole
dma            input            phy             scsi_generic   watchdog
dmi            iommu            powercap        scsi_host      wmi_bus
karim-eshapa@karimeshapa-Inspiron-5537:/sys/class$ ls -al i2c-dev/i2c-0
lrwxrwxrwx 1 root root 0 Aug 27 16:42 i2c-dev/i2c-0 -> ../../devices/pci0000:00/
0000:00:02.0/i2c-0/i2c-dev/i2c-0
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/firmware$ ls
acpi  dmi  memmap
karim-eshapa@karimeshapa-Inspiron-5537:/sys/firmware$ ls acpi/tables/
 APIC    ASPT   data   DSDT       FACP   FPDT   LPIT   SSDT1   SSDT3   SSDT5
'ASF!'   BOOT   DBGP   dynamic    FACS   HPET   MCFG   SSDT2   SSDT4   UEFI
```



```
What:           /sys/firmware/acpi/fpdt/
Date:           Jan 2021
Contact:        Zhang Rui <rui.zhang@intel.com>
Description:
                ACPI Firmware Performance Data Table (FPDT) provides
                information for firmware performance data for system boot,
                S3 suspend and S3 resume. This sysfs entry contains the
                performance data retrieved from the FPDT.
```

Special Filesystems

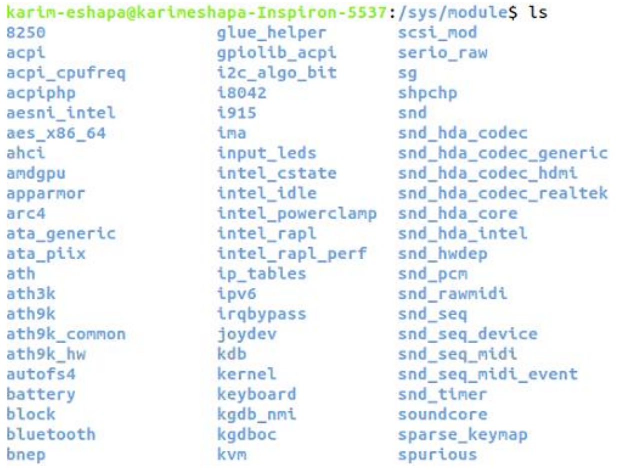
- Special FS, *Cont'd*
  - Sysfs, *Cont'd*

    - ***module*** : contains subdirs that represent each kernel module currently deployed. Each dir is enumerated with the name of the module it is representing. Each module dir contains information about a module such as ***refcount***, ***modparams***, and its core size.

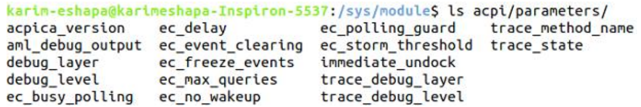
  - Debugfs : Unlike procfs and sysfs, which are implemented to present specific information through the virtual file interface, debugfs is a generic memory filesystem that allows kernel developers to export any arbitrary information that is useful for debugging.

    Generally mounted to the ***/sys/kernel/debug*** dir.

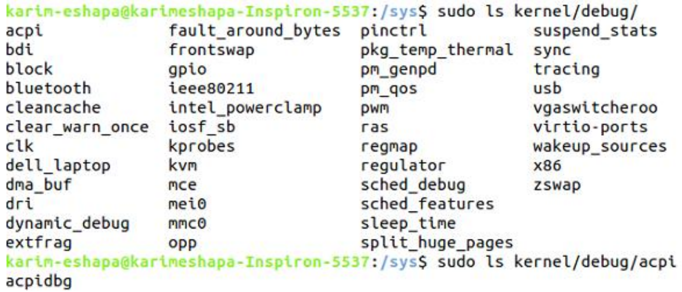
  - Many other special filesystems such as **pipefs**, **mqueue**, and **sockfs.**



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/module$ ls
8250                 glue_helper       scsi_mod
acpi                 gpiolib_acpi      serio_raw
acpi_cpufreq         i2c_algo_bit      sg
acpiphp              i8042             shpchp
aesni_intel          i915              snd
aes_x86_64           ima               snd_hda_codec
ahci                 input_leds        snd_hda_codec_generic
amdgpu               intel_cstate      snd_hda_codec_hdmi
apparmor             intel_idle        snd_hda_codec_realtek
arc4                 intel_powerclamp  snd_hda_core
ata_generic          intel_rapl        snd_hda_intel
ata_piix             intel_rapl_perf   snd_hwdep
ath                  ip_tables         snd_pcm
ath3k                ipv6              snd_rawmidi
ath9k                irqbypass         snd_seq
ath9k_common         joydev            snd_seq_device
ath9k_hw             kdb               snd_seq_midi
autofs4              kernel            snd_seq_midi_event
battery              keyboard          snd_timer
block                kgdb_nmi          soundcore
bluetooth            kgdboc            sparse_keymap
bnep                 kvm               spurious
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys/module$ ls acpi/parameters/
acpica_version    ec_delay           ec_polling_guard     trace_method_name
aml_debug_output  ec_event_clearing  ec_storm_threshold   trace_state
debug_layer       ec_freeze_events   immediate_undock
debug_level       ec_max_queries     trace_debug_layer
ec_busy_polling   ec_no_wakeup       trace_debug_level
```



```
karim-eshapa@karimeshapa-Inspiron-5537:/sys$ sudo ls kernel/debug/
acpi             fault_around_bytes  pinctrl           suspend_stats
bdi              frontswap           pkg_temp_thermal  sync
block            gpio                pm_genpd          tracing
bluetooth        ieee80211           pm_qos            usb
cleancache       intel_powerclamp    pwm               vgaswitcheroo
clear_warn_once  iosf_sb             ras               virtio-ports
clk              kprobes             regmap            wakeup_sources
dell_laptop      kvm                 regulator         x86
dma_buf          mce                 sched_debug       zswap
dri              mei0                sched_features
dynamic_debug    mmc0                sleep_time
extfrag          opp                 split_huge_pages
karim-eshapa@karimeshapa-Inspiron-5537:/sys$ sudo ls kernel/debug/acpi
acpidbg
```