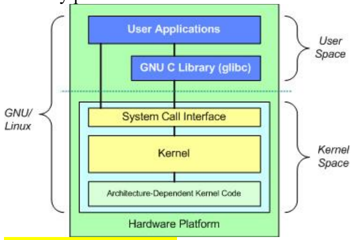
Linux

Not Unix, but implements the Unix API (POSIX and SUS). Under GNU General Public License (GPL).

OS kernal

Kernel-space refers to the elevated system state (full access to hardware) and its protected memory space. Applications execute a system call in kernel space, and the kernel is running in process context.The interrupt handlers run in an interrupt context, which is not associated with any process.



GNU C Library (glibc)

Library for system call interfaces, provides entries for switching from user mode to kernel mode.

Linux Kernel

system call interface, process management, memory management, virtual file system, network stack, device drivers, hardware dependent code.

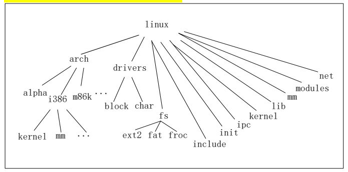
Linux Feature

Dynamic loading of modules. A task can be preempted when executing in the kernel. All threads are implemented as processes that share resources. Removed poorly designed Unix features (such as STREAMS). Free and practical, no market bullets. SMP support.

Linux Kernel Subsystems

System Call Interface: Functional call from user space to kernel space. Basic functions like "read" or "write". **Process Mgmt:** manages the CPU usage for processes. **Memory Mgmt:** Multiple processes share the main memory safely. **VFS:** hides the hardware complexity, provides unified interface for all devices.

Linux kernel Code Tree



arch: architecture dependent code, e.g. i386 is the subtree for Intel CPU and its compatible CPUs. **include:** most header files are stored here. Those hardware-independent headers are under include/linux, those for Intel CPU are under include/asm-i386, while those for scsi devices are under include/scsi. **init:** kernel boot & initialization code, including two files main.c and version.c. **mm:** memory mgmt independent on CPU architecture, e.g. paging allocation and deallocation, those for specific architectures are under arch/*/mm/, e.g. arch/i386/mm/fault.c. **kernel:** the main kernel code. Most linux kernel functions are implemented here. Sched.c contains the code for scheduler; architecture-dependent code is under arch/*/kernel. **drivers:** device drivers. /block for block device drivers. Device initialization code is in device_setup() in drivers/block/genhd.c. **lib:** library code for the kernel. **net:** network code in the kernel. **ipc:** IPC code. **fs:** All FS code and all sorts of file operations. Each FS is in a subdirectory. E.g. FAT and ext2. **scripts:** scripts for compiling the kernel.

special about Kernel Programming

No libc or standard headers: kernel source files cannot include outside headers, cannot use outside libs. **Uses GNU C**: inline functions, inline assembly, branch annotation such as 'asm' and 'unlikely'. **No memory protection**: No trap, simply a kernel error, not pageable. **Synchronization & concurrency**: preemptive kernel, supports SMP. Different code may preempt and access the same resource. Need spinlocks and semaphores.**No use of floating point:** user space FP instruction traps and enters FP mode. Kernel cannot trap itself, must save/restore FP registers. **Small, fixed-size stack**: 8KB on 32-bit architectures, and 16KB on 64-bit architectures.

Kernel Images

Linux kernel images are stored under /boot and named like vmlinuz-2.6.15.5: **Normal kernel image**: zImage（Image compressed with gzip）, no greater than 512k. **Big zipped image**: bzImage(big Image compressed with gzip)， includes most kernel functionalities.

Compiling Linux kernel

sudo apt-get install libncurses5-dev

Store linux-3.18.24.tar.xz in ~

tar -xvf linux-3.18.24.tar.xz

cd linux-3.18.24

cp /boot/config-`uname -r` .config

or cp /boot/config-<Tab> .config

make menuconfig //generate the .config file

make or make -j4

sudo make modules_install

sudo make install

sudo gedit /boot/grub/grub.cfg //Check

sudo reboot uname -r

Linux Boot

Real mode initialization (setup.S). **Protected mode initialization** (head.S::startup_32) Start_kernel (main.c). **User space init** (init process):1.Read /etc/inittab (system init config file); 2. Execute init scripts (/etc/rc.d/rc*.d/, …) per current run level).

Process Management in Linux

A thread in Linux is a special kind of process.

PCB

Managed by circular doubly linked list (**task list**). Priori to 2.6 kernel, **task_struct** was stored at the end of the kernel stack of each process. Now it is allocated via the slab allocator. A new structure **thread_info** is stored at the end of the kernel stack. 最多 512。

Find Process

list for each (list, &current->children) { task = list_entry(list, struct task_struct, sibling); //task now points to one of current's children} list_entry (task-stasks. next, struct task struct, tasks) //next task. Use .prev to find prev task.

#define **list_for_each(pos, head)** for (pos = (head)->next; pos != (head); pos = pos->next) #define **for_each_process(p)** for (p = &init_task ; (p = next_task(p)) != &init_task ;)

Process States

TASK_RUNNING: Runnable, either running or on a runqueue waiting to run. (running both in user space and kernel space). **TASK_INTERRUPTIBLE**: Sleeping (blocked), waiting for some condition to exist. When condition exists, set to TASK_RUNNING. If signal received it awakes prematurely also. **TASK_UNINTERRUPTIBLE**: same as above, except that it does NOT wake up when receiving a signal. The process must wait (e.g. holding a semaphore). **TASK_STOPPED**: Process is stopped by signal SIGSTOP，SIGTSTP，SIGTTIN, or SIGTTOU. Can be waken up by other signals. **TASK_TRACED**: Process is traced by another process, such as a debugger, via ptrace.

task_structure (include/linux/sched.h):

```
Line 701 struct task_struct {
    volatile long state;
/* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;
/* per process flags, defined below */...
    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;                ...
    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    unsigned long long sched_time;
/* sched_clock time spent running */
    int activated;
    unsigned long policy;……
}
```

1. 进程的状态和标志

volatile long state //进程的状态

unsigned long flags //进程的标志

2.进程的标识

int pid //进程标识号

unsigned short uid, gid //用户标识号，组标识号

unsigned short euid, egid //用户、组有效标识号

unsigned short suid, sgid //用户、组备份标识号

unsigned short fsuid, fsgid //用户、组标识号

3.进程的族亲关系

struct task_struct *p_opptr //指向祖先进程 PCB

struct task_struct *p_pptr //指向父进程 PCB

struct task_struct *p_cptr //指向子进程 PCB

struct task_struct *p_ysptr //指向弟进程 PCB

struct task_struct *p_osptr //指向兄进程 PCB

4. 进程间的链接信息

struct task_struct *next_task //指向下一个 PCB

struct task_struct *prev_task //指向上一个 PCB

struct task_struct *next_run //指向可运行队列的下一个 PCB

p->comm //comm 类型为 char[16],代表进程名

p->parent //指向父进程 task_struct 的地址

fork()

Copy-on-write (COW): delays/prevents the copying. Parent and child share a single copy. A page is duplicated only if it is written to. **Child runs first:** eliminates copy-on-write overhead if parent ran first and began writing to the address space.**The only overhead**: dup of parent's page table and the creation of PCB.

sys_clone(),sys_vfork(),sys_fork() 可以实现创建子进程，这三个系统调用最终都会调用 do_fork()。在 arch/i386/kernel/process.c 中。

do_fork() (kernel/fork.c)：

1) 调用 alloc_task_struct() 分配子进程 task_struct 空间。严格地讲，此时子进程还未生成。2) 把父进程 task_struct 的值全部赋给子进程 task_struct。3) 检查是否超过了资源限制，如果是，则结束并返回出错信息。更改一些统计量的信息。4) 修改子进程 task_struct 的某些成员的值使其正确反映子进程的状况，如进程状态被置成 TASK_UNINTERRUPTIBLE。5) 调用 get_pid( )函数为子进程得到一个 pid 号。6) 共享或复制父进程文件处理、信号处理及进程虚拟地址空间等资源。7) 调用 copy_thread( )初始化子进程的内核栈,内核栈保存了进程返回用户空间的上文。此处与平台相关，以 i386 为例，其中很重要的一点是寄存器 eax 值的位置被置 0,这个值就是执行系统调用后子进程的返回值。8) 将父进程的当前的时间配额 counter 分一半给子进程。9) 利用宏 SET_LINKS 将子进程插入所有进程都在其中的双向链表。调用 hash_pid（）,将子进程加入相应的 hash 队列。10) 调用 wake_up_process( ),将该子进程插入可运行队列。至此，子进程创建完毕，并在可运行队列中等待被调度运行。11) 如果 clone_flags 包含有 CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程空间。进程控制块中有一个信号量 vfork_sem 可以起到将进程挂起的作用。12) 返回子进程的 pid 值，该值就是系统调用后父进程的返回值。

Process Schedule

Linux scheduler was simple before 2.4 kernel. Constant-time scheduler, O(1) scheduler, was introduced in 2.5. Scales well for large server workloads, poorly for interactive applications. **Completely Fair Scheduler (CFS)** since 2.6.23. **CFS** calculates how long a process should run as a function of the total of runnable processes. CFS uses the nice value to weight the proportion of processor. CFS sets a target latency for the actual TS. Floor on target latency as minimum granularity for TS.

runqueue 结构（kernel/sched.c）

prio_array_t *active, *expired, arrays[2]

active 指向时间片没用完、当前可被调度的就绪进程，expired 指向时间片已用完的就绪进程。每一类队列用一个 struct prio_array 表示（优先级排序数组）一个任务的时间片用完后，它会被转移到"过期"的队列中。在该队列中，任务仍然是按照优先级排好序。当活动队列中的任务均执行完交换指针。

System Handle

User space application signals the kernel via a software interrupt (exception), and the system will switch to kernel mode and execute the exception handler (syscall handler). **int $0x80** (or sysenter instruction) Function system_call() implemented in **entry_64.S**. The syscall number is fed into the kernel via the **eax** register. Return value sent via **eax**. Parameters pass via registers ebx, ecx, edx,esi,edi. More arguments are passed via a pointer to user space.

应用程序执行系统调用

1、程序调用 libc 库的封装函数。

2、调用软中断 int 0x80 进入内核。

3、在内核中首先执行 system_call 函数（1. 把系统调用号和该异常处理程序用到的所有 CPU 寄存器保存到相应的栈中（SAVE_ALL）。2. 把当前进程 task_struct（thread_info）结构的地址存放在 ebx 中。3. 对用户态进程传递来的系统调用号进行有效性检查。若调用号大于或等于 NR_syscalls，系统调用处理程序终止。（sys_call_table）。若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄存器所在的单元，再跳到 ret_from_sys_call()），接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程。

4、执行该服务例程。

5、执行完毕后，转入 ret_from_sys_call 例程，从系统调用返回。

Project2: Binding a System Call

1. Add an entry to the end of the system call table (for each architecture).

2. For each supported arch, define the syscall number in <asm/unistd.h>.

3. Compile the syscall into the kernel image. (putting the code in kernel/, such as sys.c)

Linked List in LK

The Linux kernel approach is to embed a linked list node in the structure.

```
struct list_head {
        struct list_head *next
        struct list_head *prev;
};
```

Embed a list node into the fox structure:

```
struct fox {
        unsigned long    tail_length;
        unsigned long    weight;
        bool             is_fantastic;
        struct list_head list;
};
```

Defining a list

```
struct fox *red_fox;
red_fox = kmalloc(sizeof(*red_fox), GFP_KERNEL)
red_fox->tail_length = 40;
red_fox->weight = 6;
red_fox->is_fantastic = false;
INIT_LIST_HEAD(&red_fox->list);
```

Initialize a list head: `static LIST_HEAD(fox_list);`

```
list_add(&f->list, &fox_list);
list_del(&f->list);
```

Interrupts

Polling: periodically the kernel check the status of the hardware in the system and respond accordingly. **Interrupt:** hardware signals the kernel when attention is needed Interrupt handler/Interrupt service **routine**: The interrupt handler for a device is part of the device's driver – the kernel code that manages the device.

The Interrupt Handler is the **top half**. The top half is run immediately upon receipt of the interrupt (time-critical work). Work can be later performed until the **bottom half.**

Interrupt Handlers

Interrupt handlers are declared by

```
static irqreturn_t intr_handler(int irq, void *dev)
```

irq is the number of the interrupt line.

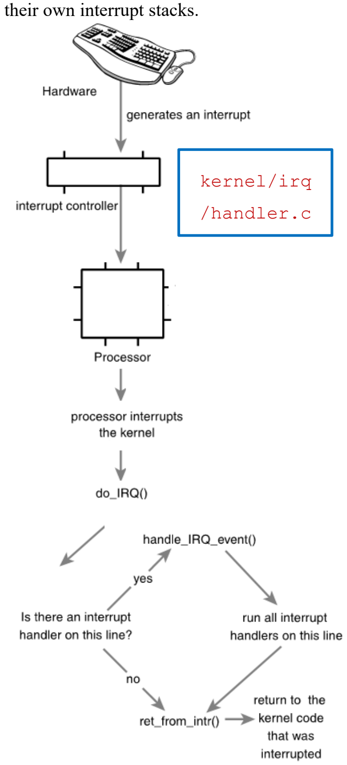
The return value can be IRQ_NONE or IRQ_HANDLED

If all the handlers on a given interrupt line return IRQ_NONE, the kernel will detect the problem. Interrupt handlers in Linux need not be reentrant! No nested interrupts.

Kernel Space Context

When executing in interrupt handler, the kernel is in **interrupt** context. When executing a system call, or running a kernel thread, it is in **process** context (macro current points to the associated task).

In interrupt context, the current macro is not relevant (no associated process), thus **it cannot sleep, as it would not reschedule**. As stack size is limited, Interrupt Handlers have their own interrupt stacks.



Interrupt Control

The Kernel provides interfaces for disabling the interrupt system for the current processor.

```
unsigned long flags;
local_irq_save(flags);
/* ... */
local_irq_restore(flags);
```

If work is time-sensitive: top half

If work related to h/w: top half

If work must not be interrupted by another interrupt: top half

Everything else: bottom half

Multiple mechanisms are available for implementing a **bottom half**. **Softirqs** are a set of statically defined bottom halves that can run simultaneously on any processor; even two of the same type can run concurrently. **Tasklets** are flexible, dynamically created bottom halves built on top of softirqs. Two different tasklets can run concurrently on different processors, but two of the same type of tasklet cannot run simultaneously. **Work queues** are a simple yet useful method of queuing work to later be performed in process context.

Physical Memory Management

An instance of the page structure is allocated for each frame.

```
struct page {
    unsigned long           flags;
    atomic_t                _count;
    atomic_t                _mapcount;
    unsigned long           private;
    struct address_space    *mapping;
    pgoff_t                 index;
    struct list_head        lru;
    void                    *virtual;
};
```

Allocating kernel memory

Due to h/w limitation, pages are not treated equally. They are divided into zones.

Allocating pages: alloc_page(), alloc_pages()

Returns struct page* pointer.

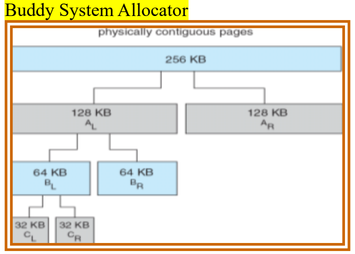
Use void *page_address(struct page *page) to return the logical address of the page.

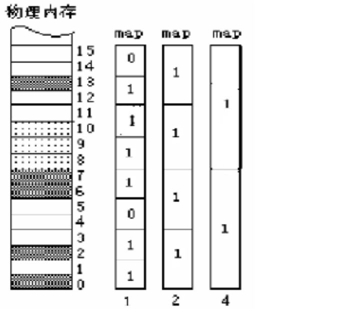
Freeing pages: free_page(), free_pages()

Defined in <linux/gfp.h>

kmalloc() is preferred compared with vmalloc() Because the latter not ensure physically contiguous memory being returned. 2. needs to set up page table entries; 3. Greater TLB thrashing. Use vmalloc() only if large regions are needed.

Buddy System Allocator





Each element of array free_area[] manages the free blocks at k-th order (free blocks of size 2k pages). All blocks at k-th order are linked via a double-linked list.

```
typedef struct free_area_struct {
        struct list_head        free_list;
        unsigned int            *map; /*  指向 bitmap */
} free_area_t;   static struct free_area_struct free_area[NR_MEM_LISTS];
```

Slab Allocator

The slab layer (allocator) acts as a generic data structure-caching layer to control free lists.

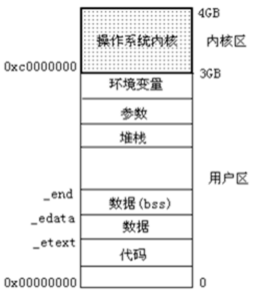
One cache (kmem_cache) **per object type**, e.g. one cache for task_structure, another for inode objects. kmalloc interface is built on top of slab layer. Caches are divided into slabs.

Slabs are composed of one or more physically contiguous frames, each contains a number of objects (data structures).

The Process Address Space

Processes can choose to share their address space with others: these are *threads*.

Process can address up to 4GB (32-bit), but it doesn't have permission to access all of it. The intervals of legal addresses are memory areas.



Kernel space cannot be swapped out.

The kernel space is mapped to the physical memory starting from 0x00000000.

The kernel image is stored at 0x00100000 (physical address starting at 1MB).

The Memory Descriptor

A process's address space is described in a mm_struct (defined in <linux/mm_types.h>). **mmap** and **mm_rb** are different data structures that contain the same thing. They form a threaded tree.

```
*mmap;                  /* list of memory areas */
mm_rb;                  /* red-black tree of VMAs */
*mmap_cache;            /* last used memory area */
free_area_cache;        /* 1st address space hole */
*pgd;                   /* page global directory */
mm_users;               /* address space users */
mm_count;               /* primary usage counter */
map_count;              /* number of memory areas */
mmap_sem;               /* memory area semaphore */
page_table_lock;        /* page table lock */
mmlist;                 /* list of all mm_structs */
start_code;             /* start address of code */
end_code;               /* final address of code */
start_data;             /* start address of data */
end_data;               /* final address of data */
start_brk;              /* start address of heap */
```

The memory descriptor associated with a task is stored in the mm field of the task_struct. Can be accessed via **current->mm** in the process context. The **copy_mm()** function copies the parent's **mm_struct** to its child during **fork()**. A process may choose to share its address space with its children by means of the CLONE_VM flag to clone(). We call the children threads!

Kernel Threads

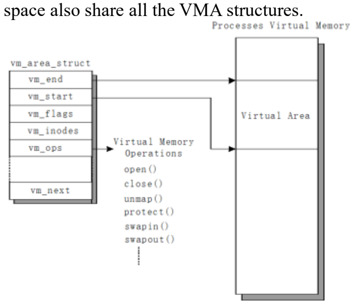
Kernel threads do not have a process address space (no memory descriptor, the mm field of kernle thread's process descriptor is NULL). This is the definition of a kernel thread – processes that **have no user context**. Kernel threads do not access any user-space memory. As kernel threads need to access memory, they use the mm_struct of whatever task ran **previously**, without wasting memory. This can be done, because the information in the address space of kernel memory is the **same for all**!

Virtual Memory Areas (VMA)

Describes a single area over a contiguous interval in a given address space. Each VMA has properties (permissions) and a set of operations. [vm_start, vm_end) is the contiguous range for the VMA.

```
vm_area_struct {
 struct mm_struct       *vm_mm;         /* associated mm_struct */
 unsigned long          vm_start;       /* VMA start, inclusive */
 unsigned long          vm_end;         /* VMA end , exclusive */
 struct vm_area_struct  *vm_next;       /* list of VMA's */
 pgprot_t               vm_page_prot;   /* access permissions */
 unsigned long          vm_flags;       /* flags */
 struct rb_node         vm_rb;          /* VMA's node in the tree
 union { /* links to address_space->i_mmap or i_mmap_nonlinear
       struct {
             struct list_head      list;
             void                  *parent;
             struct vm_area_struct *head;
       } vm_set;
       struct prio_tree_node prio_tree_node;
 } shared;
 struct list_head       anon_vma_node;     /* anon_vma entry */
 struct anon_vma        *anon_vma;         /* anonymous VMA objec
 vm_operations_struct   *vm_ops;           /* associated ops */
 unsigned long          vm_pgoff;          /* offset within file
 struct file            *vm_file;          /* mapped file, if any
 void                   *vm_private_data;  /* private data */
```

If two separate processes map the same file to their respective address space, each has a unique vm_area_struct to identify its unique memory area. Two threads sharing an address space also share all the VMA structures.



The operations are defined in <linux/mm.h>

```
vm_operations_struct {
 void (*open) (struct vm_area_struct *);
 void (*close) (struct vm_area_struct *);
 int (*fault) (struct vm_area_struct *, struct vm_fault *);
 int (*page_mkwrite) (struct vm_area_struct *vma, struct vm
 int (*access) (struct vm_area_struct *, unsigned long ,
                void *, int, int);
```

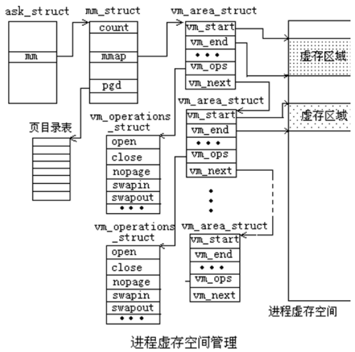
The fault() function is invoked by the page fault handler.

find_vma(): finds the VMA in which a given address resides.

find_vma_prev(): works as find_vma(), but also returns the last VMA before it.

find_vma_intersection(): returns the first VMA that overlaps a given address interval.

do_mmap() function is used to create a new linear address interval (not always new VMA).

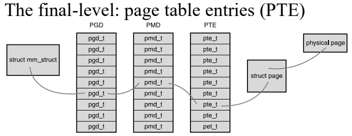


Virtual-to-physical address lookup

The top-level: page global directory (PGD)

The second-level:page middle directory (PMD)

The final-level: page table entries (PTE)



Linux File System

/ : root directory where the file system begins

/bin, /usr/bin: These two directories contain most of the programs for the system. The /bin directory has the essential programs that the system requires to operate, while /usr/bin contains applications for the system's users.

/boot: where the Linux kernel and boot loader filesarekept.The kernel is a file called vmlinuz.

/etc: contains the configuration files for the system. **/etc/fstab** contains a table of devices that get mounted when your system boots. This file defines your disk drives. **/etc/hosts** lists the network host names and IP addresses that are intrinsically known to the system. **/dev**: contains devices that are available to the system. **/etc/init.d**: contains the scripts that start various system services typically at boot time. **etc/passwd**: contains the essential information for each user. It is here that users are defined. **/home**: /home is where users keep their personal work. In general, this is the only place users are allowed to write files. **/lib:**The shared libraries (similar to DLLs in that other operating system) are kept here. **/media:** used for mount points. **/mnt**: provides a convenient place for mounting these temporary devices. **/proc:** doesn't contain files. This directory does not really exist at all. It's virtual. **/root**: This is the superuser's home directory. **/usr**: The /usr directory contains a variety of things that support user applications. **/var:** directory contains files that change as the system is running. This includes: **/var/log** Directory that contains log files. These are updated as the system runs. **/var/spool:** This directory is used to hold files that are queued for some process, such as mail messages and print jobs.

Linux FCB

Linux filesystems (ext2, ext3, ext4) separate storage for filename & FCB info. An FCB is stored as an inode. Each inode has a unique ID. Directory contains filename and inode info. Directory entry contains the filename and the inode ID.

Ext2

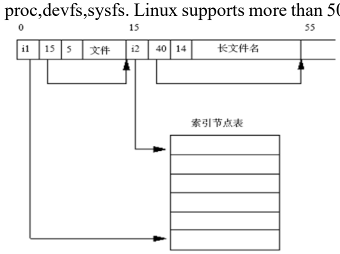
```
#define EXT2_NAME_LEN 255
struct ext2_dir_entry_2 {
    __u32   inode;      /* Inode number */
    __u16   rec_len;/* Directory entry length */
    __u8    name_len; /* Name length */
    __u8    file_type;
    char    name[EXT2_NAME_LEN];
    /*File name */
};
```

Special Files

Device I/O is implemented as file operations. Devices can be *character-based* or *block-based.* **FIFO 管道文件**：用于在进程间传递数据。Linux 对管道的操作与文件操作相同，它把管道做为文件进行处理。**Symbolic Links** 符号链接文件，它提供了共享文件的一种方法。**Sockets.**

File System Types

Disk based FS: ext2/ext3/ext4 (standard)、VFAT、NTFS. **Network FS**: NFS. **Special FS:** proc,devfs,sysfs. Linux supports more than 50.



Linux drivers

IDE硬盘机	/dev/hd[a-d]
SCSI/SATA/U盘硬盘机	/dev/sd[a-p]
U盘快闪碟	/dev/sd[a-p](与SATA相同)
软盘机	/dev/fd[0-1]
打印机	25针: /dev/lp[0-2] U盘: /dev/usb/lp[0-15]
鼠标	U盘: /dev/usb/mouse[0-15] PS2: /dev/psaux
当前 CDROM/DVDROM	/dev/cdrom
当前的鼠标	/dev/mouse
磁带机	IDE: /dev/ht0 SCSI: /dev/st0

/proc

procinfo 命令显示大量的系统信息

/proc/sys 目录是一个特殊目录，支持直接使用文件系统的操作，可以更改一些系统配置

/proc/self 是当前进程目录的符号链接。

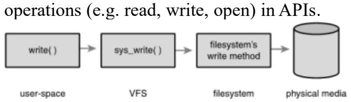
/proc/meminfo 当前内存使用信息

/proc/cpuinfo CPU 的详细信息

/proc/mounts 系统当前挂在的文件系统

Virtual File System

VFS (Virtual File Switch/System) supports Common File Interface. Interoperate between various file systems. New filesystems and new storage media can work without program rewritten/recompiled. VFS abstracts file operations (e.g. read, write, open) in APIs.

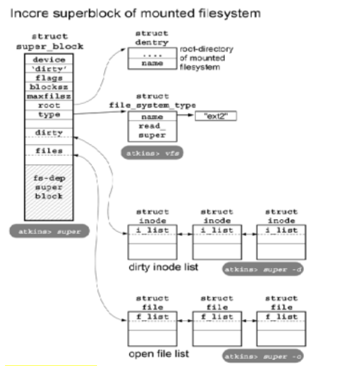


Four primary object types

superblock object (<linux/fs.h>)

一个超级块对应一个文件系统。**s_list:** 指向超级块链表中前一个和后一个超级块的指针 **s_dev:** 超级块所在的设备 **s_blocksize, s_blocksize_bits:** 指定了磁盘文件系统的块的大小 **s_type:** 指向文件系统类型的指针 **s_dirty:** 标记是否被修改 **s_maxbytes:** 文件大小上限 **s_o:** 指向超级块操作的指针 **_root:** 指向目录的 dentry 项

Incore superblock



inode object (<include/linux/fs.h>)

inode **包含文件的元信息**，具体来说有以下内容：文件的字节数，文件拥有者的 User ID，文件的 Group ID，文件的读、写、执行权限，文件的时间戳（ctime 指 inode 上一次变动的时间，mtime 指文件内容上一次变动的时间，atime 指文件上一次打开的时间），链接数（多少文件名指向这个 inode）文件数据 block 的位置。注意里面没有**文件名**。

inode 有两种，一种是 VFS 的 inode，一种是具体文件系统的 inode。前者在内存（动态）中，后者在磁盘（静态）中。

i_dev: 设备号 **i_ino:** 唯一编号 **i_mode:** 文件的类型和访问权限 **i_nlink:**与该节点建立链接的文件数(硬链接数) **i_uid:** 文件拥有者标号 **i_gid:**文件所在组标号

VFS 的 inode 组成一个双向链表，全局变量 **first_inode** 指向链表的表头。

dentry object (<include/linux/dcache.h>)

用来描述文件的逻辑属性，只存在于内存中。每个 dentry 代表路径中的一个特定部分。一个有效的 dentry 结构必定有一个 inode 结，但是一个 inodes 可以对应多个 dentry。

d_count: 引用计数 **d_inode:** 与该目录项关联的 inode **d_parent:** 父目录的目录项 **d_hash:** 目录项形成的哈希表 **d_subdirs:** 本目录所有孩子目录链表头 **d_child:** 该目录在父目录下的下一个兄弟。

d_op: 操作目录项的函数

file object (<include/linux/fs.h>)

文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。该对象由相应的 open() 系统调用创建，由 close() 系统调用销毁。

f_list: file 结构链表 **f_dentry:** 指向与文件对象关联的 dentr 对象。