Linux

Linux进程管理

1. 现代操作系统允许一个进程有多个执行流，即在相同的地址空间中可执行多个指令序列
2. 每个执行流用一个线程表示，一个进程可以有多个线程
3. Linux使用轻量级进程实现对多线程应用程序的支持，一个轻量级进程就是一个线程

Linux进程的组成

1. 在用户态运行时，进程映像包含有代码段、数据段、堆、用户栈
2. 进程在核心态运行时，访问内核的代码段和数据段，并使用各自的核心栈
3. 进程描述符（task_struct）：描述进程的数据结构
4. 主要的数据元素
    1. state / exit_state :进程的7种状态
    2. thread_info : 当前进程的基本信息
    3. mm_struct : 指向当前进程的有用的虚拟内存描述符(红黑树的指示地址)
    4. thread_struct : 保存进程硬件上下文
    5. files_struct : 指向该进程打开文件信息
    6. signal_struct : 所接收的信号
    7. dentry : 指向目录结构的指针
    8. list_head : list_head类型的数据结构代表Linux下的一些进程链表，很多条链表
5. 注意要点
    - Linux将每个进程的核心栈和基本信息**thread_info**结构体存放在两个连续的页框中(8KB)
    - PCB中的thread_info指针指向该thread_info结构体，thread_info结构体中的task指针指向PCB
    - esp寄存器存放的是核心栈的栈顶指针，内核很容易从esp寄存器的值获得正在CPU上运行的进程的thread_info结构的地址。进而获得进程描述符的地址。
    - 进程刚从用户态切换到核心态时，其核心栈为空，只要将栈顶指针减去**8k**，就能得到thread_info结构的地址。
6. 进程的状态
    - 可运行状态：进程正在或准备在CPU上运行的状态
    - 可中断的等待状态：进程睡眠等待系统资源可用或收到一个信号后，进程被唤醒
    - 不可中断的等待状态：进程睡眠等待一个不可被中断的事件的发生。如进程等待设备驱动程序探测设备的状态
    - 暂停状态
    - 跟踪状态 : 进程被另一个进程跟踪(debugger)
    - 僵死状态 : 进程中止执行，等待父进程善后处理
    - 死亡状态 : 父进程删除终止进程

Linux进程链表

1. 传统进程链表
    - 所有进程链表：
        - 利用list_head tasks字段的prev / next指针将操作系统中的所有的PCB连接起来
        - 构成双向链表
        - 链表头是0号进程 (idle进程)
    - 可运行进程链表：
        - 利用list_head run_list字段将所有处于可运行状态的进程连接起来，构建可运行进程队列
        - 按照优先级(0-139)构建140个可运行进程队列
        - 目的 : 提高调度程序执行的速度
        - 每个CPU一个
    - 子进程链表

        - 利用list_head children字段将该进程的所有子进程连接在一起，构成子进程列表
    - 兄弟进程链表
        - 利用list_head sibling字段将具有兄弟关系的进程连接在一起
    - 等待进程链表
        - 进程因为等待某些事件处于等待状态，放弃了CPU的使用权
            - 互斥等待访问临界资源的进程,每次只释放一个进程
            - 非互斥等待的进程，所有进程都被唤醒
2. 哈希链表
    1. 引入原因
        - 使用进程的PID在进程链表中检索虽然可行但是效率极低，为了加速进程的检索，内核设置了**4**类哈希表
    2. 内核初始化期间会给4个哈希表分配存储空间(pid_hash数组4大小),数组的容量大小取决于RAM
        - PID : 进程的PID
        - TGID : 线程组(一个进程的多个线程)领头的进程的PID
        - PGID : 进程组领头的进程的PID
        - SID : 会话领头的PID
    3. 构造哈希表时，总会发生碰撞，使用双向链表来组织冲突的PID(pid_chain)
    4. 一个线程组的所有的线程PCB的tgid属性相同，将一个进程组的进程组织在一个链表中
        - 目的 :

          如果不适用线程组链表的话，我们如果想要删除一个进程，只能获得同一个哈希值的多个进程的线程，不好处理删除操作(将进程的线程组织起来，方便处理)

```
* 使用PCB的pids字段组织
  * nr : 线程组号
  * pid_chain : 散列到同一个哈希表项的组织指针
  * pid_list : 组织同意线程组号的链表指针
```

Linux进程控制

1. 创建进程的函数fork()、clone()、vfork()
    1. clone()函数

       实现对多线程应用程序的支持。共享进程在内核的很多数据结构，如页表、打开文件表等等
    2. fork()函数
        - 创建进程的系统调用
        - 子进程采用写时复制技术和父进程共享系统资源 : 仅当父或子要写一个页时，才为其复制一个私有的页的副本
        - 父子进程随机
        - 之后启用exec函数可以执行新的进程
    3. vfork()
        - 创建成功后，子进程挂起父进程，子进程先运行
2. 0,1号进程
    - 0号是一个内核线程。使用静态分配的数据结构
    - 0号进程是所有进程的祖先进程
    - 每个CPU都有一个0号进程
    - 1号进程是由0号进程创建的内核线程init : 负责完成内核的初始化工作，和0号进程共享内核的数据结构
    - 在系统关闭之前，init进程一直存在，它负责创建和监控在操作系统外层执行的所有用户态进程
3. 撤销进程
    - exit()系统调用只终止某一个线程
    - exit_group()系统调用能终止整个线程组(在tgid哈希表上删除)

- 父进程先结束的子进程会成为孤儿进程，系统会强迫所有的孤儿进程成为init进程的子进程。不然会一直占据着RAM
- Init进程在用wait()类系统调用检查并终止子进程时，就会撤消所有僵死的子进程

Linux进程切换

1. 进程切换 : 暂停在CPU上的运行进程并恢复某一个进程的运行
2. 进程切换只发生在核心态。在发生进程切换之前，用户态进程使用的所有寄存器值都已被保存在进程的核心栈中
3. 之后大部分寄存器值存放在进程描述符的 thread_struct字段里，一小部分仍在核心栈中
4. 步骤
    1. 切换页目录表以安装一个新的地址空间
    2. 切换核心栈和硬件上下文。由schedule()函数完成进程切换

Linux进程调度

1. Linux2.6系统采用可抢先式的动态优先级调度方式 ：无论进程处于用户态还是核心态(防止内核低优先级组织高优先级进程运行)运行，都可能被抢占CPU
2. 调度方式
    1. 先进先出的实时进程
    2. 时间片轮转的实时进程
    3. 普通的分时进程
3. 实时进程的基本优先数为1～99，而分时进程和批处理进程的基本优先数为100～139
4. 优先数越小优先级越高
5. 调度涉及的数据结构
    1. 每个处理机都有自己的可运行队列，存放在一维数组runqueues中，每个CPU是数组中的一项(list_head run_list组织链表队列)
    2. runqueue结构

       为了防止低优先级进程被饿死,每个进程链表均存在140个优先级和对应的队列位图

        1. 活动进程链表：没有用完自己的时间片，允许运行
        2. 过期进程链表：已经用完了自己的时间片，禁止运行，直到所有的活动进程过期为止

       每个链表都存在的公有数据结构

        - 链表中包含的进程数目
        - 优先级队列位图(指示140个队列的进程个数情况)
        - 可运行进程集合(140个优先级列表)
    3. 为了平衡每个CPU之间的负载，内核会将可运行进程从一个运行队列迁移到另一个运行队列中

Linux内核同步

1. UNIX内核的各个组成部分并不是严格按照顺序依次执行的，而是采用交错方式执行的，以响应来自运行进程的请求和来自外部设备的中断请求
2. 互斥访问内核共享资源
3. 内核的同步技术
    - 每CPU变量 : 内核的变量，禁止抢占使用
    - 原子操作
    - 优化和内存屏障
    - 自旋锁
    - 读—拷贝—更新
    - 信号量
    - 禁止本地中断
    - 禁止和激活可延迟函数

Linux存储器管理

进程地址空间的管理

- 32位机，每个进程的地址空间为4GB
- 进程的私有地址空间是前3G，进程的公有地址空间是后1G的内核虚空间（公有空间是操作系统都可以访问的吗）
    - 私有空间：用户态或者核心态都可以访问
    - 公有空间：核心态可以访问
    - 内核1GB虚空间中的前896MB用来映射物理内存的前896MB
        - 内存的物理地址等于内核的虚地址减去3G(系统空间没有页表)
    - 后128MB用来实现超过896MB的物理内存的映射

虚拟内存区域

1. 对于进程的地址空间，是一些为程序的程序、数据、用户栈等分配保留的虚空间
2. 需用一组虚拟内存区域描述符vm_area_struct来描述进程地址空间的使用情况(类似VAD)
3. 主要的数据结构
    - struct mm_struct * vm_mm;指向拥有这个内存区域的进程的内存描述符
        - unsigned long vm_start; */起始地址/*
    - unsigned long vm_end; */结束地址/*
    - struct vm_area_struct *vm_next; /*单链表*/*
    - struct rb_node vm_rb; */红-黑树/*
    - struct file * vm_file; 映射磁盘上的文件到进程的私有区域
4. 组织方式

   Linux系统对进程已分配的虚拟内存区域采用两种数据结构管理
    - 单向链
        - 将进程的每个虚拟内存区域按照地址递增顺序链接在一起
        - 默认最多只有65536个
        - 适合内存中的虚拟内存区域数目较少的时候管理
    - 红黑树
        - 适合大量的内存区域的管理
        - 排好序的平衡二叉树
        - 管理虚拟内存区域
        - 规则
            - 树中的每个节点或为红或为黑
            - 树的根节点必须为黑
            - 红节点的孩子必须为黑
            - 从一个节点到后代诸叶子节点的每条路经，都包含相同数量的黑节点，在统计黑节点个数时，空指针也算作黑节点
        - 具有n个节点的红黑树，其高度至多为2*log(n+1)
    - Linux中的使用方式
        - 即使用单链表，也是用红黑树
        - 红黑树用来快速确定还有指定地址的虚拟内存区域
        - 单链表用来快速的扫描整个虚拟内存区域集合

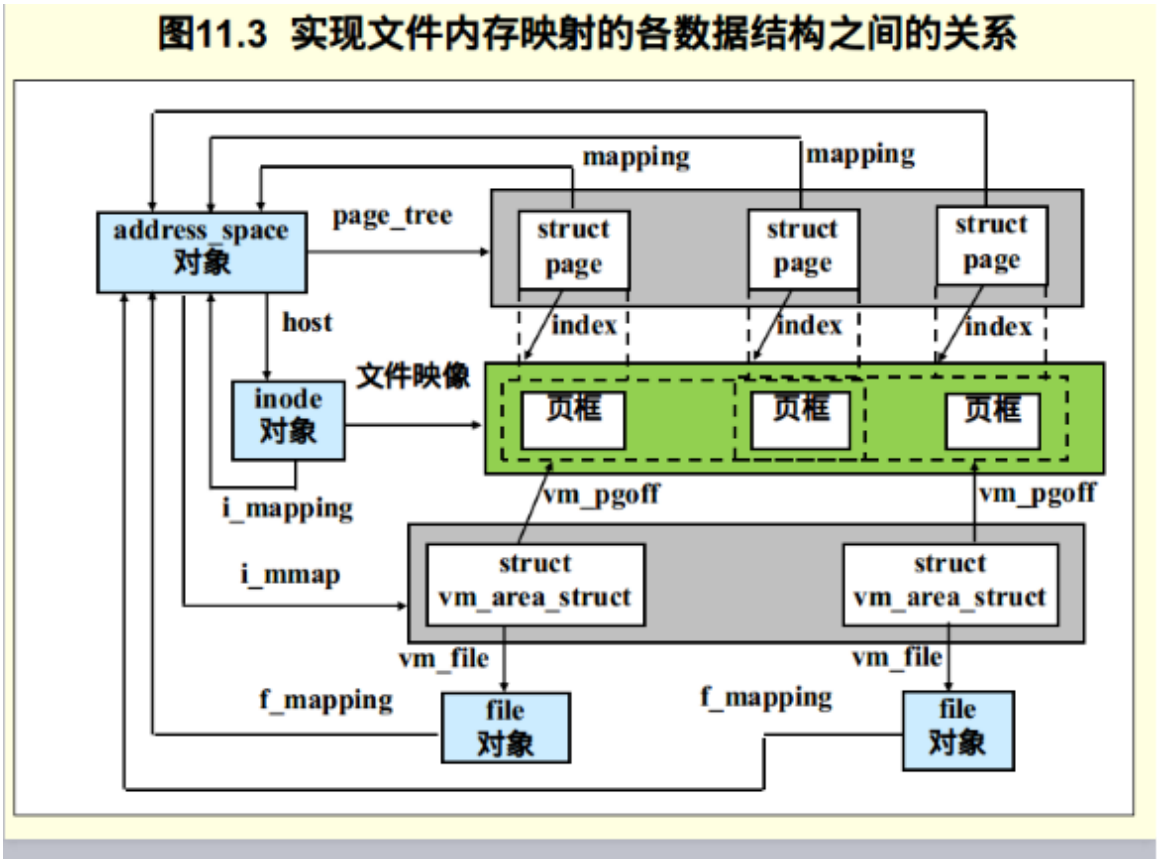
虚拟内存

1. 管理内存的地址空间
2. 虚存描述符mm_struct(PCB中的属性项)
3. 主要的属性

1. struct vm_area_struct *mmap 指向虚拟内存区域的链表的表头
2. struct rb_root mm_rb; *|指向虚拟内存区域构成的红-黑树的根|*
3. pgd_t *pgd;* /指向页目录表*/
4. 起始地址
5. 页表长度
6. atomic_t mm_users; *|次使用计数器|*
    - mm_users记录共享mm_struct的线程数
    - 记录进程的线程数目
7. atomic_t mm_count; *|主使用计数器|* ？？？？？？？决定是否释放？？？？？？？
    - 进程的所有线程在主使用计数器中只占一个单位
    - 0 : 不在有用户使用该虚拟内存，可以释放进程的空间
    - 若把mm_struct暂时借给一个内核线程(不是本进程的线程)使用，则mm_count值增1。
8. unsigned long start_code, end_code; *|可执行代码所占用的地址区间|*

物理内存的管理

1. 页框0由BIOS使用，存放加电自检期间检查到的系统硬件配置
2. 从0x000a0000到0x000fffff的物理地址通常留给BIOS例程
3. Linux跳过RAM的第一个MB的空间,1MB的大小页框操作系统不能使用
4. 页框大小4KB
5. 所有页框描述符struct page存放在mem_map数组(类似页框数据库)中
6. 页框描述符的主要属性
    - unsigned long flags
        - 页框状态标识
        - I/O错误 / 访问过 / 页修改 / 在slab中 / 空闲 / 活动页框链表中 / 非活动页框链表中 / …
    - atomic_t _count 页框引用计数
    - tomic_t _mapcount 页框对应的页表项索引
    - unsigned long private 空闲由伙伴系统使用来管理空闲页框
    - struct address_space *mapping



- 页框被插入页高速缓存的时候使用(组织起来属于同一个文件的页框)
- 页高速缓存的核心数据结构
- 文件的读写操作依赖于页高速缓存

- struct list_head lru　　被链入页活动页链和非活动页链?????????

7. 内存的3个管理区
    1. 物理内存被划分成几个内存节点
    2. 对于不同的节点内的页面访问的速度不一致
    3. Linux对每一个内存节点划分出了3个管理区
        - ZONE_DMA：包含低于16MB的常规内存页框。用于对老式的基于ISA设备的DMA支持
        - ZONE_NORMAL：包含高于16MB且低于896MB的常规内存页框
        - ZONE_HIGHMEM：包含从896MB开始的高端物理页框。内核不能直接访问这部分页框。在64位体系结构上，该区总是空的
    4. 管理区的数据结构
        - unsigned long free_pages;管理区空闲页框数
        - struct free_area free_area[11]; *伙伴系统中的11个空闲页框链表|*
        - struct list_head active_list; *活动页框链表，存放最近正被访问的页框|*
        - struct list_head inactive_list; *非活动页框链表，存放最近未被访问的页框|*
    5. 对管理区的内存分配操作使用页框分配器
8. 页框分配器
    1. 负责处理对连续物理页框的分配请求
    2. 每CPU页框高速缓存
        - 为CPU预先分配一些页框
        - 满足本地的CPU对单个页框的请求
    3. 伙伴系统
        - 满足本地的CPU对连续页框的请求，解决外碎片问题（夹杂在已分配页框中的小的空闲页框）
        - 伙伴算法把空闲页框组织成11个链表，分别链有大小为1，2，4，8，16，32，64，128，256，512和1024个连续空闲页框

- 假设要请求一个具有8个连续页框的块，该算法先在8个连续页框块的链表中检查是否有一个空闲块。如果没有，就在16个连续页框块的链表中找。如果找到，就把这16个连续页框分成两等份，一份用来满足请求，另一份插入到具有8个连续页框块的链表中。如果在16个连续页框块的链表中没有找到空闲块，就在更大的块链表中查找。直到找到为止
- 每个块的第一个页框的物理地址是块大小的整数倍

slab管理

1. 大块内存的分配请求可以通过伙伴算法实现(页框为单位)
2. 但是小内存的分配不适合(字节为单位)
3. slab分配器将小内存看做是对象
4. 分配过程
    1. slab初始化 : 从页框分配器获得几组连续空闲页框
    2. slab分配器为不同类型的对象生成不同的高速缓存，每个高速缓存存储相同类型的对象
    3. 高速缓存由一连串的**slab**构成，每个slab包含了若干个同类型的对象
5. 组织层次

    空闲页框 > 高速缓存 > slab > 同类型对象

地址转换

1. 32位处理机普遍采用二级页表模式，为每个进程分配一个页目录表，页表一直推迟到访问页时才建立，以节约内存
2. 虚地址分成3个域：页目录索引（前10位）、页表索引（中10位）和页内偏移（后12位）
3. Linux系统的页目录项和页表项的数据结构相同
4. 页表项常用属性
    - present : 为1，表示页（或页表）在内存；为0，则不在内存。
    - 页框物理地址 : 20位(4KB站走12位)
    - Accessed : 页框访问标志，为1表示访问过
    - dirty : 每当对一个页框进行写操作时就设置这个标志

请求调页

1. 请求调页增加了系统中的空闲页框平均数

    对页框的分配推迟到进程要访问的页不在RAM引起缺页中断,将页调入内存才分配????????????

2. 页面置换策略是LFU　－　最少频率使用页面淘汰算法
3. 不在内存的页的情况(理解)
    1. 该页从未被进程访问过，且没有相应的内存映射
    2. 该页已被进程访问过，但其内容被临时保存到磁盘交换区上
    3. 该页在非活动页框链表中
    4. 该页正在由其它进程进行I/O传输过程中

盘交换区空间管理

1. 每个盘交换区都由一组4KB的页槽组成
2. 盘交换区的第一个页槽用来存放该交换区的有关信息，有相应的描述符
3. 存放在磁盘分区中的交换区只有一个子区，存放在普通文件中的交换区可能有多个子区，原因是磁盘上的文件不要求连续存放
4. 内核尽力把换出的页存放在相邻的页槽中，减少访问交换区时磁盘的寻道时间

Linux文件系统

Ext2文件卷的布局（文件系统）

？？？？盘块　＝　数据块　＝　物理页框？？？？

1. 若干个磁盘块构成，Ext2把磁盘块分为组，每组包含存放在相邻磁道的数据块和索引节点。块组的大小相等并顺序安排
2. 块组
    - 超级块 : 存放整个文件卷的资源管理信息
        - 索引节点总数
        - 盘块的总数
        - 盘块大小
    - 块组描述符 : 一块，每个块组都有一个块组描述符，用来记录块组管理的重要信息
    - 数据块位图 : 一块
    - 文件的索引节点位图 : 一个单独管理索引节点的使用情况
    - 索引节点区 : 文件的管理控制信息
    - 文件数据区 : 文件数据存放
3. Ext2用块组描述符来描述这些块组本身的结构信息，同时将超级块和所有块组描述符重复存储于每个块组中
4. Ext2通过位图来管理每个块组中的磁盘块和索引节点。盘块位图，索引节点位图
5. 只有块组0中所包含的超级块和组描述符才由内核使用，而其余的超级块和组描述符保持不变，事实上，内核甚至不考虑它们
6. 文件目录和索引节点结构
    - 优点
        - 索引节点 : 系统只保留一个索引节点，就可实现多条路径共享文件，减少信息冗余
        - 简单目录可以加快检索速度，存放的也多了

- 文件目录项
    - 简单目录项 : 包含了文件名和索引节点号等，可以提高文件目录的检索速度
    - 至少12个字节
    - 最后的文件名是变长的（4-255B，是4的整数倍不足用NULL填充），实际的长度是name_len
- 索引节点(文件描述符)
    - 存放在连续的盘块中，存放在索引节点区
    - 一个索引节点的大小是128B(一个1KB磁盘块可以存放8个索引节点)
    - 主要属性
        - 文件类型和访问权限
        - 硬链接计数
        - 文件访问控制表ACL表
        - i_block[15]文件索引表(4T的文件) — 文件的索引存放
            1. 是一个有15个元素的数组，每个元素占4B。数组的15个元素有4种类型(60B)
            2. 最初的12个元素是直接索引项，给出文件最初的12个逻辑块号对应的物理块号
            3. 索引12是一次间接索引块，对应的文件逻辑块号从12到（b/4）+11，b是盘块大小(字节，4个字节描述一个盘块)
            4. 索引13是二次间接索引块，对应的文件逻辑块号从b/4+12到 (b/4)^2+(b/4)+11
            5. 索引14是三次间接索引块，对应的文件逻辑块号从 (b/4)^2+(b/4)+12到(b/4)^3+(b/4)^2+(b/4)+11
            6. 一个间接块的大小是 4 **kb**
            7. 如果符号链接文件的路径名小于**60**字符，也放在索引表中
    - 设备文件、管道文件、套接字文件：信息存放在索引节点中，无需数据块??????????????

Ext2的主存数据结构

1. 为了提高效率，ext2磁盘数据中的大部分信息被复制到了RAM主存中，避免大量的重复的磁盘读写操作
2. 数据结构

1. 超级块：
    1. 磁盘结构：ext2_super_block
    2. 内存结构：ext2_sb_info
2. 索引节点：
    1. 磁盘结构：ext2_inode
    2. 内存结构：ext2_inode_info

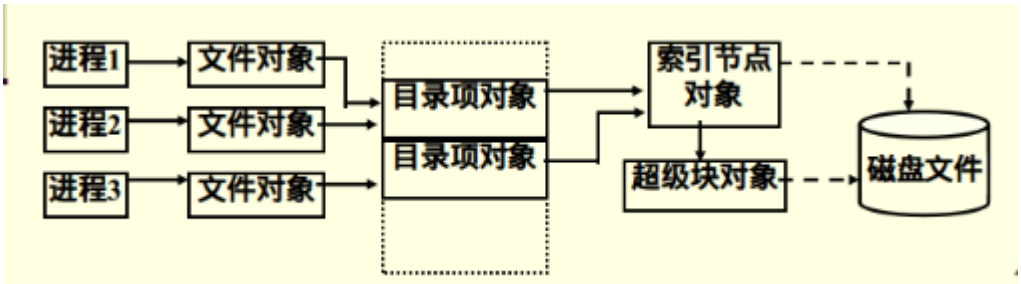
    ????里面的vfs_inode是不是下面的索引节点对象????

磁盘空间管理

1. 磁盘块和索引节点的分配和回收
2. 文件的数据块和其索引节点尽量在同一个块组中
3. 文件和它的目录项尽量在同一个块组中
4. 父目录和子目录尽量在同一个块组中 : 快速进入目录
5. 每个文件的数据块尽量连续存放 : 减少访问次数

Linux虚拟文件系统

1. 虚拟文件系统
    - 虚拟文件系统工作在核心态
    - VFS的主要思想在于引入一个通用的文件模型，该模型能够表示其支持的所有文件系统
    - VFS涉及的所有数据结构在系统运行时才在内存建立，在磁盘上没有存储
    - 不同的文件系统和Linux的虚拟文件系统VFS之间的接口是通过数据结构ile_operations实现的(包括**Ext2**)
2. 虚拟文件系统涉及的数据结构
    - 超级块对象：
        - Linux为每个安装好的文件系统都建立一个超级块对象
        - 存放整个文件卷的资源管理信息
        - 每个文件系统对应一个
        - void *s_fs_info; 指向一个具体文件系统的超级块结构
    - 索引节点对象：
        - FCB
    - ????????????目录项对象????????????
        - 一旦一个目录项被读入主存，VFS将其装换为dentry的目录项对象
        - 代表一个目录项，是文件路径的组成部分，存放目录项和对应文件之间的连接信息
        - 没有对应的磁盘映像
        - 对应一个索引节点
        - struct inode *d_inode; 指向inode对象（内存的inode）
    - 文件对象（不在磁盘上存储）
        - 记录了进程与打开的文件之间的交互信息
        - 包含文件读写指针
        - struct dentry *f_dentry; 指向目录项对象
        - ？？？？？？为什么有页高速缓存的数据结构，怎么用的？？？？？
    - 组织信息



- 与进程打开文件有关的数据结构

        - struct file **fd; 指向文件对象指针数组
        - struct file *fd_array[ ];文件对象指针数组，存放文件对象的指针
        - 每个进程最多同时打开的文件数为1024个
    - 进程打开文件的过程
        1. 进程->文件对象控制块->文件对象->目录对象->索引节点对象
3. 文件系统的注册与安装
    - Linux系统支持的所有文件系统在使用前必须进行注册，调用register_filesystem()完成注册。
    - 之后，再用mount命令将该文件系统安装到根文件系统的某个目录结点上
    - 安装表 : 内核将安装点与被安装的文件系统信息保存在vfsmount结构中，形成一个链式安装表
        - 指向安装点的目录项对象
        - 指向被安装文件系统的根目录
4. VFS系统调用的实现
    1. open
    2. close
    3. read
    4. write