History:
UNIX: Ken Thomphon & D.Richie (Bell)
GNU: (GNU is Not Unix) Richard Stallman(MIT)
LINUX Linus Torvalds （芬兰） Unix + GNU(内核+应用工具)

Chmod chmod +/- r/w/x chmod nnn(三位八进制数)
Ln：ln [Option] Target Link_Name(默认为硬链接)
常用选项：-s 做符号链接 -b 备份
Ls：ls [Option]
常用选项： -a (all)显示所有，包括隐藏文件
-l (long)长格式显示（modification time）
-h (human readable)文件大小以 B，K B，M B 等人易读的形式显示
-r (reverse)文件名逆序显示 -s 文件名前显示其大小
-X (extension)按文件扩展名字母
-S (size)按文件大小从大到小显示
-t (time)按文件被修改时间从大到小显示
-ul 长格式，显示 access time
Touch: touch [Option] FILE
作用：修改 FILE 的 access 和 modify 时间（时间戳）(如果文件不存在,就会创建一个新文件)
常用选项：-a 只修改 access time -m 只修改 modification time
-t STAMP 时间戳用 STAMP，而不是当前时间
Cat：cat [OPTION] [FILE]
Concatenate FILE(s), or standard input, to standard output
常用选项：-n 给每一行编 -b 给每个非空行编号
-T 显示 TAB 为^I -E 显示行末尾，加$
-v 显示不可打印字符 -s 连续多个空行只显示一行
More：more [Option] [-num] [+/ PATTERN] [+num] FILE
常用选项：-num 指定每屏显示的行数 +num 从第 num 行开始
-s 多行空格只显示一行
+/ 查找匹配的串
Less: 与 more 类似,但是允许向后翻页(more 不能)
Head：head [Option] [FILE] 如果 FILE 没有，或为'-'的话，获取 stdin
作用：显示文件前 10 行到 standard output
常用选项：-c[-]N -cN 前 N 个 bytes，-c-N 除了后 N 个 bytes
-n[-]N -nN 前 N 行， -n-N 除了后 N 行

System Calls:
(1). **int read(int fd, char *buf, int n);** 读文件
(2). **int write(int fd, char *buf, int n);** 写文件
(3). **int open(char *name, int rwmode);** 打开文件
rwmode: 0 read 1 write 2 w/r
(4). **int creat(char *name, int perms);** 创建文件 perms: rwx 三位八进制数
(5).**int lseek(int fd, long offset, int origin)** 在文件中定位
origin: 0 文件开始 1 当前位置 2 文件结束
(从当前读写位置移动到 offset 处, offset 是相对 origin 计算得出的)
定位至文件结尾处: lseek(fd, 0L, 2);
定位至文件开始处: lseek(fd, 0L, 0);
取得当前位置: pos = lseek(fd, 0L, 1);
(6).创建低级进程: 新进程会覆盖老进程（几个函数的不同之处主要在于命令传递方式）
int execl(const char* path, const char *arg, …)
int execlp(const char* file, const char *arg, …)
int execle(const char* path, const char *arg, …, char *const envp[]);
int execv(const char* path, char *const argv[]);
int execvp(const char* file, char *const agrv[]);
(7).**int dup(int fd)**
在最低序号的未分配的文件描述符上复制文件描述符 fd, 返回指向相同打开文件的一个新的文件描述符.
int dup2(int oldfd, int newfd); 让 newfd 和 oldfd 指向同一个文件描述符 oldfd, 如果可能的话关闭 newfd
(8)**pid_t fork(void);** 创建一个子进程（Copy-On-Write）
int clone(int (*fn)(void *), void * child_stack, int flags, void * args); 子进程可以使用父进程一些 execution context
pid_t vfork(void); 创建一个子进程并把父进程挂起
(9)**void exit(int status)** 结束进程
pid t wait(int *status) 父进程挂起，直到其中一个子进程结束

(10).**int fcntl(int fd, int cmd);**
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock* lock);
在以文件描述符为 fd 的文件上执行 cmd 指定的命令
Cmd: F_DUPFD, F_GETFD, F_SETFD
mode_t umask(mode_t mode); 把 calling process 的 file mode 设置为 mode & 0777
(11)**int chdir(const char* path)** 把当前工作目录改为 path 指定的目录
int fchdir(int fd); directory 是由 fd 指定的，其它跟 chdir 类似
int chmod(const char* path, mode_t mode)由 path 指定的文件的模式改为 mode
(12).**int mkdir(const char* path, mode_t mode)** 以 mode 为模式创建由 path 指定的目录
int rmdir(const char* path) 移除由 path 指定的目录，该目录必须为空
int rename(const char* oldpath, const char* newpath) 改变文件的位置或名称
(13).**int link(const char* oldpath, const char* newpath)** 为已存在的文件创建一个链接，创建后两个文件名指向同一个文件，完全等同
int symlink(const char* oldpath, const char* newpath) 为 oldpath 指定的文件创建一个名为 newpath 的符号链接（可能存在，也可能不存在）
int unlink(const char* path)删除 path 指定的文件，如果 path 指定的文件为该文件的最后一个链接，那么调用 unlink 后，文件被删除，而且空间被释放。如果还有进程在使用该文件，那么直到进程结束，文件才会被删除。

Tail：tail [Option] [FILE] 如果 FILE 没有，或为'-'的话，获取 stdin
作用：显示文件后 10 行到 standard output
常用选项：-cN 后 N bytes
-nN 后 N lines
Grep: grep [Option] PATTERN [FILE]
作用：显示文件中匹配特定模式的行
常用选项：-A num 匹配行后显示 num 行(after)
-B num 匹配行前显示 num 行(before)
-b 显示匹配处至文件开始处的 bytes offset
-C NUM 在匹配行前后各显示 NUM 行的语境
-c 统计匹配次数
-E PATTERN 解释为扩展的正则表达式(extended)
-F PATTERN 解释为固定字符串（fixed）
-G PATTERN 解释为基本正则表达式
-n 显示匹配行在文件中的行号
-v 反向匹配(与 pattern 不匹配)
Egrep = grep -E
Fgrep = grep -F
Find：find [-H] [-L] [-P] [path] [expression]
-H Never follow symbolic links.(Default)
-L Follow symbolic links.
-P Do not follow symbolic links except while processing the command line arguments
Path: 指定要查找的目录
Expression: 由 option 和 action 组成：
常用的 option 有：
-amin n 文件最后一次访问是在 n 分钟以前
-cmin n 文件最后一次修改是在 n 分钟以前
-fstype type 文件在类型为 type 的文件系统中
-name pattern 文件名匹配 pattern 类型(pattern 加双引号)
-iname pattern 文件名不区分大小写的情况下与 pattern 匹配
-type c 按文件类型进行搜索
常用的 action 有：
-delete
-exec cmd {} \ ;(把 find 的结果作为 cmd 的参数,运行 cmd)
Kill: kill –signal pid(向 PID 为 pid 的进程发送一个信号)
常用的 signal: kill -9 -1 kill all the processes you can kill
Kill –l 11 把 11 转化为信号名称(这里为-l(L))
SIGHUP 1 exit; SIGINT 2 exit, interrupt from keyboard;
SIGKILL 9 exit, cannot be blocked, forced terminate;
SIGSEGV 11 dump, Invalid memory reference;
SIGTERM 15 exit(Default); SIGSTOP 15 stop, Stop the process;
Fg 可用来激活某个被挂起的进程并使它在前台运行。
Bg 通过将暂挂的作业作为后台作业运行，可在当前环境中恢复执行这些作业
JobID:%Number 用作业号指代作业。%String 指代以指定的字符串作为其名称的开头的作业。 %?String 指代其名称包含指定字符串的作业。 %+ OR %% 指代当前作业。 %- 指代前一个作业
! 查看终端下命令执行历史
!! 最近一次执行的命令 **!n** .bash_history 中第 n 个命令
!-n .bash_history 倒数第 n 个命令

Shell 编程:
"…." 在…中的$ `…` 和\得到解释后,将…作为文本文字
'….' 将….当作文本文字
`….` 运行….中的命令后,输入结果代替….
$# 命令行参数个数
$? 上一个命令的执行结果,如果执行成功,为 0,否则为 1
$!** 最后在后台执行的进程的 ID **$$ 当前进程 ID
$-** 打开的 SHELL 的清单 **$@ 命令行参数列表
P1&&P2 运行 P1;若成功,再运行 P2
P1||P2 运行 P1;若不成功,再运行 P2
> file 输出重定向到 file **< file** 输入重定向 file
>> 输出重定向, 不覆盖原有输出设备上的内容, 在原有内容的基础上追加
<< END 将这个命令后直到"END"前的内容作为输入(here doc)
m>&n 将 m 指定的输出与 n 指定的输出合并,所有输出送到 n 指定的输出设备
P1&P2 P1,P2 同时执行; **P &** 将进程 P 放在后台运行
P1;P2 先执行 P1,再执行 P2

Gcc: gcc [option] file …
常用选项: -c compile, assemble, not link,生成.o 文件
-o 指定输出文件 -E preprocess only, 结果输出到 stdout
-S compile only 生成.s 文件
Du: du [option] .. [FILE](计算文件或目录的磁盘使用情况)
常用选项: -a all; –c total -b bytes; –h human_readable
Df: df [option] … [FILE] 显示文件系统相关信息
常用选项: -a all; -h human_readable; -i inode; -t TYPE
Ctrl-Z 挂起一个进程 Ctrl-D EOF
Ctrl-U 删除一行光标前字符 Ctrl-K 删除一行光标后字符
Ctrl-P 上一次执行的命令 Ctrl-C 终止一个进程
Mount: mount –t TYPE DEV MOUNT POINT

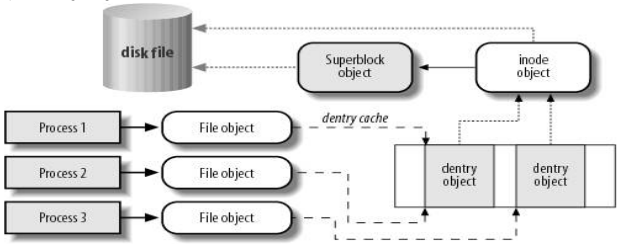
(14).文件系统 I 节点相关信息获得:
struct stat stbuf;
stat(char *name, &stbuf); 文件由 name 指定
fstat(fd, &stbuf); 文件由 fd 指定

(15)**int gettimeofday(struct timeval* tv, struct timezone *tz);** 获取时间
int settimeofday(const struct timeval* tv, const struct timezone *tz); 设置时间
struct timeval
{
time_t tv_sec; //seconds
suseconds tv_usec; //microseconds
};
Stcuct timezone
{
int tz_minuteswest; //minutes west of Greenwich
int tz_dsttime; //type of DST correction
};
int stime(time_t * t); 把系统时间设为 t，t 为从 00：00：00 GMT 1970.1.1 开始算的秒
time_t time(time_t * t) 得到系统时间，返回值为从 00:00:00 UTC 1970.1.1 开始算的秒，如果 t 不为 NULL，那么结果也将保存到 t 里
clock times(struct tms* buf); 获取进程时间,把当前进程时间存在 buf 里
struct buf
{
clock_t tms_utime; //user time
clock_t tms_stime; //system time
clock_t tms_cutime; // user time of dead children
clock_t tms_cstime; //system time of dead children
};

do_page_fault()工作原理: compares the linear address that caused the Page Fault against the memory regions of the `current` process; it can thus determine the proper way to handle the exception

(16)**int uname(struct utsname* buf);**获取当前内核的名称和相关状态
struct utsname
{
char sysname[];
char nodename[];
char release[];
char version[];
char machine[];
#ifdef _GNU_SOURCE
char domainname[];
#endif
};
int brk(void *end_data_segment);把进程的 data segment 的 end 的值设为 end_data_segment，以达到改变 data segment 大小的目的。
void *mmap(void * start, size_t length, int prot, int flags, int fd, off_t offset);
将由 fd 指定的文件（或设备）的 offset 处开始的 length 字节映射到内存，start 指定希望映射后的内存的起始地址，但不一定是，所以一般可以设为 0，而映射后的真正起始地址将由 mmap() 函数返回。Prot 指定贴身后内存的保护级别。
int munmap(void* start, size_t length); 取消文件（设备）的内存映射
uid_t getuid() 获取当前进程 real user 的 ID
uid_t geteuid() 获取当前进程 effective user 的 ID
int setuid(uid_t uid) 设置当前进程 effective user 的 ID
(17). **typedef void (*sighandler_t)(int);**
sighandler signal(int signum, sighandler_t handler); 为由 signum 指定的信号添加新的处理该信号的函数
int kill(pid_t pid, int sig) 向由 pid 指定的进程发由 sig 指定的信号
int pipe(int filedes[2]) 为管道创建一对文件描述符存在 filedes[0]和 filedes[1]中，其中 filesdes[0]是用来读的，filedes[1]是用来写的

其它比较零散的知识点:
1. 系统启动过程:
BIOS→boot loader→setup()→start_up32()→start_kernel()
2. VFS 中的 common file model:
(1). Super_block object: 可以称之为 filesystem control block, 主要存储被挂载了的文件系统的相关信息
(2). Inode object: 可以称之为 file control block, 用来存储一般文件的相关信息
(3). F ile object: 存储已经打开的文件与进程交互的信息
(4). Dentry object: 存储目录与相关文件的信息



```
/*
 * error_code:
 *      bit 0 == 0 means no page found, 1 means protection fault
 *      bit 1 == 0 means read, 1 means write
 *      bit 2 == 0 means kernel, 1 means user-mode
 *      bit 3 == 1 means use of reserved bit detected
 *      bit 4 == 1 means fault was an instruction fetch
 */
fastcall void __kprobes do_page_fault(struct pt_regs *regs,unsigned long error_code)
{
        struct task_struct *tsk;
        struct mm_struct *mm;
        struct vm_area_struct * vma;
        unsigned long address;
        int write, si_code;
        int fault;
      address = read_cr2();//从 CR2 寄存器中读取引起缺页的地址
        tsk = current; //当前进程描述符
        si_code = SEGV_MAPERR;
//接下来,检查引起缺页的地址是否属于当前进程地址空间
        if (unlikely(address >= TASK_SIZE)) {
                if(!(error_code&0x0000000d)&& malloc_fault(address) >= 0)//内核访
问不存在的 page frame
                        return;
                if(notify_page_fault(regs,error_code)== OTIFY_STOP)
                        return;
                goto bad_area_nosemaphore;        }
        if (notify_page_fault(regs, error_code) == NOTIFY_STOP)
                return;
        if (regs->eflags & (X86_EFLAGS_IF|VM_MASK))
        local_irq_enable();//将当前进程设为可以接受中断信号
        mm = tsk->mm;//内存管理描述符
if (in_atomic()||!mm)//检查内核是否在进行处理中断,在临界区
                goto bad_area_nosemaphore;
        if (!down_read_trylock(&mm->mmap_sem)) {
        if((error_code&4)=0&& !search_exception_tables(regs->eip))
                        goto bad_area_nosemaphore;
                down_read(&mm->mmap_sem);
        }//下面:查找一个包含 faulty address 的内存区域
     vma = find_vma(mm, address);
        if (!vma)//没有找到包含 faulty address 的内在区域
                goto bad_area;
if (vma->vm_start <= address)//找到了.且在当前进程的地址空间
                goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN))//缺页不发生在用用户空间
                goto bad_area;
        if (error_code & 4) {//?
                if (address + 65536 + 32 * sizeof(unsigned long) < regs->esp)
                        goto bad_area;
        }
        if (expand_stack(vma, address))
                goto bad_area;
good_area://正常的缺页处理程序由此开始
        si_code = SEGV_ACCERR;
        write = 0;
        switch (error_code & 3) {
                default:        /* 3: write, present */
                                /* fall through */
                case 2:         /* write, not present */
                        if (!(vma->vm_flags & VM_WRITE))
                                goto bad_area;
                        write++;
                        break;
                case 1:         /* read, present */
                        goto bad_area;
                case 0:         /* read, not present */
                        if (!(vma->vm_flags & (VM_READ | VM_EXEC | VM_WRITE)))
                                goto bad_area;
        }
 survive://这里为真正的缺页处理的地方
        fault = handle_mm_fault(mm, vma, address, write);//处理缺页
        if (unlikely(fault & VM_FAULT_ERROR)) {
                if (fault & VM_FAULT_OOM)
                        goto out_of_memory;
                else if (fault & VM_FAULT_SIGBUS)
                        goto do_sigbus;
                BUG();
        }
        if (fault & VM_FAULT_MAJOR)
                tsk->maj_flt++;
        else
                tsk->min_flt++;
        if (regs->eflags & VM_MASK) {
                unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
                if (bit < 32)
                        tsk->thread.screen_bitmap |= 1 << bit;
        }
        up_read(&mm->mmap_sem);
        return;
bad_area://faulty address 不属于当前进程空间
        up_read(&mm->mmap_sem);
bad_area_nosemaphore://user mode accesses just cause a SIGSEGV
        if (error_code & 4) {
                local_irq_enable();
                if (is_prefetch(regs, address, error_code))
                        return;
                if(show_unhandled_signals&&          unhandled_signal(tsk,          SIGSEGV)
&&printk_ratelimit()) {
                        printk("%s%s[%d]: segfault at %08lx eip %08lx "
                            "esp %08lx error %lx\n",
tsk->pid > 1 ? KERN_INFO : KERN_EMERG,
```

```
    tsk->comm, tsk->pid, address, regs->eip,
                        regs->esp, error_code);
            }
            tsk->thread.cr2 = address;
            /* Kernel addresses are always protection faults */
            tsk->thread.error_code = error_code | (address >= TASK_SIZE);
            tsk->thread.trap_no = 14;
            force_sig_info_fault(SIGSEGV, si_code, address, tsk);
            return;//向进程发送 SIGSEGV 信号,且要保证不能被阻塞
    }
no_context://缺页是在 kernel mode 下发生的
    if (fixup_exception(regs))
            return;
    if (is_prefetch(regs, address, error_code))
            return;
    bust_spinlocks(1);
    if (oops_may_print()) {
            __typeof__(pte_val(__pte(0))) page;
            if (address < PAGE_SIZE)
                    printk(KERN_ALERT "BUG: unable to handle kernel NULL "
"pointer dereference");
            else
                printk(KERN_ALERT "BUG: unable to handle kernel paging"
                                    " request");
            printk(" at virtual address %08lx\n",address);
            printk(KERN_ALERT " printing eip:\n");
            printk("%08lx\n", regs->eip);
            page = read_cr3();
            page = ((__typeof__(page) *) __va(page))[address >> PGDIR_SHIFT];
            if ((page >> PAGE_SHIFT) < max_low_pfn
                  && (page & _PAGE_PRESENT)) {
                    page &= PAGE_MASK;
                    page = ((__typeof__(page) *) __va(page))[(address >>
PAGE_SHIFT)
                                                            &
(PTRS_PER_PTE - 1)];
                    printk(KERN_ALERT "*pte = %0*Lx\n", sizeof(page)*2,
(u64)page);//这是主要是处理在 kernel mode 下发生 fault, 首先是报告
            }//内核有 BUG,打出出错信息,然后直接中止进程
    }
    tsk->thread.cr2 = address;
    tsk->thread.trap_no = 14;
    tsk->thread.error_code = error_code;
    die("Oops", regs, error_code);
    bust_spinlocks(0);
    bust_spinlocks(0);
    do_exit(SIGKILL);
//在 kernel 状态下,引起 fault 有两种情况: (1).系统调用的参数中含有地址//,处理时发生错误; (2). Kernel 存在 BUG
```

```
out_of_memory://内存溢出,或不能正解处理缺页
    up_read(&mm->mmap_sem);
    if (is_init(tsk)) {
            yield();
            down_read(&mm->mmap_sem);
            goto survive;
    }
    printk("VM: killing process %s\n", tsk->comm);
    if (error_code & 4)
            do_exit(SIGKILL);
    goto no_context;

do_sigbus://总线出错
    up_read(&mm->mmap_sem);

    /* Kernel mode? Handle exceptions or die */
    if (!(error_code & 4))
            goto no_context;

    /* User space => ok to do another page fault */
    if (is_prefetch(regs, address, error_code))
            return;
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    force_sig_info_fault(SIGBUS, BUS_ADRERR, address, tsk);
}
```

do_page_fault()函数，该函数有两个参数：一个是指针,指向异常发生时寄存器值存放的地址。另一个错误码,由三位二进制信息组成：第 0 位——访问的物理页帧是否存在；第 1 位——写错误还是读错误或执行错误；第 2 位——程序运行在核心态还是用户态。do_page_fault()函数的执行过程如下：1. 首先得到导致异常发生的线性地址,对于 X86 该地址放在 CR2 寄存器中。2. 检查异常是否发生在中断或内核线程中,如是,则进行出错处理。
3. 检查该线性地址属于进程的某个 vm_area_struct 区间。如果不属于任何一个区间,则需要进一步检查该地址是否属于栈的合理可扩展区间。一但是用户态产生异常的线性地址正好位于栈区间的 vm_start 前面的合理位置,则调用 expand_stack()函数扩展该区间,通常是扩充一个页面,但此时还未分配物理页帧。至此,线性地址必属于某个区间。
根据错误码的值确定下一个步骤:如果错误码的值表示为写错误,则检查该区间是否允许写,不允许则进行出错处理。如果允许就是属于前面提到的写时拷贝(COW)。如果错误码的值表示为页面不存在,这就是所谓的按需调页(demand paging)。写时拷贝的处理过程：首先改写对应页表项的访问标志位,表明其刚被访问过,这样在页面调度时该页面就不会被优先考虑。如果该页帧目前只为一个进程单独使用,则只需把页表项置为可写。如果该页帧为多个进程共享,则申请一个新的物理页面并标记为可写,复制原来物理页面的内容,更改当前进程相应的页表项,同时原来的物理页帧的共享计数减一。

```
/*
*    Ok, this is the main fork-routine.
* It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long do_fork(unsigned long clone_flags,
                unsigned long stack_start,
                struct pt_regs *regs,
                unsigned long stack_size,
                int __user *parent_tidptr,
                int __user *child_tidptr)
{
        struct task_struct *p;
        int trace = 0;
        struct pid *pid = alloc_pid();//为子程分配一个PID
        long nr;
        if (!pid)//PID分配失败
                return -EAGAIN;
        nr = pid->nr;//检查父进程是否在被跟踪(?)
        if (unlikely(current->ptrace)) {
                trace = fork_traceflag (clone_flags);
                if (trace)
                        clone_flags |= CLONE_PTRACE;
        }
//把父进程的进程描述符复制,返回创建的task_struct的地址
        p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr,
child_tidptr, pid);
        /* Do this prior waking up the new thread - the thread pointer
         * might get invalid after that point, if the thread exits quickly.
         *///复制成功
        if (!IS_ERR(p)) {
                struct completion vfork;
                if (clone_flags & CLONE_VFORK) {
                        p->vfork_done = &vfork;
                        init_completion(&vfork);
                }
//当前进程的状态为TASK_STOPPED, 一直保持这个状态,直到
//有中断信号来改变其状态
                if ((p->ptrace & PT_PTRACED) || (clone_flags &
CLONE_STOPPED)) {
                        /*
                         * We'll start up with an immediate SIGSTOP.
                         */
                        sigaddset(&p->pending.signal, SIGSTOP);
                        set_tsk_thread_flag(p, TIF_SIGPENDING);
                }
//当前进程的状态不为TASK_STOPPED, 调整高度参数,了里程创建//完毕,
可以加入调度队列
if (!(clone_flags & CLONE_STOPPED))
                        wake_up_new_task(p, clone_flags);
                else
                        p->state = TASK_STOPPED;
        if (unlikely (trace)) {//父进程在被跟踪
                        current->ptrace_message = nr;
                        ptrace_notify ((trace << 8) | SIGTRAP);
                }//结束当前进程,向父进程发SIGTRAP信号
//下面: 如果vfork flag 是1, 那么父进程将被挂起,等待子进程运行
                if (clone_flags & CLONE_VFORK) {
                        freezer_do_not_count();
                        wait_for_completion(&vfork);
                        freezer_count();
                        if (unlikely (current->ptrace & PT_TRACE_VFORK_DONE)) {
                                current->ptrace_message = nr;
                                ptrace_notify ((PTRACE_EVENT_VFORK_DONE << 8) |
SIGTRAP);
                        }
                }
        } else {//复制失败,释放资源
                free_pid(pid);
                nr = PTR_ERR(p);
        }
        return nr;
}
```

do_fork 分析:

1) 调用 alloc_task_struct（）为子进程控制块分配空间。严格地讲，此时子进程还未生成。
2) 把父进程控制块的值全部赋给子进程控制块。
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) 修改子进程控制块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) 依次调用 copy_files(),copy_fs(), copy_sighand( ), copy_mm( )分别复制父进程文件处理、信号处理及进程空间的信息。以上函数的具体行为取决 clone_flags 参数，例如,copy_mm( )时,如果 clone_flags 包含有 CLONE_VM 标志，则子进程共享父进程的空间，不会进行复制。
7) 调用 copy_thread( )初始化子进程的核心模式栈时,核心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存储寄存器 eax 值的位置被置 0,这个值就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