**进程描述符（内核2.6）**

**task\_struct结构**：包含了进程相关的所有信息

      state：描述进程状态（可运行，可中断的等待，不可中断的等待，暂停，跟踪）

      thread\_info：线程描述符

      flags：

      run\_list

      tasks

      mm:指向内存区描述符mm\_struct的指针

      real\_parent:

      parent：

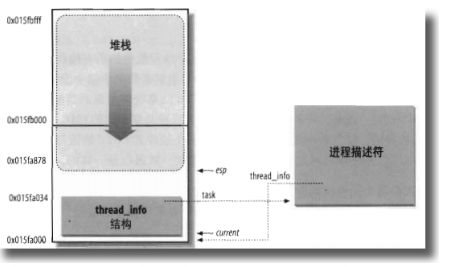
      tty：指向进程相关的tty\_struct的指针

      fs：指向当前目录fs\_struct的指针

      files：指向文件描述符的指针

      signal：指向所接收信号signal\_struct的指针

      对每个进程，Linux内核都把两个不同的数据结构紧凑的存放在一个单独为进程分配的内存区域中：一个是内核态的进程堆栈，另一个是thread\_info，叫做线程描述符。



 注：进程描述符在动态内存中，而不是放在永久分配给内核的内存区。内核态的进程访问处于内核数据段的栈，当用户态刚切到内核态后，进程的内核栈是空的，esp寄存器指向栈顶端。

**标识进程：**进程和进程描述符之间是严格一一对应关系，PID放在进程描述符的pid字段，pid被顺序编号，上限为32767。新建进程时PID顺序累加，当达到上限后，开始循环使用未被占用的号码。

**进程链表：**  
  
      进程链表把所有进程的描述符链接起来，它的本质是一个双向链表。每个task\_struct结构都包含一个list\_head结构的tasks字段，这个类型的prev和next字段分别指向前面和后面的task\_struct元素。有一个宏for\_each\_process，它的功能是扫描整个进程链表。选择进程运行实质是选择一个处于TASK\_RUNNING状态的进程，linux 2.6开始建立多个可运行进程链表，每种进程优先权对应一个链表。

**2.进程间的关系**

    父子/兄弟关系

    对应进程描述符中字段:real\_parent / parent / children / sibling

    同一进程组的非亲属关系:一个进程可能是进程组或一个会话的领头进程,也可能是线程组的领头进程

    对应字段: group\_leader / signal->pgrp / tgid / signal->session

**3.进程组织**

    运行队列:运行队列链表把所有处于TASK\_RUNNING状态的进程组织在一起

    等待队列:等待队列表示一组睡眠的进程，进程经常等待一些特定事情的发生，内核可以把等待特定事情的进程放入合适的等待队列，用于终端处理,进程同步等。在这个链表中，有两种数据结构：等待队列头（wait\_queue\_head\_t）和等待队列项

（wait\_queue\_t）。

struct \_\_wait\_queue{

  unsigned int flags; //0表示非互斥进程,1表示互斥进程

  struct task\_struct \*task; //进程描述符地址

  wait\_queue\_func\_t func;//判断该进程用什么方式唤醒,互斥或者非互斥

  struct list\_head task\_list;

};

typedef struct \_\_wait\_queue wait\_queue\_t;

    init\_waitqueue\_head 初始化等待队列头，对lock和task\_list进行初始化

    init\_waitqueue\_entry(q,p) 初始化等待元素（非互斥）

    内核通过default\_wake\_function()唤醒睡眠进程，从等待队列中删除指定元素。

**4.fork，vfork，clone**

用于Linux创建新的进程这三个函数分别调用了sys\_fork、sys\_vfork、sys\_clone，最终都调用了do\_fork函数，差别在于参数的传递和一些基本的准备工作不同。三者最终达到的最本质的目的都是创建一个新的进程。

**fork**是最简单的调用，不需要任何参数，仅仅是在创建一个子进程并为其创建一个独立于父进程的空间。fork使用COW（写时拷贝）机制，并且COW了父进程的栈空间。

**vfork**也是创建一个子进程，但是子进程共享父进程的空间。在vfork创建子进程，因为子进程使用父进程的空间资源，因此父进程阻塞，直到子进程执行了exec()或者exit()。＃exec时如何COW

**clone**是fork的升级版本，不仅可以创建进程或者线程，还可以指定创建新的命名空间（namespace）、有选择的继承父进程的内存、甚至可以将创建出来的进程变成父进程的兄弟进程等等。clone和fork的调用方式也很不相同，clone调用需要传入一个函数，该函数在子进程中执行。此外，clone和fork最大不同在于clone不再复制父进程的栈空间，而是自己创建一个新的。

    int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg)

    fn为函数指针，此指针指向一个函数体，即想要创建进程的静态程序；child\_stack为给子进程分配系统堆栈的指针；arg就是传给子进程的参数；

    flags为要复制资源的标志：

    CLONE\_PARENT   创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”

    CLONE\_FS           子进程与父进程共享相同的文件系统，包括root、当前目录、umask

    CLONE\_FILES      子进程与父进程共享相同的文件描述符（file descriptor）表

    CLONE\_NEWNS   在新的namespace启动子进程，namespace描述了进程的文件hierarchy

    CLONE\_SIGHAND   子进程与父进程共享相同的信号处理（signal handler）表

    CLONE\_PTRACE   若父进程被trace，子进程也被trace

    CLONE\_VFORK     父进程被挂起，直至子进程释放虚拟内存资源

    CLONE\_VM           子进程与父进程运行于相同的内存空间

    CLONE\_PID          子进程在创建时PID与父进程一致

    CLONE\_THREAD    子进程与父进程共享相同的线程群

    fork()可以看出是完全版的clone()，而clone()克隆的只是fork()的一部分。

**exec函数：**

在Linux中，并不存在exec()函数，exec指的是一组函数，一共有6个，分别是：

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 argv[]);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

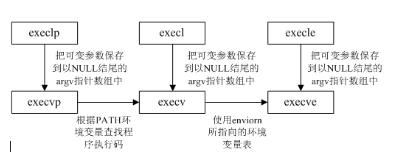
  这6个函数中真正的系统调用只有execve，其他5个都是库函数，它们最终都会调用execve这个系统调用

l：以逗号分隔的参数列表

v：参数以一个以NULL结尾的字符串数组的指针形式

e：函数传递指定参数envp，允许改变子进程的环境

p：是一个以NULL结尾的字符串数组指针，查找子程序文件



exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，只有进程ID等仍保持原样。调用失败时，会设置返回-1，然后从原程序的调用点接着往下执行。

**内核态和用户态：**

    内核态与用户态是操作系统的两种运行级别,有Ring0-Ring3三种级别的运行模式，Ring0级别最高，Ring3最低。Linux使用了Ring3级别运行用户态，Ring0作为 内核态，没有使用Ring1和Ring2。Ring3状态不能访问Ring0的地址空间，包括代码和数据。Linux进程的4GB线性地址空间，3G-4G部分是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块。用户运行一个程序，该程序所创建的进程开始是运在用户态的，如果要执行文件操作，网络数据发送等操作，必须通过write，send等系统调用，这些系统调用会调用内核中的代码来完成操作，这时，必 须切换到Ring0，然后进入3GB-4GB中的内核地址空间去执行这些代码完成操作，完成后，切换回Ring3，回到用户态。这样，用户态的程序就不能 随意操作内核地址空间，具有一定的安全保护作用。