系统调用：

系统调用（System Call）是操作系统为在用户态运行的进程与硬件设备（如CPU、磁盘、打印机等）进行交互提供的一组接口。当用户进程需要发生系统调用时，CPU 通过软中断切换到内核态开始执行内核系统调用函数。

系统调用的三种方法：

1. 通过glibc提供的库函数：

glibc是Linux下使用的开源的标准C库，它是GNU发布的libc库，即运行时库。glibc为程序员提供丰富的API（Application Programming Interface），除了例如字符串处理、数学运算等用户态服务之外，最重要的是封装了操作系统提供的系统服务，即系统调用的封装。那么glibc提供的系统调用API与内核特定的系统调用之间的关系是什么呢？

通常情况，每个特定的系统调用对应了至少一个 glibc 封装的库函数，如系统提供的打开文件系统调用 sys\_open 对应的是 glibc 中的 open 函数；

其次，glibc一个单独的API可能调用多个系统调用，如glibc提供的printf函数就会调用如sys\_open、sys\_mmap、sys\_write、sys\_close等等系统调用；

另外，多个API也可能只对应同一个系统调用，如glibc下实现的malloc、calloc、free等函数用来分配和释放内存，都利用了内核的sys\_brk的系统调用。

实例：通过glibc提供的chmod函数（man 2 chmod）来改变文件的属性：

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <stdio.h>

int main()

{

int rc;

rc = chmod("/etc/passwd", 0444);

if (rc == -1)

fprintf(stderr, "chmod failed, errno = %d\n", errno);

else

printf("chmod success!\n");

return 0;

}

其中errno在/usr/include/asm-generic/errno\*.h中定义。

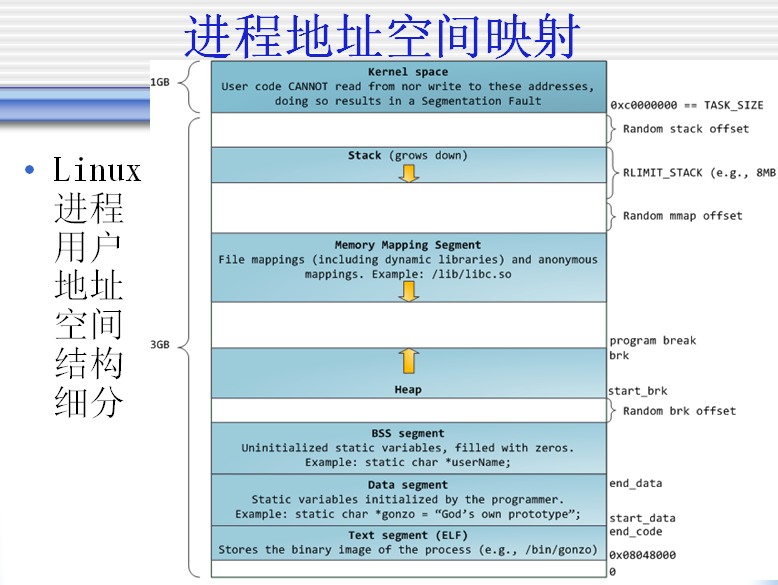
1. 使用syscall直接调用：

当glibc没有封装某个内核提供的系统调用时，救没法通过上面的方法来调用该系统调用，比如自己手动往内核中添加系统调用。此时我们可以使用glibc提供的syscall函数直接调用。该函数定义在unistd.h头文件中，函数原型如下：

int syscall(int number,...);

sysno是系统调用号，每一个系统调用都有唯一的系统调用号来标识。在sys/syscall.h中有所有可能的系统调用号的宏定义。

...

**Linux进程用户地址空间结构细分;**

**内核 4.8.0**

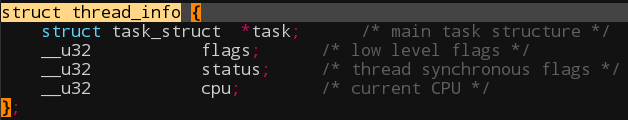
**进程描述符及任务结构：**

内核把进程的列表存放在叫做任务队列的双向循环链表中。链表中的每一项都是类型为task\_struct、称为进程描述符的结构。该结构定义在内核源码树include/linux/sched.h文件中。进程描述符中包含一个具体进程的所有信息。

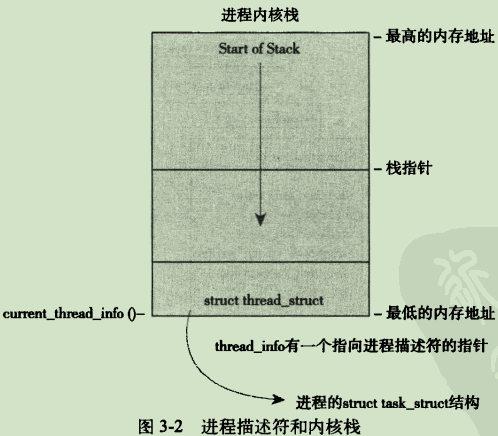
tash\_struct包含的数据能完整的描述一个正在执行的程序：它打开的文件，进程的地址空间，挂起的信号，进程的状态，还有其他更多信息。详细见该结构的具体定义。

在内核2.6之后，Linux通过slab分配器动态分配task\_struct结构，所以只需要在栈底(对于向下增长的栈来说)或栈顶(对于向上增长的栈来说)创建一个新的结构struct thread\_info。

在x86上，struct thread\_info文件在内核源码arch/x86/include/asm/thread\_info.h中定义：



其中\_\_u32定义： #typedef unsigned int \_\_u32;



**进程描述符的存放：**

内核通过唯一的进程标识值（PID）来标识每个进程，PID最大的默认值为32768（short int的最大值），但是这个值可以定义最大高达400万。在内核源码include/linux/threads.h中定义。在编译完的内核中，可以通过/proc/sys/kernel/pid\_max来提高。



通过/etc/sysctl.conf可以修改最大值：

添加：kernel.pid\_max = 所需要的值

在内核中，访问任务通常需要获得指向其task\_struct的指针。实际上，内核中大部分处理进程的代码都是直接通过task\_struct进行的。因此，通过current宏找到当前正在运行进程的进程描述符的速度就显得尤为重要。硬件体系结构不同，该宏的实现也不同，他必须针对专门的硬件体系结构做处理。有的硬件体系结构可以拿出一个专门的寄存器来存放指向当前进程task\_struct的指针，用于加快访问速度。而像x86这样的体系结构，寄存器不富裕，只能在内核栈的尾端创建thread\_info结构，通过计算偏移间接的查找task\_struct结构。

**进程状态：**

进程描述符中的state域描述了进程的当前状态，系统中的每个进程都必然处于五种进程状态的一种，该域的值也必为下列五种状态标志之一（define在文件include/linux/sched.h）：

1.TASK\_RUNNING（运行）：进程是可执行的；它或者正在执行，或者在队列中等待执行。这是进程在用户空间中执行的唯一可能状态；这种状态也可以应用到内核空间中正在执行的进程。

2.TASK\_INTERRUPTIBLE（可中断）：进程正在睡眠（也就是说它被阻塞），等待某些条件的达成。一旦这些条件达成，内核就会把这些进程状态设置为运行。处于此状态的进程也会因为接收到信号而提前被唤醒并随时准备投入运行。

3.TASK\_UNINTERRUPTIBLE（不可中断）：除了就算是接收到信号也不会被欢迎或准备投入运行外，这个状态与可打断状态相同。这个状态通常在进程必须在等待时不受干扰或等待时间很快就会发生时出现。由于处于此状态的任务对信号不做响应，所以较之可中断状态，使用得较少。

4.\_\_TASK\_TRACED:被其他进程跟踪的进程，例如通过ptrace对调试程序进行跟踪。

5.\_\_TASK\_STOPPED（停止）：进程停止执行；进程没有投入运行也不能投入运行。通常这种状态发生在接收到SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU等信号的时候。此外，在调试期间接收到任何信号，都会使进程进入这种状态。

