**一个进程的大致轮廓**

进程的产生是内核调用clone函数的结果，进程创建会创建一个内核栈，如果是用户空间调用的clone系统调用还会创建一个进程地址空间，进程比较重要的东西都在内核中，一些重要的结构如进程描述符都存放在内核栈中，进程从内核而生，进程用户空间的部分相当于调用进程内核部分提供的服务service

进程在用户空间运行时是在进程地址空间上运行，进程陷入内核或者直接是内核线程时是在内核栈及内核3-4G的地址范围运行

创建的进程如果没有提供用户空间服务的必要则就是单纯的内核线程，内核线程只在内核空间运行，其使用的局部变量和函数调用链放在内核栈中，内核栈一般是固定大小为一页或者两页，其使用的全局等变量则直接在内核空间的全局地址中3G-4G，内核中所有的进程共用此地址空间。

创建的进程如果要给用户空间提供服务，则会创建一个进程的地址空间，这个地址空间供进程的用户空间部分使用，这个空间一般为系统0-3G地址，当用户空间需要空间时，会向内核申请一段内存区域vm\_area\_struct,内核创建这个vm\_area\_struct结构后，将其链接进该进程用户空间的地址空间，然后返回一个用户空间的连续区域给用户空间，常见的vm\_area\_struct区域有.bss .text .data段，用户空间只能访问申请过的用户地址空间。

内核对于物理内存的管理是通过page结构管理，一个物理页分配一个page结构，进程的内核线程和用户空间的部分当需要空间时都最终调用的是内核的alloc\_page函数分配物理页，然后转换成内核空间一个地址或者用户空间一个地址。

**进程及进程用户空间的地址空间的详细介绍**

内核用task\_struct表示一个进程，task\_struct中的mm\_struct表示该进程的地址空间，mm\_struct是内存描述符，包含和进程地址空间有关的全部信息。当进程没有用户空间部分，也就是单纯的内核线程时，mm\_struct则指向NULL

系统中所有的mm\_struct结构体通过自身的mmlist域链接在一个双向链表中，首元素为init\_mm内存描述符，表示init进程的地址空间。

每个进程的内存描述符mm\_struct中有该进程所有内存区域vm\_area\_struct的链表，一个进程的所有VMA以链表和树形结构的形式分别存放在进程描述符的mmap和mm\_rb域。

进程地址空间虽然范围为1-3G,但并不是都会用到，进程把可被访问的合法地址空间称为内存区域，每个内存区域具有相同的读写执行属性，每个内存区域用vm\_area\_struct结构体描述，简称VMA,VMA是一段连续的虚拟内存范围。如进程的.bss .data .text等区域都对以相应的VMA

VMA可以被多个线程共享.这时访问VMA所指向的物理内存是一样的，VMA结构体中的vm\_flags域标志VMA所包含的物理内存在被程序处理的时候的限制，如可读可写可执行，是否可以共享等，特别的当vm\_flags为VM\_SHARD时标明内存区域包含的映射可以在多进程间共享，如C库的VMA是共享的，每个进程需要时都可以链接进自己的地址空间中

mmap()和do\_mmap();创建地址区间

内核使用do\_mmap()函数创建一个新的线性地址区间，如果不能和邻近的区间合并，则创建一个新的VMA,将一个地址区间加入到进程的地址空间中

unsigned long do\_mmap(struct file \*file,unsigned long addr,unsigned long len,unsigned long prot,unsigned long flag,unsigned long offset)

该函数映射由file指定的文件，具体映射的是从文件中偏移offset处开始，长度为len字节的范围内的数据，如果file为NULL且offset参数也是0，那么就代表这次映射没有和文件相关，该情况称作匿名映射，如果指定了文件及偏移量，则称为文件映射

addr是可选参数，指定搜索空闲区域的起始地址

prot参数指定内存区域中的页面的访问权限

flag参数指定VMA标志，如VMA是否可以被共享

返回新分配的地址区间的初始地址。

用户空间可以通过mmap()系统调用获取内核函数do\_mmap()的功能

**用户空间进程的创建具体过程**

进程的创建都是需要通过内核的clone()接口实现的，clone接口根据传入的参数来标明父子进程需共享的资源，用户空间的进程创建新进程一般用fork() vfor() \_\_clone()库函数来调用系统调用clone()

fork() vfor() \_\_clone() -> | ->clone()->do\_fork()->copy\_process()

copy\_process()函数完成如下工作：

1：调用dup\_task\_struct()为新进程创建一个内核栈，并复制一个父进程的thread\_info结构（包含task\_struct结构）

2：进程描述符task\_struct部分与父进程区分开来，子进程的状态设置为TASK\_UNINTERRUPTIBLE

3:调用alloc\_pid为新进场分配一个有效pid

4:最重要的部分，根据clone系统调用传入的参数标志，拷贝或者共享父进程打开的文件，文件系统信息，信号处理函数，进程地址空间，命名空间

copy\_process()接口成功返回后，新创建的子进程被唤醒并投入运行

特别注意第4点，传统fork的做法是把父进程所有资源复制给新进程，此时父子进程的打开的文件，文件系统信息，信号处理函数，进程地址空间，命名空间都是单独不共享的。特别是其中的进程地址空间，有时父进程地址空间的数据庞大，为了创建子进程迅速，后续的优化引入了写时拷贝机制。此时父子共享地址空间，子进程mm\_struct域指向父进程内存描述符mm\_struct，即task->mm = current->mm，当二者其一需要修改地址空间内容时，子进程才会拷贝一个自己的进程地址空间，如果不共享地址空间，则子进程会从slab缓存中分配一个mm\_struct结构体。

一般fork执行完后，子进程会执行exec族接口，这个接口并不是创建新进程，而是用一个全新的程序代替当前程序的进程地址空间各个内存区域，新的程序从main函数开始执行。所以当父进程执行fork后立马执行exec，则子进程拷贝父进程进程地址空间毫无意义。

现在讲述用户空间创建进程的库函数fork() vfor() \_\_clone()有什么区别：

vfork()的实现如下clone(CLONE\_VFORK | CLONE\_VM | SIGCHLD,0)

fork()的实现如下clone(SIGCHILD,0)

这些库函数都会调用clone系统调用，传入的参数各不相同,clone()系统调用传入的参数有如下几种

CLONE\_VM:父子进程共享地址空间

CLONE\_FILES：父子进程共享打开的文件

CLONE\_FS：父子进程共享文件系统信息

CLONE\_NEWNS：为子进程创建新的命名空间

CLONE\_SIGHAND：父子进程共享信号处理函数及被阻断的信号

CLONE\_VFORK：调用vfork,父进程睡眠等待子进程将其唤醒

备注：vfork接口比fork接口多了CLONE\_VFORK | CLONE\_VM 这两个参数，一个表示子进程开始执行，父进程睡眠，直到子进程退出或者执行exec,另外一个参数表示不拷贝父进程的页表项，父子进程共享地址空间。

备注：linux中没有所谓的线程概念，linux把所有的线程都当做进程来实现，线程仅仅是与其他进程共享某些资源的进程

备注：linux中的进程都是PID为1的进程的后代，task\_struct中有指向父进程的parent指针和一个children子进程的链表

**内核进程的创建具体过程**

只在内核运行的内核线程，内核线程是独立运行在内核空间的标准进程，没有独立的用户空间的进程地址空间，进程的进程描述符中的mm域指针被设置为NULL,他们只在内核空间运行，从来不切换到用户空间中去，内核线程可以被调度和抢占。

运行ps -ef可以看到内核线程，内核线程只能由其他的内核线程创建，内核是通过从kthreadd内核线程衍生出所有新的内核线程来自动处理这一点的。

创建内核线程的接口如下：

struct task\_struct \* **ktread\_create**(int (\*threadfn)(void \*data),void \*data,const char namefmt[],...)

这个接口的是实现会调用clone系统调用实现，clone具体完成内容可见上面的描述，新的进程运行threadfn函数，传入data参数，进程命名为namefmt,新创建的进程处于不可运行状态，通过**wake\_up\_process()**明确唤醒它。或者调用如下接口创建并运行

struct task\_struct \* **kthread\_run**(int (\*threadfn)(void \*data),void \*data,const char namefmt[],...)

内核线程通过调用do\_exit()或者如下接口指定特定进程退出

int kthread\_stop(struct task\_struct \*k);//传入的是创建时返回的task\_struct

**linux系统对于物理内存的管理**

内核对于物理内存是通过page结构管理的，一个物理页对应一个page结构，一页可以为4KB或者8KB.因系统而异，如支持4Kb页大小有1G物理内存的机器上，物理内存被划分为261244页，page结构表示物理页，其中的virtual域表示此物理页的虚拟地址，page结构体中还有页的引用计数，是否空闲，谁拥有这页

linux系统把物理页划分为区，形成不同的内存池，这样就可以根据用途进行分配了，所有的物理页可以分为四种，每种称为区，有：ZONE\_DMA ZONE\_DMA32区（物理内存中的0-16M）这个区的物理页由于物理地址符合DMA的要求，可用于DMA操作，ZONE\_NORMAL区（物理内存中的16M-896M）这个区的物理页在系统启动后会直接映射到内核空间中，ZONE\_HIGHEM区 （物理内存中的896M以后的内存）这个区的物理内存为高端内存，其中的物物理页不能永久映射到内核空间

系统的物理内存中的低端内存一般在系统启动以后，就会被直接映射到内核的常规地址空间，一般就是3G-3G+896M的地址，也就是内核空间如果调用kmalloc或者\_get\_free\_pages()则直接返回这个page中的virtual域。当然这些映射的物理内存也可以释放然后供用户空间使用，有些内存（即所谓的高端内存）并不永久映射到内核地址空间，这时，这个域的值为NULL,需要的时候必须动态映射这些页。

内核向系统申请内存的一些接口个，内核分配页最核心的函数为

struct page\* alloc\_pages(gfp\_t gfp\_mask,unsigned int order);

该函数分配2的order次方的连续物理页，返回指向第一个页面的page结构体，否则返回NULL;

然后调用

void \* page\_address(struct page \*page)

返回此物理页的逻辑地址（虚拟地址）或者直接调用

unsigned long **\_\_get\_free\_pages**(gfp\_t gfp\_mask,unsigned int order)

直接返回请求的连续物理页的第一页逻辑地址，上述两个接口对于分配一页的函数如下

struct page \*alloc\_page(gfp\_mask)

unsigned long **\_\_get\_free\_page**(gfp\_t gfp\_mask)

分配一页清零的接口

unsigned long get\_zeroed\_page(unsigned int gfp\_mask)

进一步的当内核空间需要以字节为单位的分配物理内存时，内核的slab分配器提供了kmalloc接口

void \***kmalloc**(size\_t size,gfp\_t flags)

返回的内存块至少要有size大小，且分配的物理内存时物理上连续的

如上接口都有gfp\_mask标志，此标志分为三类

行为修饰符（分配时是否可以睡眠等），区修饰符（从哪个区分配），类型（前两个的组合）

常用行为修饰符:\_\_GFP\_WAIT(分配可睡眠) \_\_GFP\_HIGH（分配不可睡眠）\_\_GFP\_IO(分配可以启动磁盘IO) \_\_GFP\_FS(分配可以启动文件系统IO)

常用区修饰符:\_\_GFP\_DMA(从ZONE\_DMA分配) \_\_GFP\_HIGHMEM(从ZONE\_HIGHMEM区或者ZONE\_NORMAL区分配)

备注：\_**\_get\_free\_pages()或者kmalloc返回的都是逻辑地址,不能使用\_\_GFP\_HIGHMEM**

常用类型标志：GFP\_ATOMIC 分配不能睡眠 GFP\_KERNEL GFP\_USER 分配可以睡眠

vmalloc()接口介绍

void \*vmalloc(unsigned long size)

vmalloc只确保页在虚拟地址空间的连续区域中，它通过分配非连续的物理内存块，专门建立页表项，vmalloc获取的页必须一个一个的进行映射,可以睡眠

物理内存中高端内存部分的映射,高端内存一般不能永久的映射到内核的地址空间，因此通过alloc\_page以标志\_\_GFP\_HIGHMEM获取的高端内存物理页不可能有逻辑地址,映射一个给定page结构到内核地址空间，可以使用

void \*kmap(struct page\* page)

这个函数对于高端低端物理内存都可用，如果为低端内存中的一页，则单纯的直接返回系统初始化时对低端内存的映射的常规映射地址(逻辑地址)，如果为高端内存中的一页，则会建立一个永久映射，返回地址，可以睡眠，这个永久映射需手动解除

**进程调度**

内核用进程描述符task\_struct表示进程，进程描述符包含一个具体进程的所有信息，task\_struct相对较大，其包含的数据能完整的描述一个正在执行的程序：进程打开的文件，进程的地址空间，挂起的信号，进程的状态等

内核把进程的task\_struct用全局双向链表链接，这个链表叫做任务队列task list

task\_struct结构是创建进程时从slab分配器分配的，task\_struct存放在每个进程内核栈的尾端。作为thread\_info的task域。内核中访问进程通常需要获取进程的task\_struct结构体，可以直接通过current宏获取。

进程的状态存放在task\_struct中的state域，有如下五种：

TASK\_RUNNING(可运行状态)：进程正在执行或者在运行队列中等待执行

TASK\_INTERRUPTIBLE(可中断)：进程正在睡眠，等待某些条件达成，可被信号提前唤醒并随时准投入运行

TASK\_UNINTERRUPTIBLE(不可中断)：进程正在睡眠，等待某些条件达成，不可被信号唤醒

\_\_TASK\_TRACED

\_\_TASK\_STOPPED：进程停止运行，移出运行队列

内核设置进程的状态（内核空间的接口）：

set\_task\_state(task,state);//将task进程设置为state状态

set\_current\_state(state);//设置当前进程为state状态

linux系统提供抢占式多任务模式，此模式下由调度程序决定什么时候停止进程运行，让其他进程得到执行机会，这种又调度程序挂起进程的动作叫做抢占，调度程序事先已经设置好每个进程运行的时间（进程的时间片），时间片就是分配给每个可运行进程的处理器时间段，相应的非抢占多任务模式则是指除非进程主动停止运行，否则它会一直执行，进程主动挂起叫做让步。linux是抢占式的多任务模式

进程调度程序简称调度程序可以看做在**可运行态**TASK\_RUNNING**进程**之间分配有限处理器时间资源的内核子系统，调度程序是负责觉得哪个进程投入运行，何时运行，运行多长时间。

有了调度程序，系统就会自动在TASK\_RUNNING状态的进程中选择进程不断执行，可运行的进程都链接可执行红黑树队列中。

某一时刻，可能因为条件没达到等需让进程退出可执行红黑树队列，进入TASK\_INTERRUPTIBLE或者TASK\_UNINTERRUPTIBLE的队列，此时称进程休眠或者阻塞，内核实现此操作经过如下步骤：

1：进程把自己标记为休眠状态TASK\_INTERRUPTIBLE或者TASK\_UNINTERRUPTIBLE

2：从可执行的红黑树中移除

3：放入**等待队列**(队列中的进程处于TASK\_INTERRUPTIBLE或者TASK\_UNINTERRUPTIBLE状态)

4：调用schedule()选择和执行其他一个在可执行红黑树中的进程

详细介绍等待队列，内核用wake\_queue\_head\_t代表等待队列，等待队列可以通过DECLARE\_WAITQUEUE()静态创建，或者init\_waitqueue\_head()动态创建,等待队列由等待队列头开始，可以自己创建等待队列，也可以直接使用内核已经创建好的等待队列

**DEFINE\_WAIT(wait)**;//创建一个等待队列项，其中包含当前进程信息，稍后会将此项链接进某个等待队列中

**add\_wait\_queue(q,&wait)**;//将wait等待队列项链接进系统中的以等待队列头q开头的等待队列中

while(!codition){

**prepare\_to\_wait**(&q,&wait,TASK\_INTERRUPTIBLE);//将进程设置为TASK\_INTERRUPTIBLE状态，如wait不在q上，则让其链接在q上

if(signal\_pending(current)){

...//处理信号

}

**schedule()**;//调度一个可以运行的进程运行，**进程在此休眠**

}

finish\_wait(&q,&wait);//把进程移除等待队列

上述的接口被封装为**wait\_event()**接口。进程在schedule()处休眠，当其他的内核线程执行了**wake\_up(&q)**时，会唤醒，然后将等待队列头q中所有的等待队列项代表的进程都唤醒，设置为TASK\_RUNNING状态，移入可运行红黑树中，投入运行，这些进程在合适的时候执行，当某个进称判断while(!condition)满足后，则不再加入等待队列中而一直处于可运行进程的红黑树中，如果某个进程发现条件没满足，则执行prepare\_to\_wait(&q,&wait,TASK\_INTERRUPTIBLE)再次将进程设置为TASK\_INTERRUPTIBLE状态，并链接进等待队列中，等待下次wake\_up唤醒。

现在讨论进程切换的过程，从一个可执行进程切换到另外一个可执行进程，叫**上下文切换，**主要由schedule()接口调用context\_switch()来完成，这个函数主要完成如下两方面：

1：调用switch\_mm()，把虚拟内存从上一个进程切换到新的进程

2：调用switch\_to(),把处理器状态切换到新进程状态，保存上一个进程栈和cpu寄存器的信息

那内核何时调用schedule()来切换进程呢？除了主动调用schedule()来切换进程，内核还提供一个need\_resched的标志标明需要执行一次调度，也就是说当内核觉得某个进程应该被抢占或者更高优先级的进程进入可执行状态时，内核就会设置这个进程的这个标志（每个进的thread\_info都包含一个need\_resched域），内核检测到此标志设置后，就会调用schedule()来切换到一个新的可执行进程，内核什么时候检测这个标志呢，有如下几种情况：

1：当内核即将放回用户空间时（包含中断处理程序返回或者系统调用返回）,此时调度是安全的，内核会检测标志，若设置，则内核会选择一个其他的进程投入运行

2：当内核线程在运行是，只要内核检测到当前调度是安全的或者显示的调用schedule()，内核就可以抢占正在运行的内核线程