# Linux系统开发专栏





## linux内核分析笔记----进程管理

进程和线程的概念我就不讲了。总之，你记着：内核调度的对象是线程，而不是进程。linux系统中的线程很特别，它对线程和进程并不做特别区分。进程的另外一个名字叫任务(task).我和作者一样，习惯了把用户空间运行的程序叫做进程，把内核中运行的程序叫做任务。

内核把进程存放在叫做任务队列(task list)的双向循环链表中，链表中的每一项都是类型为task_struct,名称叫做进程描述符(process descriptor)的结构，该结构定义在include/linux/sched.h文件中，它包含了一个具体进程的所有信息。

linux通过slab分配器分配task_struct结构，这样能达到对象复用和缓存着色的目的。在2.6以前的内核中，各个进程的task_struct存放在它们内核栈的尾端。由于现在用slab分配器动态生成task_struct，所以只需在栈底或栈顶创建一个新的结构(struct thread_info),他在asm/thread_info.h中定义，需要的请具体参考。每个任务中的thread_info结构在它的内核栈中的尾端分配，结构中task域存放的是指向该任务实际task_struct指针。

在内核中，访问任务通常需要获得指向其task_struct指针。实际上，内核中大部分处理进程的代码都是通过task_struct进行的。通过current宏查找到当前正在执行的进程的进程描述符就显得尤为重要。在x86系统上，current把栈指针的后13个有效位屏蔽掉，用来计算thread_info的偏移，该操作通过current_thread_info函数完成，汇编代码如下：

```
movl $-8192, %eax
andl  %esp, %eax
```

最后，current再从thread_info的task域中提取并返回task_struct的值：current_thread_info()->task;

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中的一种，什么运行态啦，阻塞态啦，它们之间转化的条件啦等等，这一点我也不细说了，为啥？随便一本操作系统的书里，讲得都比我好，要讲就要讲别人讲不好的，是不？现在我关心的问题是：当内核需要调整某个进程的状态时，该怎么做？这时最好使用set_task_state(task, state)函数，该函数将指定的进程设置为指定的状态，必要的时候，它会设置内存屏蔽来强制其他处理器作重新排序。(一般只有在SMP系统中有此必要)否则，它等价于：task->state = state; 另外set_current_state(state)和set_task_state(current, state)含义是等价的。

一般程序在用户空间执行。当一个程序执行了系统调用或者触发了某个异常，它就陷入内核空间。系统调用和异常处理程序是对内核明确定义的接口，进程只有通过这些接口才能陷入内核执行----对内核的所有访问都必须通过这些接口。

linux进程之间存在一个明显的继承关系。所有的进程都是PID为1的init进程的后代，内核在系统启动的最后阶段启动init进程。该进程读取系统的初始化脚本并执行其他的相关程序，最终完成系统启动的整个过程。

系统中的每个进程必有一个父进程，每个进程也可以拥有一个或多个子进程。进程既然有父子之称，当然就有兄弟之意了。每个task_struct都包含一个指向其父进程task_struct且叫做parent的指针，同时包含一个称为children的子进程链表。所以访问父进程：struct task_struct *task = current->parent;按照如下方式访问子进程：

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children){
        task = list_entry(list, struct task_struct, sibling);
}
```

其中init进程描述符是作为init_task静态分配的。通过上面的init进程，父子进程关系，兄弟进程关系以及进程描述符的结构，我们可以得到一个惊人的事实：可以通过这种关系从系统的任何一个进程出发查找到任意指定的其他进程。而且方式还挺多的，这个就看书了，内容挺多我就不说了，只是最后需要指出的是，在一个拥有大量进程的系统中通过重复来遍历所有的进程是非常耗费时间的，因此，如果没有充足的理由千万别这样做。爱要一万个理由，这么做呢，没看出来.

许多的操作系统都提供了产生进程的机制，linux这优秀的系统也不例外。Unix很简单：首先fork()通过拷贝当前进程创建一个子进程。子父进程的区别仅仅在于PID,PPID和某些资源和统计量。然后exec()函数负责读取可执行文件并将其载入地址空间并执行。从上面分析可以看出，传统的fork()系统调用直接把所有的资源复制给心

创建的进程。这种方式过于简单但效率底下。在Linux下使用了一种叫做写时拷贝(copy-on-write)页实现。这种技术原理是：内存并不复制整个进程地址空间，而是让父进程和子进程共享同一拷贝，只有在需要写入的时候，数据才会被复制。不懂？简单点，就是资源的复制只是发生在需要写入的时候才进行，在此之前，都是以只读的方式共享。

linux通过clone()系统调用实现fork(),通过参数标志来说父子进程共享的资源。无论是fork(),还是vfork(),__clone()最后都根据各自需要的参数标志去调用clone().然后有clone()去调用do_fork().这样一说，我想大家明白我的意思了，问题的关键纠结于do_fork(),它定义在kernel/fork.c中，完成了大部分工作，该函数调用copy_process()函数，然后让进城开始运行，copy_precess()函数完成的工作很有意思：

1.调用dup_task_struct()为新进程创建一个内核栈，它的定义在kernel/fork.c文件中。该函数调用copy_process()函数。然后让进程开始运行。从函数的名字dup就可知，此时，子进程和父进程的描述符是完全相同的。
2.检查这个新创建的的子进程后，当前用户所拥有的进程数目没有超过给他分配的资源的限制。
3.现在，子进程开始使自己与父进程区别开来。进程描述符内的许多成员都要被清0或设为初始值。
4.接下来，子进程的状态被设置为TASK_UNINTERRUPTIBLE以保证它不会投入运行。
5.调用copy_flags()以更新task_struct的flags成员，表明进程是否拥有超级用户权限的PF_SUPERPRIV标志被清0。表明进程还没有调用exec函数的PF_FORKNOEXEC标志。
6.调用get_pid()为新进程获取一个有效的PID.
7.根据传递给clone()的参数标志，拷贝或共享打开的文件,文件系统信息,信号处理函数。进程地址空间和命名空间等。一般情况下，这些资源会被给定进程的所有线程共享；否则，这些资源对每个进程是不同的，因此被拷贝到这里.
8.让父进程和子进程平分剩余的时间片
9.最后，作扫尾工作并返回一个指向子进程的指针。

经过上面的操作，再回到do_fork()函数，如果copy_process()函数成功返回。新创建的子进程被唤醒并让其投入运行。内核有意选择子进程先运行。因为一般子进程都会马上调用exec()函数，这样可以避免写时拷贝的额外开销。如果父进程首先执行的话，有可能会开始向地址空间写入。

说完了fork，接下来说说他的兄弟---vfork(),兄弟就是兄弟，这像！两者功能相同，不同点在于vfork()不拷贝父进程的页表项。子进程作为父进程的一个单独的线程在它的地址空间里运行，父进程被阻塞，直到子进程退出或执行exec(),子进程不能向地址空间写入。按照刚才的方法，分析一下vfork(),它是通过向clone()系统调用传递一个特殊标志来进行的，过程如下：

1.在调用copy_process时，task_struct的vfor_done成员被设置为NULL
2.在执行do_fork()时，如果给定特别标志，则vfork_done会指向一个特殊地址。
3.子进程开始执行后，父进程不是马上恢复执行，而是一直等待，直到子进程通过vfork_done指针向它发送信号。
4.在调用mm_release()时，该函数用于进程退出内存地址空间，如果vfork_done不为空，会向父进程发送信号。
5.回到do_fork(),父进程醒来并返回。

上面步骤的顺利完成就意味着父子进程将会在各自的地址空间里运行。说句真的，通过研究发现这样的开销是降低了，但技术上不算咋优良。

如果说进程是80年代早上初升的太阳， 那不得不说的线程就是当前正午的烈日。线程机制提供了在同一程序内共享内存地址空间运行的一组线程。线程机制支持并发程序设计技术，可以共享打开的文件和其他资源。如果你的系统是多核心的，那多线程技术可保证系统的真正并行。然而，有一件令人奇怪的事情，在linux中，并没有线程这个概念，linux中所有的线程都当作进程来处理，换句话说就是在内核中并没有什么特殊的结构和算法来表示线程。那么，说了这多，到底在linux中啥是线程，我们说在linux中，线程仅仅是一个使用共享资源的进程。每个线程都拥有一个隶属于自己的task_struct.所以说线程本质上还是进程，只不过该进程可以和其他一些进程共享某些资源信息。

这样一说，后面就明白了也好解决了，两者既然属于同一类，那创建的方式也是一样的，但总要有不同啊，这个不同咋体现呢，这个好办，我们在调用clone()的时候传递一些参数标志来指明需要共享的资源就可以了：clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);这段代码产生的结果和调用fork()差不多，只是父子俩共享地址空间，文件系统资源，文件描述符和信号处理程序。换个说法就是这里的父进程和子进程都叫做线程。也就是说clone()的参数决定了clone的行为，具体有哪些参数，我是个懒人，也不想说了。

前边说的主要是用户级线程，现在我们接着来说说内核级线程。内核线程和用户级线程的区别在于内核线程没有独立的地址空间(实际上它的mm指针被设置为NULL).它也可以被调度也可以被抢占。内核线程也只能由其他内核线程创建。方法如下：int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags).新的任务也是通过像普通的clone()系统调用传递特定的flags参数而创建的。上面函数返回时，父进程退出，并返回一个子线程task_struct的指针。子进程开始运行fn指向的函数，arg是运行时需要用到的参数。一个特殊的clone标志CLONE_KERNEL定义了内核线程常用到参数标志：CLONE_FS, CLONE_FILES, CLONE_SIGHAND.大部分的内核线程把这个标志传递给它们的flags参数。

我虽有才，还是不如书上说的好啊，讲了那么多的创建，出生，突然来点终结的的话， 多少有点感伤啊。但感伤归感伤，进程终归是要终结的。一个进程终结时必须释放它所占用的资源并把这一消息告诉其父进程。进程终止的方式有很多种，进程的析构发生在它调用exit()之后，即可能显示地调用这个系统调用，也可能隐式地从

某个程序的主函数返回。当进程接受到它即不能处理也不能忽略的信号或异常时，它还可能被动地终结。但话说回来，不管进程怎么终结，该任务大部分都要靠do_exit()来完成，它定义在kernel/exit.c中，具体的工作如下所示：

1.将tast_struct中的标志成员设置为PF_EXITING.
2.如果BSD的进程记账功能是开启的，要调用acct_process来输出记账信息。
3.调用__exit_mm()函数放弃进程占用的mm_struct,如果没有别的进程使用它们即没被共享，就彻底释放它们。
4.调用sem_exit()函数。如果进程排队等候IPC信号，它则离开队列。
5.调用__exit_files(), __exit_fs(), __exit_namespace()和exit_sighand()以分别递减文件描述符，文件系统数据，进程名字空间和信号处理函数的引用计数。当引用计数的值为0时，就代表没有进程在使用这些资源，此时就释放。
6.把存放在task_struct的exit_code成员中的任务退出代码置为exit()提供的代码中，或者去完成任何其他由内核机制制定的退出动作。
7.调用exit_notify()向父进程发送信号，将子进程的父进程重新设置为线程组中的其他线程或init进程，并把进程状态设为TASK_ZOMBIE.
8.最后，调用schedule()切换到其他进程。

经过上面的步骤，与进程相关的资源都被释放掉了，它以不能够再运行且处于TASK_ZOMBLE状态。现在它占用的所有资源就是保存threadk_info的内核栈和保存tast_struct结构的那一小片slab。此时进程存在的唯一目的就是向它的父进程提供信息。

僵死的进程是不能再运行的。但系统仍然保留它的进程描述符，这样就有办法在子进程终结时仍可以获得它的信息。在父进程获得已终结的子进程的信息后，子进程的task_struct结构才被释放。

熟悉linux系统中子进程相关知识的我们都知道在linux中有一系列wait()函数，这些函数都是基于系统调用wait4()实现的。它的动作就是挂起调用它的进程直到其中的一个子进程退出，此时函数会返回该退出子进程的PID.调用该函数时提供的指针会包含子函数退出时的退出代码。最终释放进程描述符时，会调用release_task(),完成的工作如下：

1.调用free_uid()来减少该进程拥有者的进程使用计数。
2.调用unhash_process()从pidhash上删除该进程，同时也要从task_list中删除该进程。
3.如果这个进程正在被ptrace追踪，将追踪进程的父进程重设为其最初的父进程并将它从ptrace_list上删除。
4.最后，调用put_task_struct释放进程内核栈和thread_info结构所占的页，并释放task_struct所占的slab高速缓存.

至此，进程描述符和所有进程独享的资源就全部释放掉了。

最后，我们讨论进程相关的最后一个问题:前边的一切看似很完美，很美好，美好让人还怕，不是么？哪里出问题了，父进程创建子进程，然后子进程退出处释放占用的资源并告诉父进程自己的PID以及退出状态。问题就出在这里，子进程一定能保证在父进程前边退出么，这是没办法保证的，所以必须要有机制来保证子进程在这种情况下能找到一个新的父进程。否则的话，这些成为孤儿的进程就会在退出时永远处于僵死状态，白白的耗费内存。解决这个问题的办法，就是给子进程在当前线程组内找一个线程作为父亲，如果这样也不行(运气太背了，不是)。在do_exit()会调用notify_present(),该函数会通过forget_original_parent来执行寻父过程，具体我就不讲了，讲到这个详细的地步，还不自己看看，我没办法了.

一旦系统给进程成功地找到和设置了新的父进程，就不会再有出现驻留僵死进程的危险了，init进程会例行调用wait()来等待子进程，清除所有与其相关的僵死进程。

posted on 2011-07-09 21:18 ☆&寒 烟☆ 阅读(4897) 评论(...) 编辑 收藏