linux线程与进程的理解

原创芹泽 最后发布于2018-08-21 18:05:03 阅读数 1994 收藏

展开

首先明确一点，linux对进程和线程不做区分，统一由task\_struct来管理所有进程和线程。

那么如何在linux下区分进程和线程呢？

为什么要引入线程的概念？

一个进程包含很多系统资源：进程控制块、虚存空间、文件系统，文件I/O、信号处理函数，创建一个进程的过程就是这些资源被创建的过程。

系统调用fork创建一个进程时子进程是一段独立的内存空间，其中的资源是父进程资源的副本，两个进程是完全独立不共享内存资源的，二者需要通过IPC进行通信。

这样的做法在有些场景下并不是高效的做法，例如：比如某进程fork出一个子进程后，其子进程仅仅是为了调用exec执行另一个执行文件，那么在fork过程中对于内存空间的复制就是多余的；再例如：如果通过这种方式处理并行计算问题，那么就得在不同cpu创建不同的进程，然后通过IPC再把计算结果汇总，这样做的开销往往足以抵消并行计算带来的好处。

另外，进程是系统中程序执行和资源分配的基本单元，每个进程都拥有自己的数据段、代码段和堆栈段，进程进行切换时都会伴随着上下文的切换，这也会带来开销。

所以把计算单元抽象到进程上是不充分的，这也就是许多系统中都引入了线程的概念的原因。

Linux怎么创建进程？

Linux创建进程一共有三种方式：fork  vfork  clone。三个函数分别通过sys\_fork()、sys\_vfork()和sys\_clone()调用do\_fork()去做具体的创建工作，只不过传入的参数不同。

sys\_fork()：

asmlinkage long sys\_fork(struct pt\_regs regs)

{

return do\_fork(SIGCHLD, regs.rsp, &regs, 0);

}

传入的参数SIGCHLD表示在子进程终止后将发送信号SIGCHLD信号通知父进程

sys\_vfork()：

asmlinkage long sys\_vfork(struct pt\_regs regs)

{

    return do\_fork(CLONE\_VFORK | CLONE\_VM | SIGCHLD, regs.rsp, &regs, 0);

}

sys\_clone()：

casmlinkage int sys\_clone(struct pt\_regs regs)

{

    unsigned long clone\_flags;

    unsigned long newsp;

    clone\_flags = regs.ebx;

    newsp = regs.ecx;

    if (!newsp)

        newsp = regs.esp;

    return do\_fork(clone\_flags, newsp, &regs, 0);

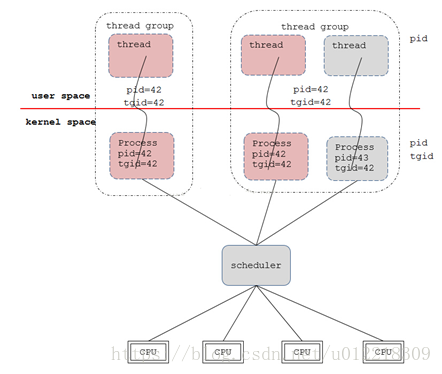
}

Linux怎么创建线程？

为了方便移植，linux使用POSIX标准创建线程，在linux2.6之前使用linux Thread库实现多线程，从linux2.6开始，都是使用NPLT库来实现线程的创建。这两个库都是使用pthread\_create()来创建线程。

抛开网上一大堆所谓的轻量级进程LWP，那些都过时了，都是2.6版本以前提出的概念，从2.6开始的多线程不需要理解这个LWP，只需要知道如果使用clone创建的进程与父进程共享内存空间则认为它是线程。

NPLT线程的创建 - - 从用户态到内核态



上图中，红色线以上是用户态，下面是内核态。左边虚线框内是单线程的，右边是多线程。

NPLT创建的线程属于“一对一”模型，为什么叫一对一？因为调用pthread\_create()时会在用户空间创建一个线程同时也会在内核空间创建与之对应的线程。

用户空间的线程 - - 负责执行线程的创建、销毁等操作

内核空间的线程 - - 负责调度

1 - pthread\_create

pthread\_create是创建线程的入口，主要参数是一个函数指针，其指向的函数为线程创建成功后要执行的函数。由于线程组中的各个线程是可以并行运行在不同的CPU上的，所以各个线程必须得有自己独立的用户态栈和内核栈。Linux用户态栈的大小一般是8M，通过mmap在MemoryMapping Area中分配一块内存（不考虑用户态栈缓存）。在用户态也有一个数据结构用来描述线程（structpthread），该数据结构就放在用户态栈的最下面的位置，其中会存放线程创建成功后要执行的函数地址。

分配完structpthread和用户态栈之后（其实还有很多事情，不过都是些琐事），差不多就可以带着这些信息进入内核申请task\_struct了。

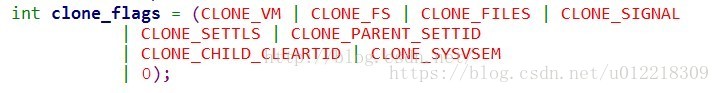
2 - clone

像创建新的进程这种资源管理工作基本上都要通过内核完成，从用户态进入内核态时，都会把进程在用户态时的状态保存在内核栈中，这样完成了内核态的任务之后返回时可以恢复用户态的状态。

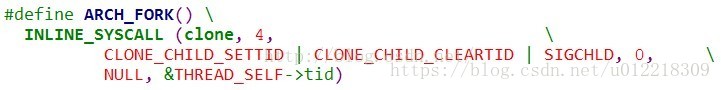
glibc在用户态对clone封装了一层，名为ARCH\_CLONE。其中会把线程创建成功后要执行的函数地址以及参数压入用户态栈，然后再调用系统调用clone，这样系统调用clone返回后再从栈中取出线程函数以及对应的参数，继而开始执行线程函数。

系统调用（syscall）clone用于创建当前进程的一个副本，fork就是调用的clone。我们这里是要创建线程，那么对clone的用法当然与fork有所区别了。区别主要在于clone的参数clone\_flags的设置：

创建线程调用的clone：

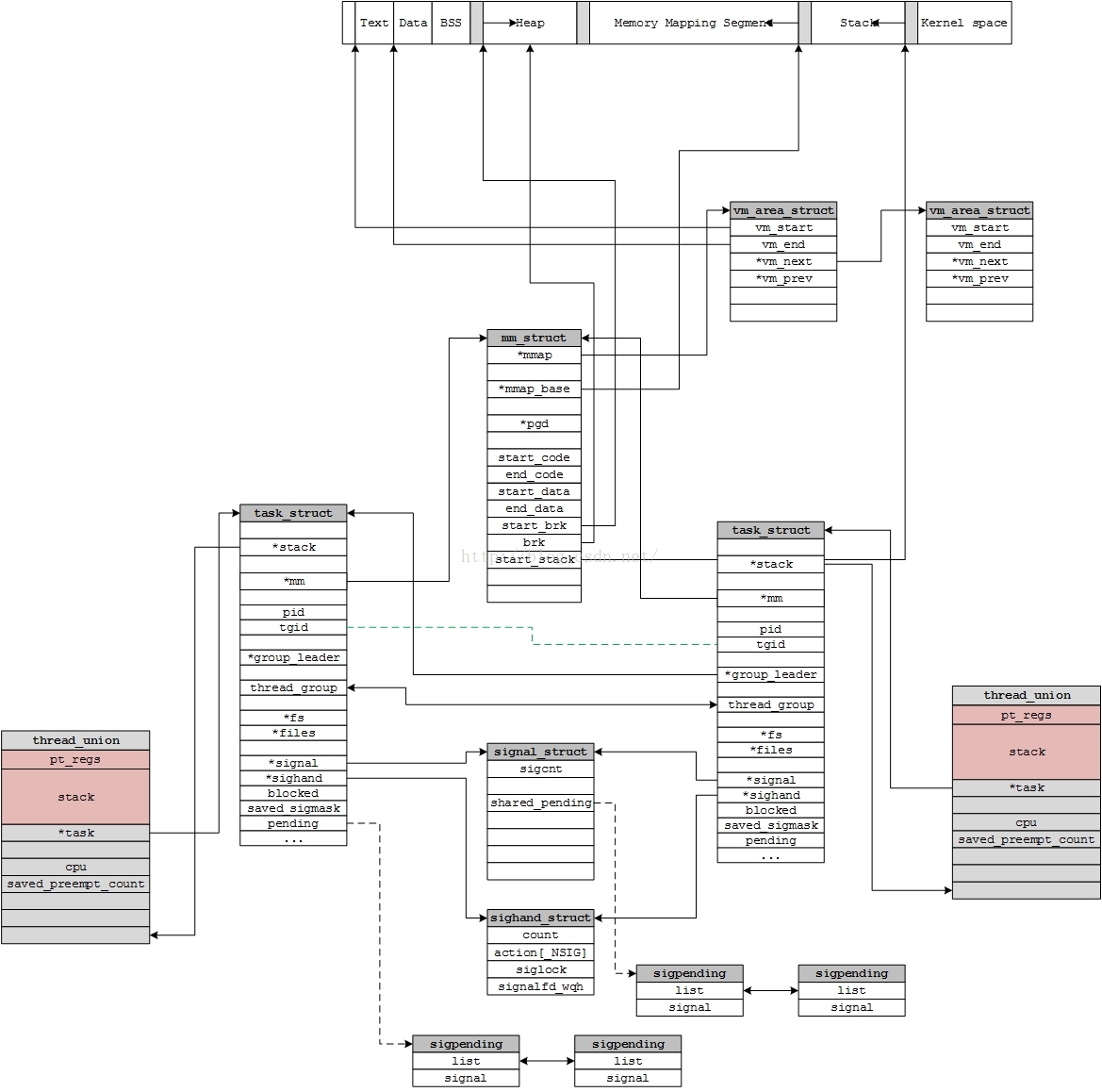


创建进程调用的clone:



可以看到创建线程时使用的clone\_flags中多了CLONE\_VM、CLONE\_FS、CLONE\_FILES和CLONE\_SIGNAL（CLONE\_SIGHAND| CLONE\_THREAD），这些是实现Posixthread规范的关键。这些参数体现在NPLT线程在内核中的数据模型图中就是其中的mm、fs、files、signal、sighand和pending字段指向相同的对象。正因为这些数据的共享，线程间的数据共享和同步比在进程间简单得多。

NPLT线程在内核中的数据模型图



不管是通过fork产生的进程还是通过pthread\_create产生的线程，其在内核中都对应着一个task\_struct，linux\_structure图中有两个task\_struct，其中右边的task\_struct是通过pthread\_create产生的，因此左边的task\_struct是threadgroup的leader。task\_struct有很多字段，其中mm就是用来描述虚拟地址空间的。我们可以看到两个task\_struct的mm指向了同一个mm\_struct对象，也就是前面提到的线程间共享同一个虚拟地址空间。

用户态中，多线程会作为一个进程看待，在内核中，会被抽象为“线程组”。线程组中的task\_struct有不同的pid字段，但是会有相同的tgid(threadgroup id)字段，这个tgid作为用户态进程的pid给上层使用。所以，在一个进程下的每个线程中获取到的pid是相同的，用户态的pid与内核态的pid不是同一个东西，想获取线程在内核态的pid可以通过系统调用（gettid）来获取。线程组中的task\_struct会通过一个链表（thread\_group）链接起来，后面创建的线程的task\_struct中的group\_leader字段会指向leader。通过共享mm\_struct、fs、signal相关的数据，再通过thread\_group相关的设置，线程的概念基本就实现了

下一篇，我们根据这篇的理论基础来总结一下pthread\_create的使用

————————————————

版权声明：本文为CSDN博主「芹泽」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/u012218309/article/details/81912074