对container原理的框架性理解

我对container原理的一些理解(基于linux kernel 2.6.38)

by kin

2011.04.17

==========

linux中称谓的container在内核层面由两个独立的机制保证，一个保证资源的隔离性，名为namespace；一个进行资源的控制，名为cgroup。

1 namespace

inux现有的namespace有6种：**uts, pid, ipc, mnt, net和user**。所有的namespace都和task_struct相关，其中uts, pid, ipc, mnt和net都处于task_struct->ns_proxy中，而user_namespace却是由task_struct相关联的user_struct指向决定的 。

1.1 uts

最简单的主机名，一个namespace结构绑定这样一个字符串，uname的时候去current->nsproxy->uts_namespace下面取字符串就好了

1.2 ipc

ipc维护的是一个id到struct的映射关系，这种映射关系在内核由公用设施idr提供。所谓ipc的namespace就是每个namespace都有自己独立的idr,即独立的key->value的映射集合，不同的

导航

博客园
首页
新随笔
联系
订阅XML
管理

公告

昵称：FengK
园龄：2年3个月
粉丝：1
关注：0
+加关注

< 2013年1月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

namespace通过key找value会在不同的idr中寻找，内核数据结构是公共可见的，隔离的仅仅是索引关系。

1.3 mnt

每个mnt_namespace有自己独立的vfsmount *root, 即根挂载点是互相独立的，同时由vfsmount->mnt_child串接起来的子mnt链表，以及继续往下都是彼此独立的，产生的外在效果就是某个mnt_namespace中的mount, umount不会 对其他namespace产生影响，因为整个mount树是每个namespace各有一份，彼此间无干扰, path lookup也在各自的mount树中进行。这里和chroot之类的又不一样，chroot改变的只是 task_struct相关的fs_struct中的root，影响的是path lookup的起始点，对整个mount树并无关系.

1.4 user

user_namespace的实现方式和ipc类似，每个namespace各自维护一个uid到user_struct的映射，用hash表实现。但是uid会在两个地方体现，一个是user_struct中的uid，还有一个是cred中的uid。user_namespace影响范围局限在user_struct中，虽然clone(NEWUSER)时会把task_struct的cred中的uid,gid都设成0,然后这种关系又可以通过fork等传递下去，但是终究user_namespace并没有影响到cred里面数据，而且vfs里面inode是只有uid的，

不会有user_struct信息，因此某个具体的文件其uid是固定的，具体在某个namespace中如何显示用户名则不关内核层的事了，由/etc/passwd中定义的映射关系决定。

另外还有个问题，比如跨两个namespace的unix套接字通信，有个选项叫PEERCRED，是拿对方节点的ucred结构，因为不同namespace，因此拿到的uid,gid都要进行user_namespace的重映射。这里重映射的策略就是：

1) 同user_namespace，OK。不需要

2) 不同，则由于每个user_namespace都会记录创建自己的那个user_struct，因此一层层往上索引到init_user_ns，如果发现需要remap的那个user_struct是我们的祖先创建者，则map为0，否则则返回一个表示不存在的MAGIC NUMBER

1.5 pid

pid_namespace是每个namespace有一个单独的pid_map作为bitmap进行pid的分配，因此各个pid namespace的pid互不干扰，独立分配。同一个task_struct会从init_ns开始，到最终它所在的namespace，每一层都会有个单独的pid（也就是深层次的task_struct创建，在每一个层次的namespace都会进行pid的alloc），而所有这些pid信息都是栈式保存在struct pid结构中。

pid是唯一一个底层namespace会把上层namespace信息都保留下来的namespace, pid信息保存在struct pid中，而具体的(pid, ns)信息对则保存在upid中，pid会根据自己的namespace深度扩展一个upid的栈，在这个pid结构中，该task_struct从init_ns到实际所处的namespace整个树路径上的(pid,ns)信息都记录了，于是上面所说的跨namesapce unix socket通信取PEERCRED对pid的remap就很简单了，有父子关系，直接从pid的不同深度取另一个值就行了；如果没父子关系，则MAGIC NUMBER。

同时pid的upid栈中每个upid都会连入对应pid namespace的hash表中，那么该task_struct在每个namespace层次中都是可见的（可在ns对应hash表中找到），且pid号互不相关（看到的是对应栈层次上的upid->nr）。

由于历史因素,task_struct中除了用pid索引pid, ppid, pgid,sid，结构体本身还有pid,tgid等，这里的数据都是 取的init_ns中的对应数值.

1.6 net

net_ns是非常复杂的一块。mainline都做了几个版本来稳定它，主要是各个关键数据结构都加上了net信息，比如sock, 路由查找的fib， netfilter的rules,net_device等， net对于不同net的数据隔离和前面几种每个namespace自己建索引表不同，net的一般都在同一个索引表中，只是数据多加一维net信息，在查询时多加上对这一维的比较。相应的网络层关键处理函数都会多带一个net_namespace的参数. 在net_namespace结构体内部，主要保存了网络各层关键的sysctl信息，用于实现对不同net namespace可以进行不同的内核参数配置，以及不同的proc entry导出。

1.7 task_struct在不同ns之间的转移

最新的mainline kernel: 2.6.39rc3是没有实现这样的一个系统调用的。lxc内有个lxc-attach依赖一个setns的系统调用，这个系统调用原理就是通过你提供的/proc/pid/ns这样一个proc inode，来找到对应的namespace结构体，然后把current->nsproxy->xx_namespace设为那个，可能还要进行些额外的操作（比如pid的remap，user的remap之类的）就OK了。但是namespace本身就是提供的安全性隔离，这样做有安全风险，所以mainline kernel并没有merge这个patch。

1.8 如何创建新的namespace

创建一个新的进程，man 2 clone

当前进程，man 2 unshare

2. cgroup

cgroup是通过vfs的接口来进行配置的，它本身充当一个resource controller的角色，对一组进程进行资源控制，核心角色便是一组task_struct。

2.1 cgroup的几个核心概念

cgroup核心的有三个概念：

hierarchy: 就是mount的一个cgroup fs。

subsystem: 一个subsystem对一种资源进行control

cgroup: 一个hierarchy下面对进程的分组

整体的逻辑可以这样来看：

1) 一个hierarchy和多个subsystem绑定，该hierarchy只对这几个subsystem对应的resouce进行control

2）不同的hierarchy绑定的subsystem是互斥的

3）一个hierarchy下面可以通过mkdir方式创建不同的cgroup，这个cgroup下面可以attach一组进程，通过该cgroup指定的参数对这种进程资源使用进 行track和控制

这样分可以使task A和task B在hierarchy A中在一个cgroup中控制CPU, 而task A和task B在hierarchy B中分别在两个cgroup中控制memory，提供 更灵活的策略

2.2 cgroup在内核层次和task_struct的交互

一个cgroup和一个subsystem决定了一个cgroup_subsys_state, 每个进程都以css_set（一个静态数组）的方式保存了所有和它有关的cgroup_subsys_state,同样，每个cgroup也会保存这样一组。 cgroup_subsys_state又会作为每个subsystem自己结构体的内部成员包含，这样通过container_of很容易就可以找到subsystem自己的控制结构体，来进行各子系统自己的控制，于是每个task_struct都可以通过css_set找到它所属的subsystem结构体，来进行后面的操作.

而对于每个css_set, 也会把所有使用它的task连起来，这样cgroup导出所有当前绑定进程也有了依据(每个Cgroup链接一组css_set，每个css_set又会串联一组task_struct，挨个遍历)

2.3 cgroup 当前已有的subsys

kinwin@ustc-king:/data/linux-2.6$ grep "SUBSYS" include/linux/cgroup_subsys.h

/* Add subsystem definitions of the form SUBSYS(<name>) in this

SUBSYS(cpuset)

SUBSYS(debug)

SUBSYS(ns)

SUBSYS(cpu_cgroup)

SUBSYS(cpuacct)

SUBSYS(mem_cgroup)

SUBSYS(devices)

SUBSYS(freezer)

SUBSYS(net_cls)

SUBSYS(blkio)

SUBSYS(perf)

其中ns_cgroup已经被抛弃，通过cgroup.clone_children和手动attach进程来取代

2.4 cpu_cgroup

对CPU的使用率进行控制，根本上利用了CFS中已有的task group概念，最近出来的autogroup也是 利用了这样一个机制。内核中的调度针对的是sched_entity这样一个结构，sched_entity本身有一个run queue，同时它也会在别人的run queue上，以此层层嵌套下来，直至最后的可执行单元（task_struct）。于是一个cgroup的cpu resource controller

便是这样的一个sched_entity，所有attach到这个Cgroup的进程都在这个sched_entity下面，调度也都在这个sched_entity下面进行。

这样的一个树形架构下，每个sched_entity只负责进行在该sched_entity runquueu上的sched_entity进行CFS，即从红黑树最左端pick一个sched_entity，由该sched_entity进行下一个层次的CFS，而每次最终pick出来的task_struct的运行对所有它的父sched_entity的运行

时间都有贡献，如此实现一个全局的CFS。并且实现task group or task group's group...

2.5 mem_cgroup

memory有全局的per zone lru list来进行page relcaim等，有全局的page数组来进行物理内存管理. memory_cgroup在cgroup层面上实现了per_cgroup lru list, 以及page_cgroup数组，memory_cgroup的lru操作及其他基于page的操作都是以page_cgroup为依据的，而page_cgroup和真正的page之间又会有个映射关系

mem_cgroup对于内存资源的控制，是对每个cgroup的使用内存情况进行记录，主要方式就是在真正分配物理页面的地方（缺页中断，或者分配page cache页等）都进行了hack(可以统计哪些页面取决于在哪些内存分配部分进行了分配物理页的hack， 目前应该是绝大部分都有进行hack)，通过current->cgroups->subsys[xx]找到对应的mem_cgroup控制结构，

判断可否分配，可以就加上自己的计数，不行或达到某个limit，就会触发基于per-cgroup lru的reclaim, 再或者就触发cgoup内部的OOM killer等。

但是内存作为一个最复杂的部分，mem_cgroup的发展也是今年Linux filesystem, storage, and memory management summit讨论最多的话题，

有以下几点

1）重复的lru list，全局内存紧张依然会page reclaim，不分cgroup。而且两个lru list两次lru也重复了

2) page_cgroup结构体可以变的没有，现在20 bytes(32bit)/40bytes(64 bit)太大了

3) 全局和单个cgroup的权衡

…

2.6 net_cls

主要利用了内核网络协议栈traffic control的相关东西，实现了一个cgroup的统一标记id，然后实现了一个叫cgoup的filter，这个filter就是根据当前进程所在的cgroup_subsys决定给sk_buff打上何样的id标记，然后这个标记就会被用于匹配相应的traffic control的qdisc和class来进行具体的流量控制策略。

2.7 device

在inode_permission中hook入cgroup的检查，对于inode->st_type为char或block类型的，会与保存在列表中的进行读，写权限匹配，根据匹配结果决定

cgoup的inode检查返回允许否？

在vfs_mknod hook入cgroup对mknod主从设备号的匹配检查，决定允许否?

2.8 freezer

给cgoup中每个task_struct设置下TIF_FREEZING，然后就开始try_to_freeze, 设置个PF_FROZON的flag，进程就开始空转了

```
for (;;) {
set_current_state(TASK_UNINTERRUPTIBLE);
if (!frozen(current))
break;
schedule();
}
```

而唤醒如上代码所示，就是去掉PF_FROZON这个flag的过程。

2.9 cpuset

同sched_setaffinity，但是是对于一组进程设置CPU亲和性了。内核在CPU亲和性逻辑跟以前没什么区别，无非是把这些进程只调度到对应CPU的run queue而已。

同时cpuset还提供了一个只在对应cpu间进行负载均衡的特性，就是把对应的cpu作为一个sched doamin，可以在其中负载均衡。不过要求最好各个cgoup设置互斥的cpu，否则就会取cgroup的最大互斥并集作为sched domain，这样跨cgroup的load banlance又会导致其他的复杂因素。

3. lxc

lxc是一个用户空间的管理工具，提供用户友好的接口包装了内核提供的namespace和cgroup机制.

它主要实现原理是这样的：

0) 首先准备创建好网络设备(netlink创建设备)，

- openpty指定数目的pty(创建对应的/dev/ptmx和/dev/pts/xx对),作为lxc_console所用，slave会被mount bind到container的/dev/ttyx,通过master fd可和container中

起来的shell通信

并open一个/dev/tty作为console所用，这个console设备也会被mount bind到container中，这个console fd就是用于lxc_start后获得的那个console与container的通信

1）*开始clone,创建新的namespace*:父进程clone一个子进程，clone的时候指定新建所有的namespace，如此一个完全新的namespace就建立了。

子进程：

2) *父子同步*：子进程设置自己死了给父进程发sigkill，防止父进程空等， 同时也保证container中的init死掉同时会导致host中的lxc_start死掉.

父进程：

3) *建立对应的cgroup:*父进程设置clone_children，然后创建新的cgroup，并将clone的pid attach到新创建 cgoup中。(等同于以前ns_cgroup所做的工作)

4) *挪移网络设备:*将配置在container中的网络设备通过发送netlink消息,带上IFLA_NET_NS_PID的rtattr,触发内核的dev_change_net_namespace，将net_device的namespace替换掉，

具体网络设备的change namespace涉及到在一个namespace的完全停止，解注册和在另一个namespace中的注册等流程。

子进程：

5）*设置container utsname*

6）*设置container网络设备参数*：通过netlink消息给本net namespace 网络设备设置ip addr, hw addr, flags等信息

7）*修改cgroup中resouce control参数*：由于还没有chroot，可以修改自己cgroup中相应的resource control设置

8) *建立根文件系统并建立自己的挂载：*mount rootfs, mount fstab等等

8）*建立host和container在终端方面的通讯渠道*：

把/dev/tty mount bind到container中，这样我们就可以:

pty slave <----> pty master <---->*epoll两端转发* console fd 来获得刚开始的一个Console

把前面创建的pty的slave mount bind到container中的/dev/ttyx， 这样我们就可以通过对应的pty master fd实现和container中的通信:

stdin,stdou *epoll两端转发* <--------> pty master <------> mount bind到container中的/dev/ttyx

9) *改变根挂载点*： chroot并把从host中继承过来的mount关系在container中不会用到的

umount掉及其他一些保证系统正常运转要干的其他工作

10) *exec开始init:*container 1号进程exec创建自己的进程树

父进程，子进程同时运转

11) 父*epoll循环*：父进程醒来 ，开始一个epoll循环，主要处理console和container的两端转发epoll, 以及接收一些外来查询请求的unix套接口epoll

比如查询container init在host中进程pid，这样我们可以kill -SIGKILL pid, 就可以杀掉整个container

通过unix套接字的msgcontrol传递pty的master fd(也就是被mount bind到container中/dev/ttyx的peer fd)，用来实现lxc_console取container tty的作用。

unix套接字绑定在一个已知的路径下,在lxc 0.7.4中通过这个unix套接口实现有取container init pid，取tty fd，取container state及停止container的作用。

而基本所有的lxc小工具都是基于这个unix套接字来获取一些关于container的信息。

12) *子独立进程树下的运转：*在一个单独的namespace里面，有自己独立的resource control，单独的基础设施和用户空间进程树，一个隔离并资源控制的新环境，谓之container。便

运转起来了。

以上所述皆基于个人查阅代码和文档的理解所述， 有什么不当或错漏之处，还请各位多多指正， thx

标签: lxc, cgroup, container

FengK
关注 - 0
粉丝 - 1
+加关注

0 0

(请您对文章做出评价)

« 上一篇：如何在内核中禁止route cache

posted on 2013-01-14 11:13 FengK 阅读(700) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

最新知识库文章:

· 浅析数据化设计思维
· 门户级UGC系统的技术进化路线
· 亿级用户下的新浪微博平台架构
· 技术团队的情绪与效率
· 关于请求被挂起页面加载缓慢问题的追查
» 更多知识库文章...