PDF-XCHANGE Click to buy NOW! www.docu-track.com

操作系统虚拟化底层基础之命名空间（namespace）

黎润(yijunzhu@qq.com)

目录

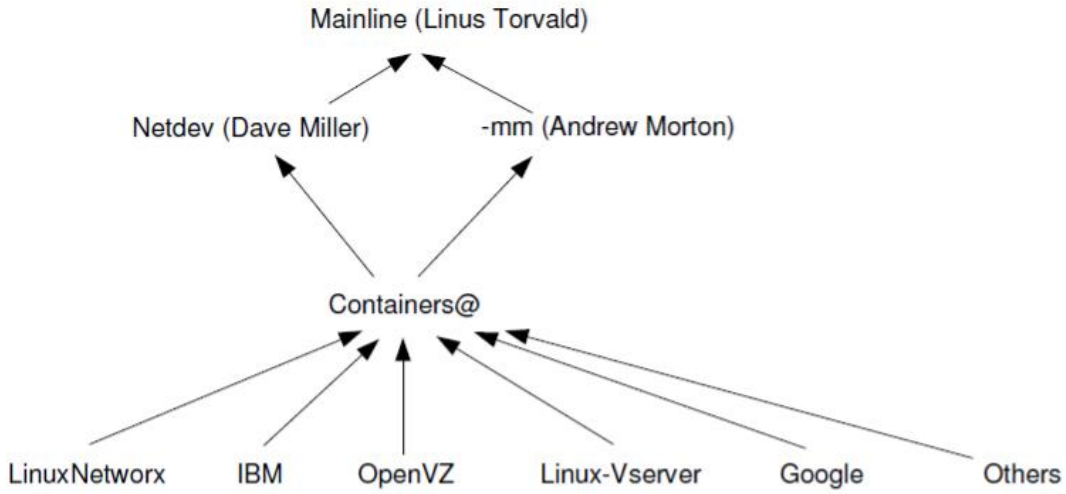
PDF-XCHANGE Click to buy NOW! www.docu-track.com

背景

随着公司业务的迅猛发展，大量的机器在线上业务号召下投入了服务于广大网民的神圣职责。不过基于一个不完全统计，我们公司的线上机器平均利用率 20%左右，这就意味着 70%左右的机器都是可回收或者复用的。

出于节约机器，统一管理以及在线迁移的初衷，我们进行了虚拟化计算的研究。经过选型测试以及具体应用场景的研究，我们选择了操作系统虚拟化技术，即 LXC。（为什么选择 LXC，OpenVZ 如何？ Xen 效果如何等等这些问题请参考其他文档，本文主要讨论 LXC 的底层实现技术）。LXC 本身不是一个具体的技术，它是一个集合技术的代称，我们可以总体上来看，LXC 主要有 namespace 和 cgroup 两大模块构建而成，本系列主要就是说说这两个技术，本文则专注于 namespace。

在我们讲述具体的技术之前，先来看看容器模块的整个状态系统，目前主要是 IBM，google 等公司的团队在负责维护更新。

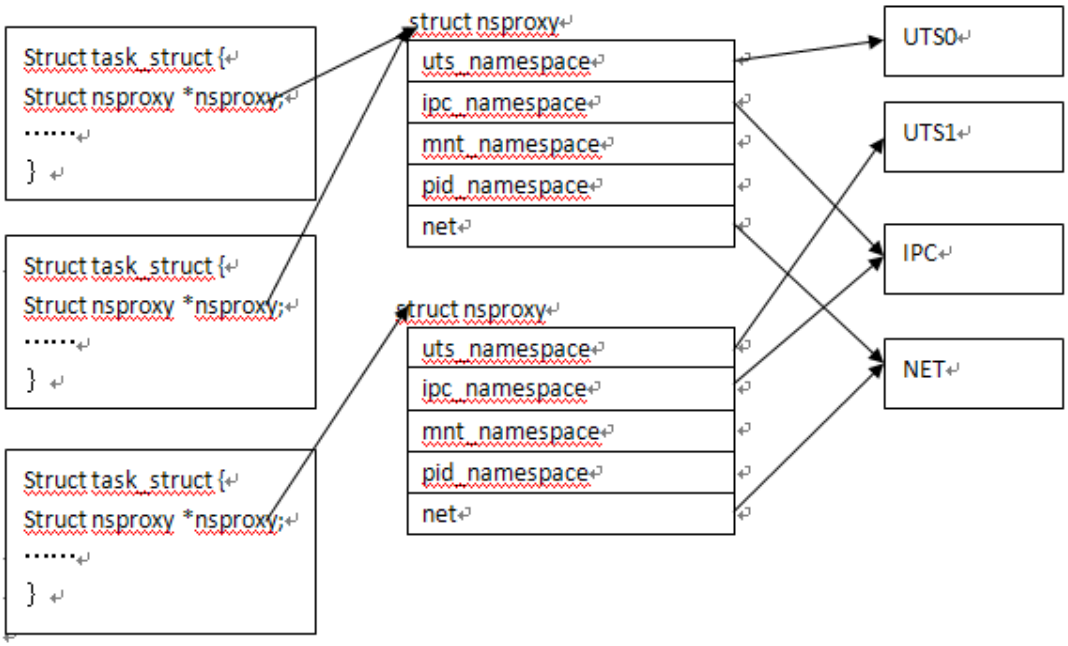


目前 container 已经被上有内核所接纳，所以不存在自己维护分支版本的问题。但是这些团队之间合作不是我们想象的和谐，不同利益集团之间是有内核的政治诉求，都想把自家的内容扶位正房，导致我们再看操作系统虚拟化的时候会有不同项目博弈的事迹。

总览

每一个进程其所包含的命名空间都被抽象层一个 nsproxy 指针，共享同一个命名空间的

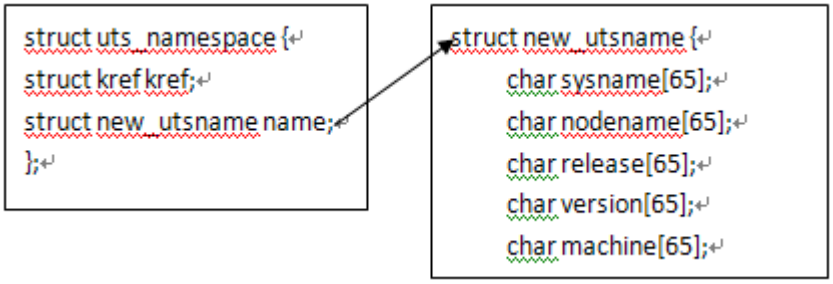
进程指向同一个指针，指针的结构通过引用计数（count）来确定使用者数目。当一个进程其所处的用户空间发生变化的时候就发生分裂。通过复制一份老的命名空间数据结构，然后做一些简单的修改，接着赋值给相应的进程。



看了上面的数据结构，我们就会基本明白，命名空间本身只是一个框架，需要其他实行虚拟化的子系统实现自己的命名空间。这些子系统的对象就不再是全局维护的一份结构了，而是和进程的用户空间数目一致，每一个命名空间都会有对象的一个具体实例。目前 Linux 系统实现的命名空间子系统主要有 UTS、IPC、MNT、PID 以及 NET 网络子模块。我们在下文会针对这些子模块进行进一步的分析。

UTS 命名空间子模块

UTS 相对而言是一个简单的扁平化命名空间子模块，其不同的命名空间之间没有层次关系。我们先来看一下 UTS 的数据结构。



New_utename 结构里面就是我们通过 uname –a 能够看到的信息。看一下机器上的输出：

```
Linux/136_8/2.6.30-TENCENT64-100310/#5 SMP Mon Jul 19 17:20:11 CST 2010/x86_64 x86_64
```

我通过红色斜线把 uname –a 的输出分隔开，分别对应上面的 new_utsname 的结构体。另外内核还把这些信息也通过 proc 文件系统导出，我们可以通过/proc/sys/kernel 目录里面的如下等变量（Ostype/ hostname/osrelease/ version）查看,当然这些变量的值也是可以更改的。

初始的时候，系统默认构造了一个 UTS 结构，他的值分别如下所述。

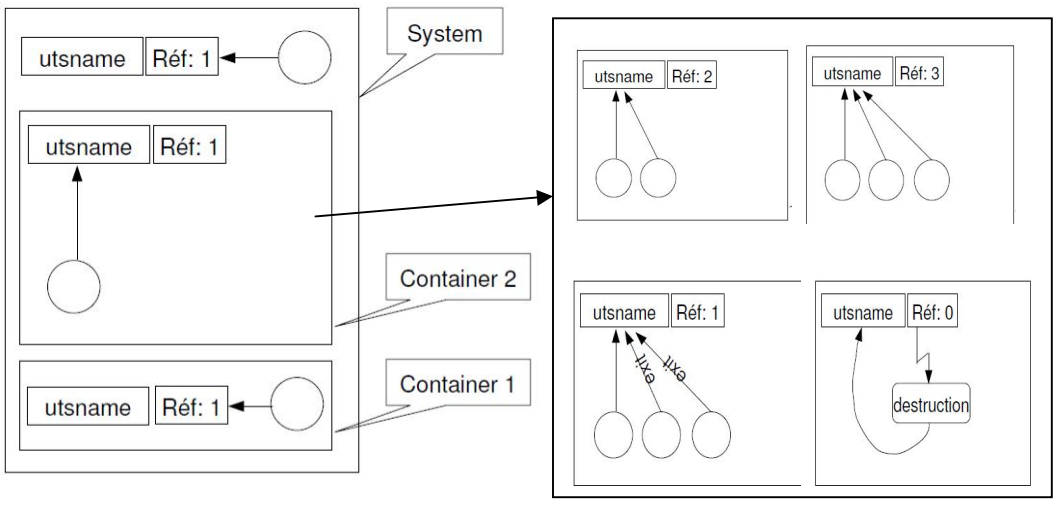
```
struct uts_namespace init_uts_ns = {
        .kref = {
                .refcount = ATOMIC_INIT(2),
        },
        .name = {
                .sysname = UTS_SYSNAME,
                .nodename      = UTS_NODENAME,
                .release   = UTS_RELEASE,
                .version   = UTS_VERSION,
                .machine = UTS_MACHINE,
                .domainname  = UTS_DOMAINNAME,
        },
};
```

当一个新的命名空间创建的时候，copy_utsname 会被调用来创建一个 UTS 的命名空间，主要工作在 clone_uts_ns 函数里面完成。

```
struct uts_namespace *copy_utsname(unsigned long flags, struct
uts_namespace *old_ns)
 {
   new_ns = clone_uts_ns(old_ns);
   ......
 }
```

```
static struct uts_namespace *clone_uts_ns(struct uts_namespace *old_ns)
 {
      ......
      ns = kmalloc(sizeof(struct uts_namespace), GFP_KERNEL);
      ......
 }
```

上面讲述了 UTS 的代码表示，我们再来只管看一下 UTS Namespace 和 Kref 配合使用的场景。



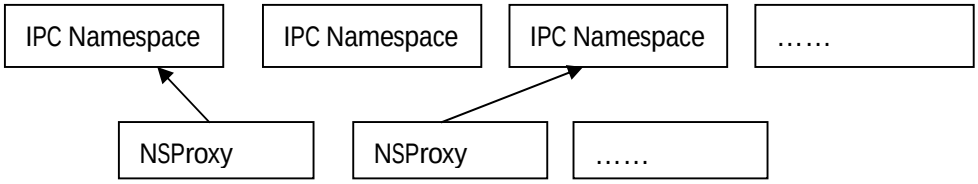
上述顺序描述了 ustname 在容器里面的局部化以及和引用计数配合完成的对象生命周期管理。

IPC 命名空间子模块

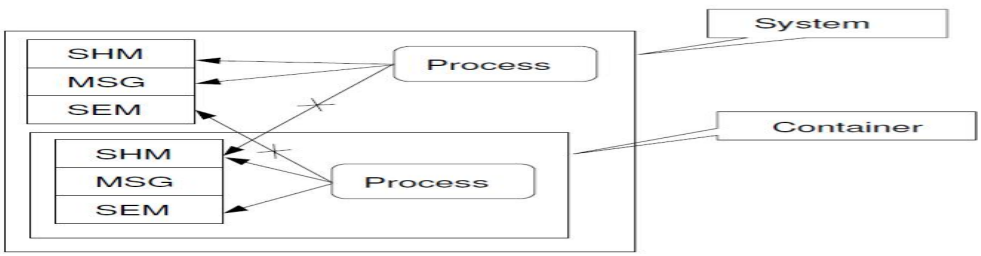
IPC 作为一个常见的进程间通信工具，命名空间对他也进行了部分支持。另外 IPC 也是一个较为简单的扁平化进程间通信工具，命名空间之间不存在层级。

```
struct ipc_namespace {
    atomic_t count;
    struct ipc_ids  ids[3];

    int     sem_ctls[4];
    int     used_sems;

    int     msg_ctlmax;
    int     msg_ctlmnb;
    int     msg_ctlmni;
    atomic_t msg_bytes;
    atomic_t msg_hdrs;
    int     auto_msgmni;

    size_t      shm_ctlmax;
    size_t      shm_ctlall;
    int     shm_ctlmni;
    int     shm_tot;

    struct notifier_block ipcns_nb;

    /* The kern_mount of the mqueuefs sb.  We take a ref on it */
    struct vfsmount *mq_mnt;

    /* # queues in this ns, protected by mq_lock */
    unsigned int    mq_queues_count;

    /* next fields are set through sysctl */
    unsigned int    mq_queues_max;   /* initialized to DFLT_QUEUESMAX */
    unsigned int    mq_msg_max;      /* initialized to DFLT_MSGMAX */
    unsigned int    mq_msgsize_max;  /* initialized to DFLT_MSGSIZEMAX */

} ? end ipc_namespace ? ;
```


上面罗列的主要是 IPC 命名空间里面包含的元素，各个命名空间之间的关系是并列的。



我们直观的给一个图描述资源隔离使用概念图。



属于不同命名空间的进程之间是不能访问对方的全局资源的，这儿展示的主要是 IPC 的 SHM， MSG 以及 SEM，在较新的代码里 MQueue 也可以被隔离。

MNT 命名空间子模块

虚拟机的一个核心功能就是完成应用的隔离，即业务之间相互不可见。这一块主要通过文件系统的视图来完成，进程创建的时候，每一个进程都有自己的文件挂节点信息。看一下经典的 struct task_struct.

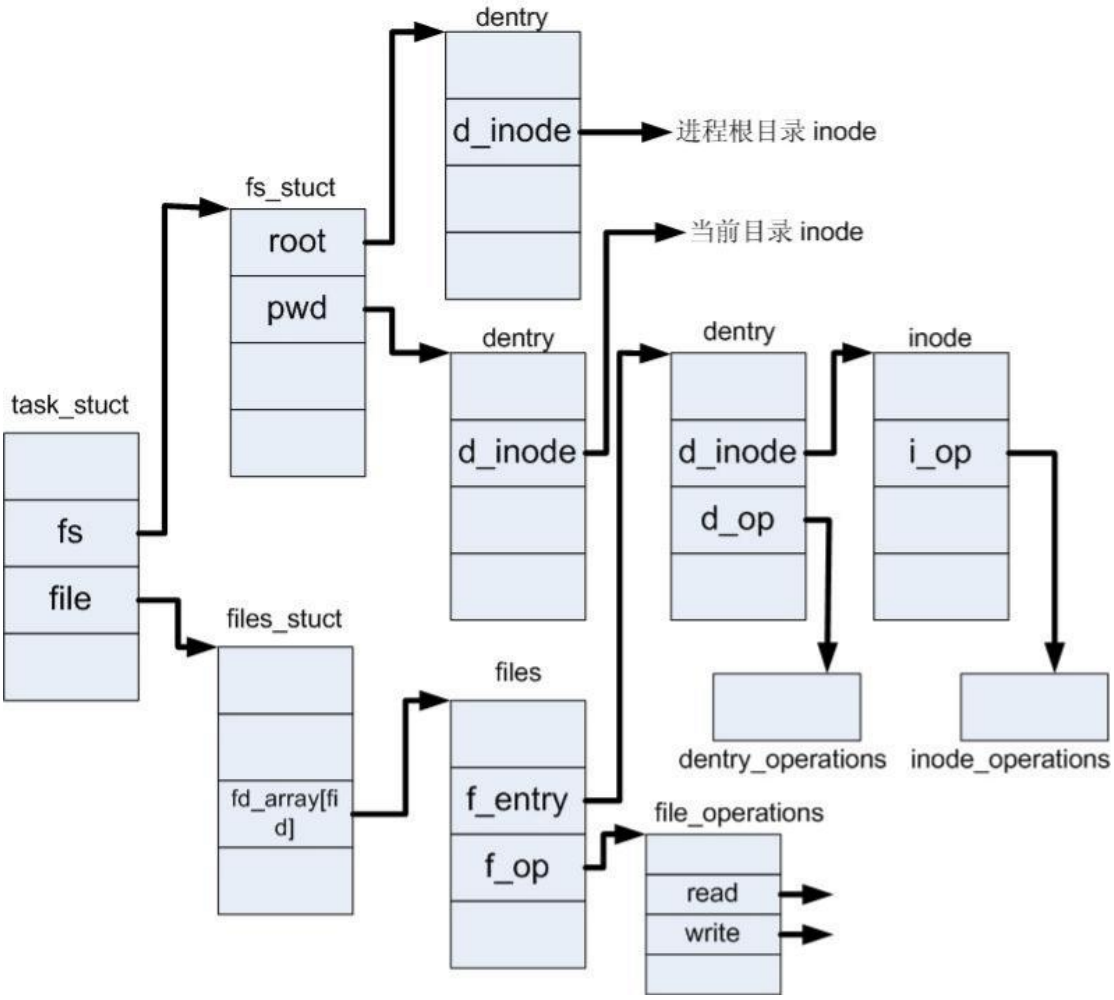
```
struct task_struct {
    ……
    /*当前进程的文件系统信息，包括根目录以及当前用户目录*/
    struct fs_struct *fs;
    /*已经打开的文件列表*/
    struct files_struct *files;
    /*命名空间*/
    struct nsproxy *nsproxy;
    ……
}
```

```
struct fs_struct {
        int users;
        rwlock_t lock;
        int umask;
        int in_exec;
        struct path root, pwd;
};
```

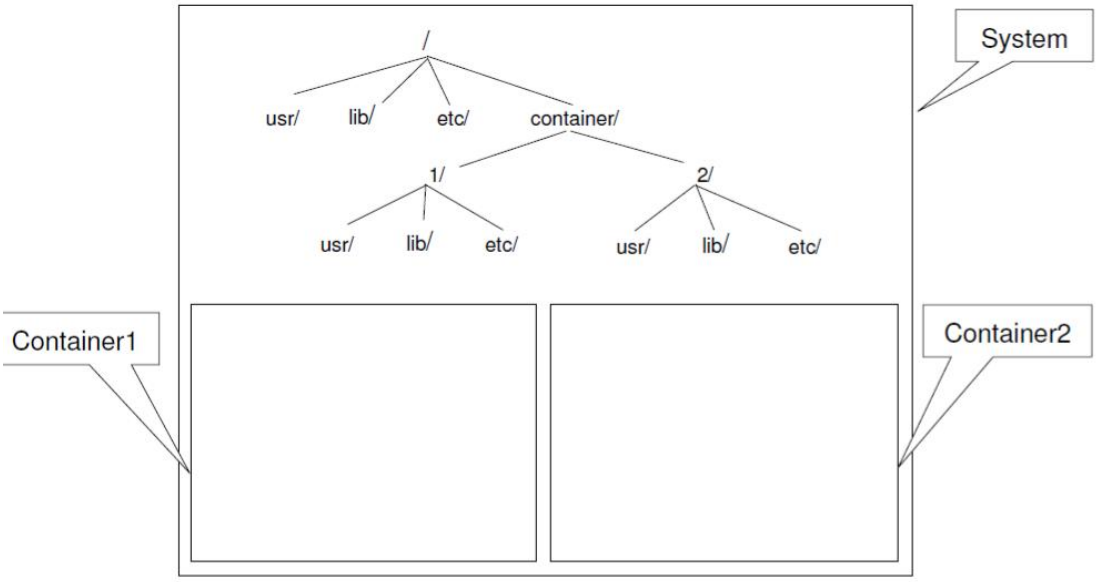
在一个系统启动的时候，0 号进程就设置好了自己所在的根目录以及当前目录。在创建子进程的时候，通过 CLONE_FS 来指明父子之间的共享信息，如果设置了两者共享同一个结构（指针加上引用计数），没有设置标记的话，子进程创建一个新的拷贝，两者之间互不影响。如果设置了 CLONE_FS，接下来通过 chroot(2), chdir(2), or umask(2)的调用结果两者之间会相互影响，反之两者是独立的。

```
static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
{
        struct fs_struct *fs = current->fs;
        if (clone_flags & CLONE_FS) {
                fs->users++;
                return 0;
        }
        tsk->fs = copy_fs_struct(fs);

        return 0;
}
```

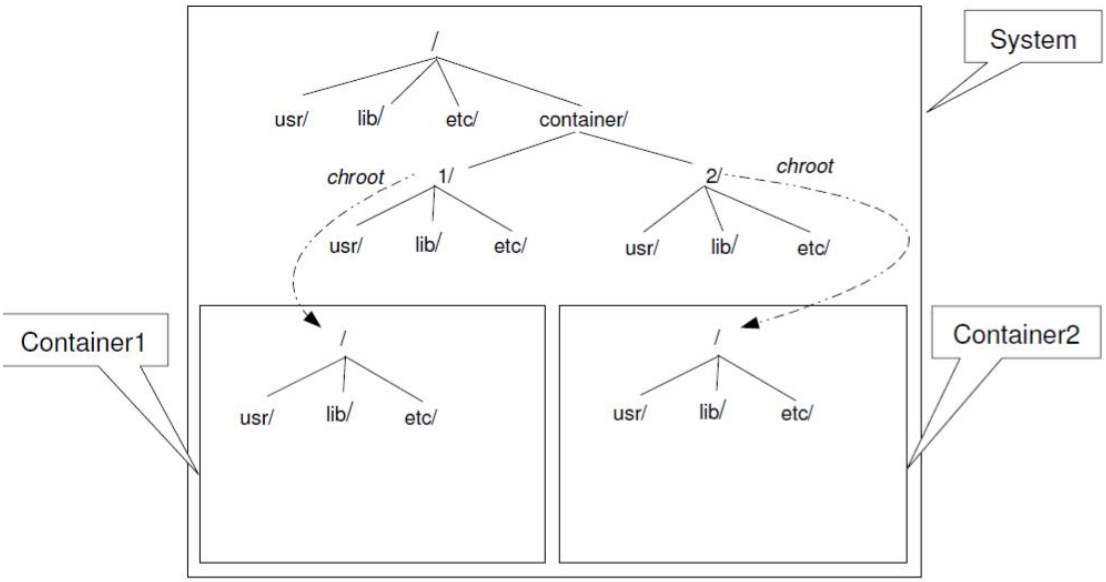
下面这张图清晰明了的刻画了进程内部的文件系统信息以及文件描述符的位置，同时还可以看到一个文件的主要组成部分。



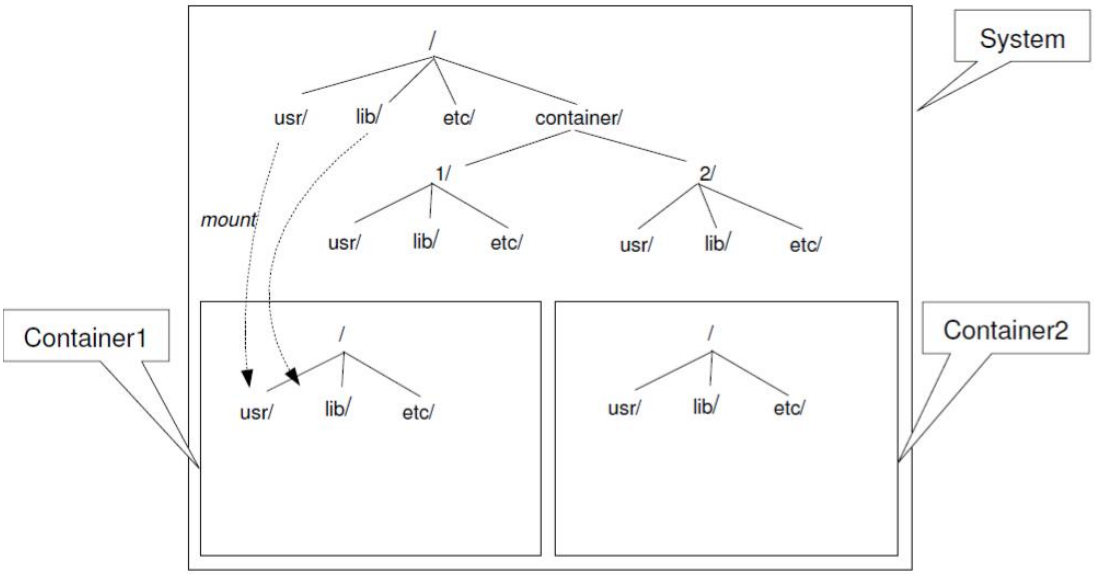
通过文字以及代码描述还是比较枯燥而且不方便直观，下面我们通过图形的方式来看一下进程的文件系统映射情况。



最初我们在系统（system）目录里面创建了一个 container 目录，然后在这个目录里面为每一个虚拟机创建了独立的目录，例如 1 和 2（本例）。在目录 1 和 2 里面分别创建相应虚拟机的根目录文件系统。这样虚拟机启动的时候，我们 chroot 到 1 或者 2 里面，看到的文件系统试图就如下所示。



在这种使用方式下，虚拟机 1 和虚拟机 2 之间的文件系统是互不可见的，而且虚拟机也看不到除了根目录之外的其他文件目录。为了和系统或者其他虚拟机部分共享文件，我们可以映射特定目录到虚拟机的根文件系统，达到部分隔离以及共享的效果，下图。

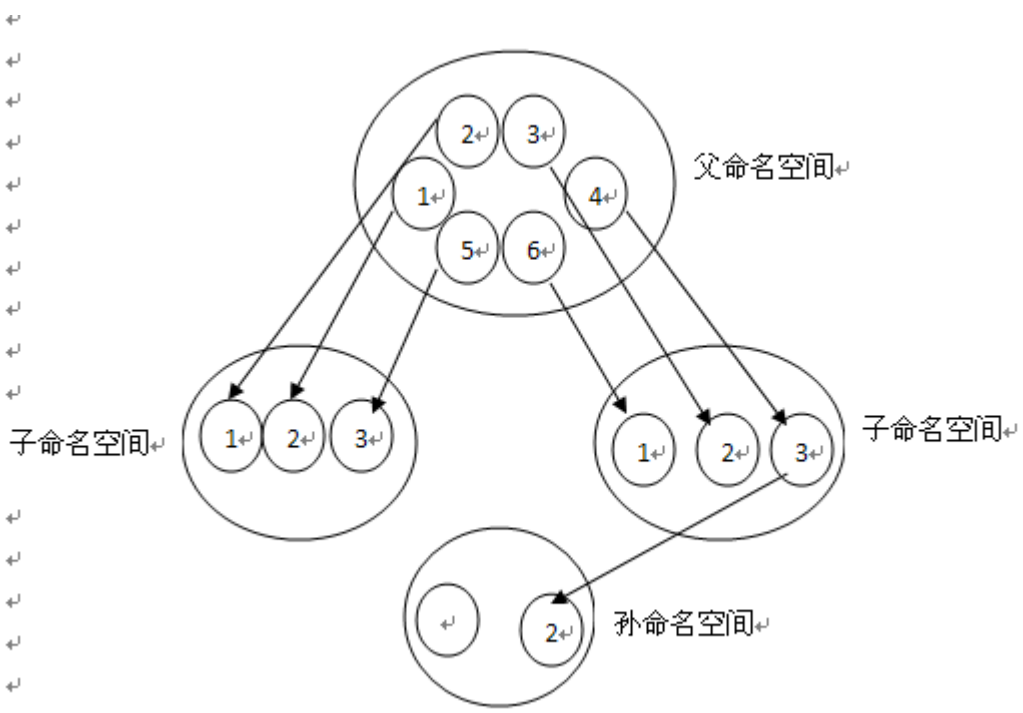


PID 命名空间子模块

PID 是虚拟化命名空间里面较复杂的模块，因为前面的命名空间基本都是扁平的，没有层次结构。但是 PID 命名空间是有层次的，在高层次命名空间能够看到所有的低层次命名空

间信息，反之则不行。

先直观来看看层次化的命名空间结构以及进程的数字变化。需要指出的是，对于命名空间里面的进程，我们看到好像有多个，其实是一一对应的，即进程只有一个，但是在不同的命名空间里面有不同的数据表示，获取一个进程信息需要进程号加上空间信息才能唯一确定一个进程。



看完了 PID 命名空间的组织后，我们来看看他的代码实现。

```
struct pid_namespace {
        struct kref kref;
        struct pidmap pidmap[PIDMAP_ENTRIES];
        int last_pid;
        struct task_struct *child_reaper;
        struct kmem_cache *pid_cachep;
        unsigned int level;
        struct pid_namespace *parent;
        ……
};
```

上图里面重要的一些字段通过红色标注了出来，child_reaper 指向的进程作用相当于全局命名空间的 init 进程，其中一个目的是对孤儿进程进行回收。Level 则表明自己所处的命名空间在系统命名空间里面的深度，这是一个重要的标记，因为层次高的命名空间可以看到低级别的所有信息。系统的命名空间从 0 开始技术，然后累加。命名空间的层次结构通过 parent 来关联。

了解完 PID 命名空间的信息后，我们再来看看 PID 为了支持命名空间所需要做的修改。以前的 PID 命名空间是全局唯一的，现在则必须是命名空间局部化，有一个可见的命名空间就必须有一个 PID 变量。

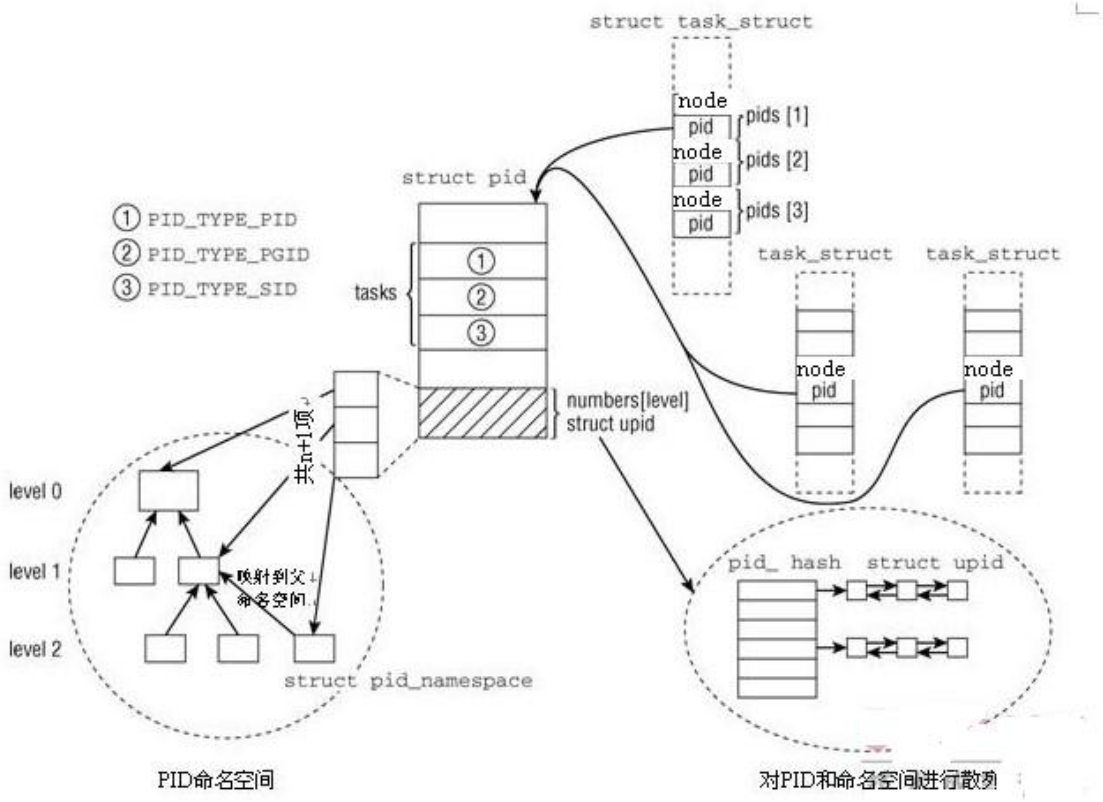
来看看 PID 的内核表示，系统对于每一个 PID 都有一个 PID 结构体来表示，但是在每一个命名空间里面的 upid 表示具体的数值。

```
struct pid
{
    atomic_t count;
    unsigned int level;
    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX];
    struct rcu_head rcu;
    struct upid numbers[1];
};
```

上面的 PID 就是我们在系统中内核的表示，一个 PID 可能对应多个 task_struct，所以在上面的表示里面通过一个 task 数组来表示。接着 numbers 数字分别表示在不同命名空间里面可以看到的 pid 数值，因为 numbers 在最后一个位置，所有本质上来说相当于一个指针，增加命名空间的时候，再增加一个 numbers 即可。

```
struct upid {
    /* Try to keep pid_chain in the same cacheline as nr for find_vpid */
    int nr;
    struct pid_namespace *ns;
    struct hlist_node pid_chain;
};
```

上述的 upid 则是具体的命名空间内数值表示，nr 表示数字，ns 则指向关联的命名空间。当然系统的所有 upid 通过 pid_chain 挂在同一个全局链表里。



PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
struct task_struct {
    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
}

struct pid_link
{
    struct hlist_node node;
    struct pid *pid;
};
```

这张表格和上图一起结合起来我们理解 PID 的管理结构。一个 task_struct 通过 pid_link 的 hlist_node 挂接到 struct pid 的链表上面去。同时 task_struct 又是用过 pid_link 找到 pid，通过 pid 遍历 tasks 链表又能够得到所有的任务，当然也可以读取 numbers 数字获取每一个命名空间里面的数字信息。

为了在 pid 和 upid 之间转换，系统提供了很多内部转换接口，我们首先来了解一些基本指导性原则。

..._nr()

通过 nr 结尾的函数就是获取以前所谓的全局 PID，这个全局 PID 和我们在以前系统里面所见的 PID 是一致的。例如 pid_nr(pid)就返回给定 pid 的全局 PID 数值。这些数值往往只有在本机有效，例如一些通过 PID 获取进程结构的代码。但是在这种情况下，保存 pid 结构往往比全局 PID 更有意义，因为全局 PID 不能随意迁移。

..._vnr()

Vnr 结尾的函数主要和局部 pid 打交道，例如一个进程可见的局部 ID。来看一个例子，task_pid_vnr(tsk)就返回它能够看到任务 PID。需要注意的是，这个数字仅仅在本命名空间内有效。

..._nr_ns()

以 nr_ns 结尾的函数能够获取到特定命名空间课间的 PID 数值，如果你想得到一些任务的 PID 数值，你就可以通过 task_pid_nr_ns(tsk, current->nsproxy->pid_ns)调用得到数字，接着通过 find_task_by_pid_ns(pid, current->nsproxy->pid_ns)反过来找到任务结构。当一个用户请求过来的时候，基本上都是调用这组函数，因为这种情况下一个任务可能需要得到另外一个命名空间的信息。

NET 命名空间子模块

NET 的命名空间隔离做的工作相对而言是最多的，但是整体思路还是一致的，即把全局的资源局部化，在每一个命名空间里面保留自己的控制信息。例如在目前的内核里面路由表，arp 表，netfilter 表，设备等等都已经空间化了，其所做的后续操作都要首先关联到特定的命名空间，然后再取出里面的数据进行后面的分析。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

首先来看看 net 命名空间的结构体

```
struct net {
        atomic_t        count;/* To decided when the network namespace should be freed. */
#ifdef NETNS_REFCNT_DEBUG
        atomic_t        use_count;      /* To track references we destroy on demand*/
#endif
        struct list_head        list;           /* list of network namespaces */
        struct work_struct  work;       /* work struct for freeing */
        struct proc_dir_entry     *proc_net;
        struct proc_dir_entry     *proc_net_stat;
#ifdef CONFIG_SYSCTL
        struct ctl_table_set sysctls;
#endif
        struct net_device             *loopback_dev;                /* The loopback */
        struct list_head        dev_base_head;
        struct hlist_head     *dev_name_head;
        struct hlist_head     *dev_index_head;
        /* core fib_rules */
        struct list_head        rules_ops;
        spinlock_t              rules_mod_lock;
        struct sock             *rtnl;                  /* rtnetlink socket */

        struct netns_core    core;
        struct netns_mib     mib;
        struct netns_packet        packet;
        struct netns_unix    unx;
        struct netns_ipv4    ipv4;
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
        struct netns_ipv6    ipv6;
#endif
#if defined(CONFIG_IP_DCCP) || defined(CONFIG_IP_DCCP_MODULE)
        struct netns_dccp    dccp;
#endif
#ifdef CONFIG_NETFILTER
        struct netns_xt              xt;
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
        struct netns_ct              ct;
#endif
#endif
#ifdef CONFIG_XFRM
        struct netns_xfrm    xfrm;
#endif
        struct net_generic  *gen;
};
```

上面这个结构实在较大，您在需要深入了解的时候慢慢了解每个子模块吧。在一个命名空间创建的时候，会做一些初始化。所有系统定义了一个回调函数，让感兴趣的模块注册。结构如下：

```
struct pernet_operations {
    struct list_head list;
    int (*init)(struct net *net);
    void (*exit)(struct net *net);
};
```

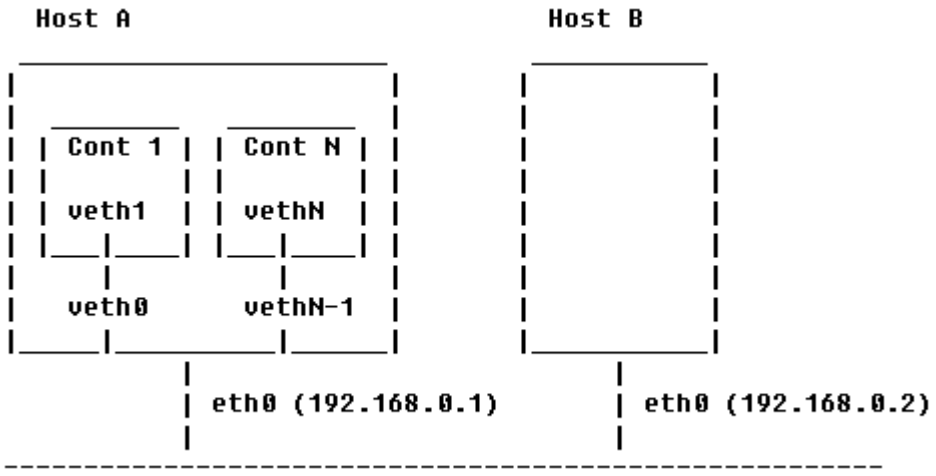
注册接口如下

```
register_pernet_operations
```

一个新的用户空间被创建的时候，注册模块的 init 结构被创建。同理，一个空间销毁的时候，exit 函数也会被调用。

那么现在有了很多命名空间，而且命名空间之间是隔离的，那么他们之间怎么通信呢。这儿需要注意的是引入了一个新的概念，就做网络设备对。一个设备对即 A 设备接收到的时间自动发送到 B 设备，反之亦然。

我们先来从直观上看一下网络概念图。



接下来的问题就是如何通信？其实可以通过二层或者三层来实现网络转发，本质上就是通过桥接还是路由。我们下面以桥接为例来说明数据报文是如何转发的。

对于容器 1 来说，veth1 需要配置一个 IP 地址，但是 veth0 和 eth0 配置在同一个桥接设备上。Veth0 和 veth1 是网络设备对。

```
linux-kgwg:/home/project/template/suse # brctl showmacs br0
port no mac addr                is local?       ageing timer
  1     00:0c:29:55:1f:4a         yes                0.00
  1     00:50:56:c0:00:08         no                 0.02
  1     00:50:56:f5:30:40         no                 0.50
  2     ba:4e:66:d8:da:83         no                 7.58
  2     d6:d1:8e:a1:b4:57         yes                0.00
```

```
Veth0     Link encap:Ethernet  HWaddr D6:D1:8E:A1:B4:57
Veth1     Link encap:Ethernet  HWaddr BA:4E:66:D8:DA:83
```

是一个实际网络环境里面的虚拟化配置，veth0 和 eth0 是通过桥接来完成转发，但是 veth0 和 veth1 之间是通过设备对来完成数据转发，其他概念都没有太多变化，除了增加一个独立的命名空间，这儿我们来看看网络设备对是如何工作的。

创建过程不再讨论，就是每次创建一对，A 和 B 的对端分别指向彼此。

```
struct veth_priv {
        struct net_device *peer;
        struct veth_net_stats *stats;
        unsigned ip_summed;
};
```

创建完了后，彼此关联。

```
static int veth_newlink(struct net_device *dev,
                        struct nlattr *tb[], struct nlattr *data[])
{
        priv = netdev_priv(dev);
        priv->peer = peer;

        priv = netdev_priv(peer);
        priv->peer = dev;
        return 0;
}
```

我们还是来看看数据通道，他们的数据是如何透传的。

```
static int veth_xmit(struct sk_buff *skb, struct net_device *dev)
{
        ......
        priv = netdev_priv(dev);
        rcv = priv->peer;
        rcv_priv = netdev_priv(rcv);

        cpu = smp_processor_id();

        skb->pkt_type = PACKET_HOST;
        skb->protocol = eth_type_trans(skb, rcv);
        if (dev->features & NETIF_F_NO_CSUM)
                skb->ip_summed = rcv_priv->ip_summed;

        netif_rx(skb);
        return 0;
        ......
}
```

过 eth_type_trans 替换设备指针，接着就通过 netif_rx 送上起，设备已经属于一个特定的命名空间了，接着就在特定的命名空间里面完成这个报文的应用层处理。但是不管怎么样，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

通过这个小函数我们就能够轻松的把一个报文从一个命名空间发送到另外一个命名空间里面。

总结

终于把内核的命名空间大致模块都看了一遍，其实我们只要在心理面把握住重要的一点就可以了。所有虚拟化的资源，在获取资源的时候，必须首先通过 nsproxy 获取到合适的命名空间，然后再进行接下来的操作。另外命名空间虽然用户空间程序不是很多，但是在内核里面很早就引入了，而且一堆人活跃的维护着。

As of kernel 2.6.26 the major Namespaces are embeded as detailed below.

Resource	Status	Article	-mm version	mainline version
SHARED SUBTREES	Done	lwn	–	2.6.15
UTSNAME	Done	lwn	–	2.6.19
PID	Done	lwn	–	2.6.24
IPC	Done	lwn	–	2.6.19
USER	Done	lwn	–	2.6.23
NETWORK	Done	lwn	–	2.6.26
/PROC	Done	none	–	2.6.26
RO BIND MOUNT	Done	lwn	–	2.6.24