空天信息安全与可信计算
教育部重点实验室

Linux分析与安全设计

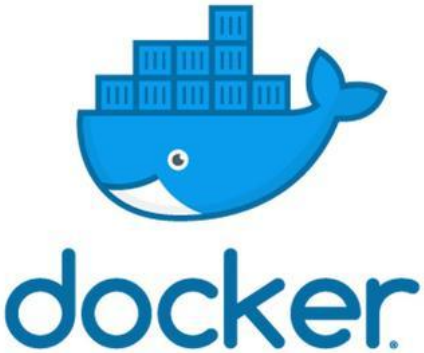
第7章　NameSpace及Cgroups机制

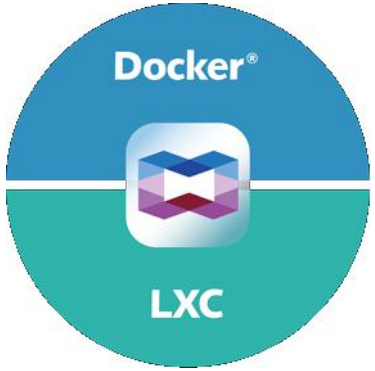
NameSpace及Cgroups机制

- NameSpace和Cgroups
- Namespace基本原理
- NameSpace的实现机制
- Cgroups基本原理
- Cgroups实现机制

NameSpace和Cgroups

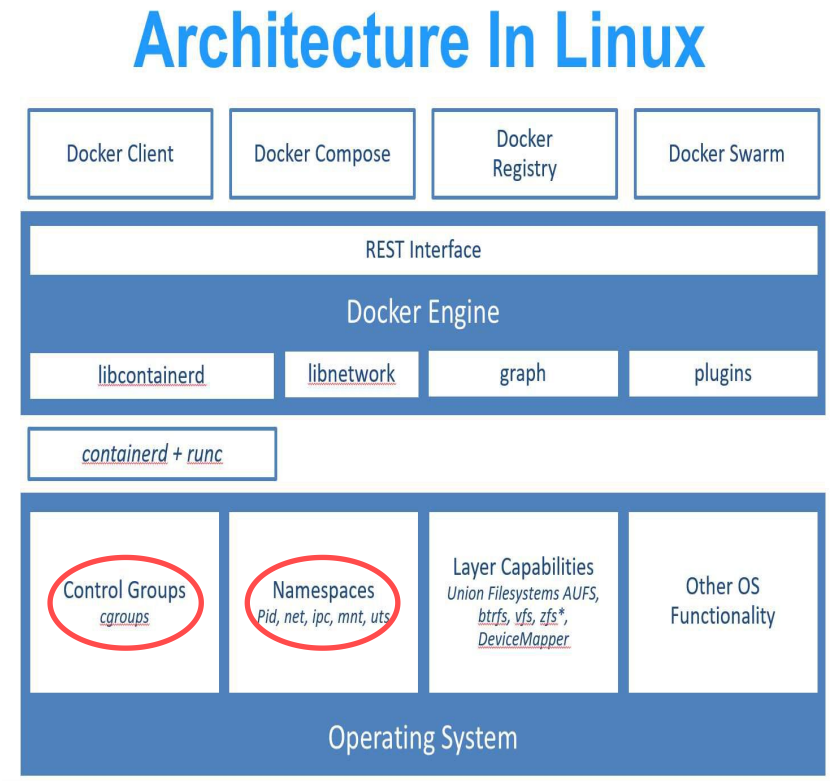
- Namespace和Cgroups是Linux里面基本的进程隔离机制
- Namespace负责进程信息隔离
- Cgroups对进程或进程组的资源大小进行限制
- 它们是LXC（Linux Container）和Docker的核心技术





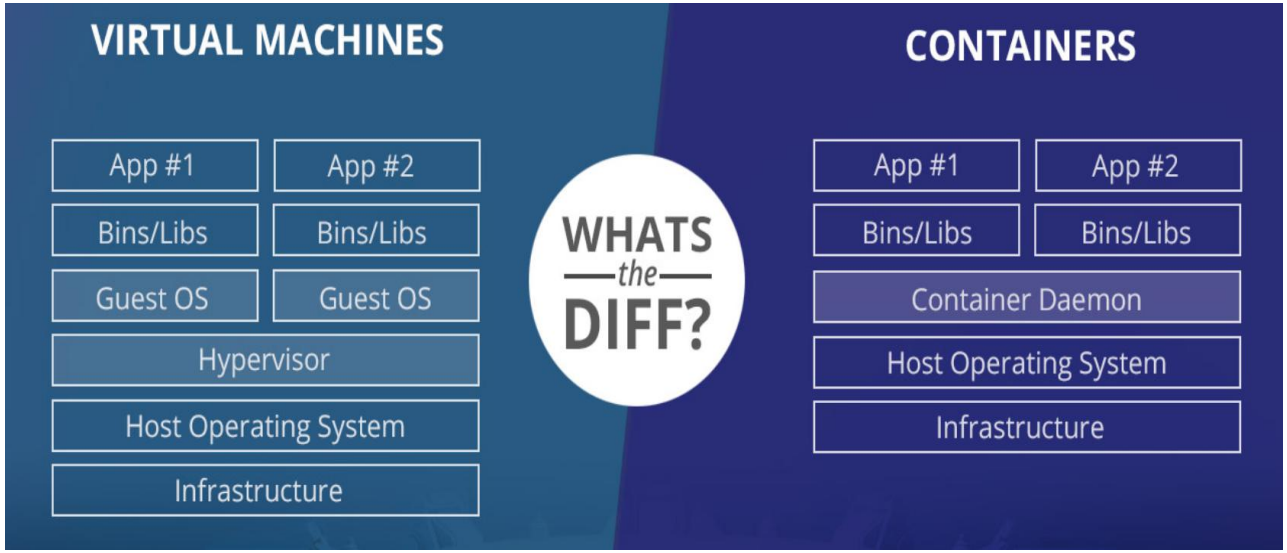
Docker

- Docker是一个开源的应用容器引擎，是基于Go语言实现的云开源项目。
- 开发者可以基于Docker打包他们的应用以及依赖包到一个可移植的镜像中，然后发布到任何流行的 Linux或Windows 机器上，也可以实现虚拟化。
- 容器使用沙箱机制,相互之间隔离，默认看不到或者使用对方资源



容器和虚拟机的区别

- 容器和容器是共享宿主机操作系统内核的
- 相比于虚拟机，容器拥有更高的资源使用效率，如实例小、启动快
- 容易管理、快速实现应用的迁移



容器（Container)

如何实现一个资源隔离的运行环境（容器）

- 进程需要隔离，每一个容器内部需要有自己的PID，容器之间相互隔离
- 需要有独立的文件系统
- 网络通信是独立的，两个容器需要有各自独立的IP，端口等等

NameSpace

- Linux Namespace 是 Linux 提供的一种内核级别环境隔离的方法
- Linux Namespace 将全局系统资源封装在一个抽象的资源标识中，从而使 namespace 内的进程认为自己具有独立的资源实例
- 改变一个 namespace 中的系统资源只会影响当前 namespace 里的进程，对其他 namespace 中的进程没有影响

NameSpace

Linux内核逐渐实现了以下几种隔离机制

- UTS namespaces——主机名与域名隔离
- IPC namespaces——信号量、消息队列和共享内存隔离
- Mount namespaces——挂载点隔离（文件系统）
- PID namespaces——进程ID隔离
- Network namespaces——网络隔离
- User namespaces——用户和用户组隔离

UTS namespace

- UTS namespace提供了hostname和NIS domainname的隔离
- 每个容器就可以拥有了独立的主机名和NIS domainname域名，在网络上可以被视作一个独立的节点而非宿主机上的一个进程

hostname

UTS namespace

include/linux/uts.h 定义了与UTS有关的数据

```
#ifndef UTS_SYSNAME
#define UTS_SYSNAME "Linux"
#endif

#ifndef UTS_NODENAME
#define UTS_NODENAME CONFIG_DEFAULT_HOSTNAME /* set by
sethostname() */
#endif

#ifndef UTS_DOMAINNAME
#define UTS_DOMAINNAME "(none)" /* set by setdomainname() */
#endif
```

NameSpace API

操作Namespace 的 API 由三个系统调用和一系列 /proc 文件组成

- clone() : 用来创建一个新的进程，通过一系列参数来指定新的进程的一些特性
- setns() : 把某进程加入到某个 namespace
- unshare() : 使某进程脱离某个 namespace

NameSpace API

为了确定隔离的到底是哪种namespace，在使用这些API时，通常需要指定以下六个常数的一个或多个，通过|（位或）操作来实现。这六个参数分别是

- CLONE_NEWIPC
- CLONE_NEWNS
- CLONE_NEWNET
- CLONE_NEWPID
- CLONE_NEWUSER
- CLONE_NEWUTS

Clone API

- clone()实际上是传统UNIX系统调用fork()的一种更通用的实现方式，它可以通过flags来控制函数的功能。

- clone()用来创建一个拥有独立namespace的进程，它的调用方式如下：

```
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);
```

Clone API

int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg)

- child_func传入子进程运行的程序主函数
- child_stack传入子进程使用的栈空间
- flags表示使用哪些CLONE_*标志位
- args则可用于传入用户参数

UTS namespace

调用 clone 函数生成新的进程时可以通过设置 CLONE_NEWUTS 标识让子进程拥有自己的 UTS namespace

```c
#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];
char* const child_args[] = {
  "/bin/bash",
  NULL
};

int child_main(void* args) {
  printf("在子进程中!\n");
  printf("pid is:%d\n", getpid());
  printf("ppid is:%d\n", getppid());
  sethostname("test",4);
  execv(child_args[0], child_args);
  return 1;
}

int main() {
  printf("程序开始: \n");
  int child_pid = clone(child_main, child_stack + STACK_SIZE, CLONE_NEWUTS | SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

新建一个hostname.c，在子进程中重新设置主机名

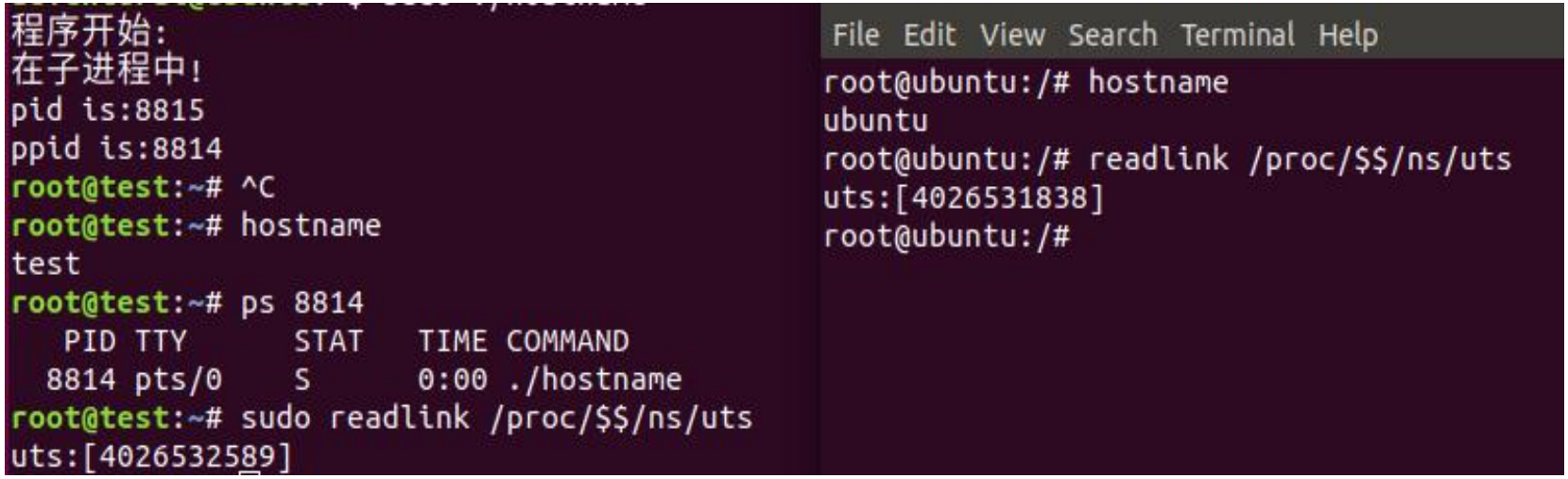
UTS namespace

Linux下编译运行后

gcc -Wall hostname.c -o hostname

./hostname

主机名由ubuntu变成了test，子进程和右边的bash进程的uts namespce编号也不同了，但是子进程可以看到父进程的信息，PID空间是共享的



UTS namespace

使用setns系统调用将一个新进程的加到一个已经存在的UTS namespace

```c
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char* uts_namespace = argv[1];
    int fd = open(uts_namespace, O_RDONLY);
    if(fd == -1){
        printf("open failed\n");
    }
    char *hostname = (char*)malloc(10);
    gethostname(hostname, 10);
    printf("%s\n", hostname);
    setns(fd,0);
    gethostname(hostname, 10);
        printf("%s\n", hostname);
    return 0;
}
```

传入一个UTS，然后将进程加入到这个UTS空间内

UTS namespace

编译运行上面的代码

- 8815是新UTS子进程的PID，调用setns后，两次调用gethostname的值已经不一样了，此时进程的UTS namespace已经改变了

```
adventural@ubuntu:~$ gcc -Wall addtotest.c -o addtotest
adventural@ubuntu:~$ sudo ./addtotest /proc/8815/ns/uts
ubuntu
test
adventural@ubuntu:~$
```

UTS 源码分析

- Linux Namespace是用来做进程资源隔离的，进程描述符task_struct中包含了对应的namespace信息
- task_struct中使用proxy这个结构体描述进程的namespace信息

```
struct task_struct {
        ...
        /* namespaces */
        struct proxy *proxy;
        ...
}
```

内核 include/linux/sched.h

UTS 源码分析

include/linux/proxy.h中proxy结构体

```
struct proxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct cgroup_namespace *cgroup_ns;
};
```

count记录指向这个proxy结构体的指针数

UTS 源码分析

proxy.h中定义的操作proxy结构体的操作

- int copy_namespaces(unsigned long flags, struct task_struct *tsk);
- void exit_task_namespaces(struct task_struct *tsk);
- void switch_task_namespaces(struct task_struct *tsk, struct proxy *new);

namespace被clone时，proxy就会跟着被clone

UTS 源码分析

proxy.h中定义的操作proxy结构体的API

- void free_proxy(struct proxy *ns);
- int unshare_proxy_namespaces(unsigned long, struct proxy **,
- struct cred *, struct fs_struct *);
- int __init proxy_cache_init(void);
- static inline void put_proxy(struct proxy *ns) { … }
- static inline void get_proxy(struct proxy *ns) { … }

UTS 源码分析

uts_namespace结构体（include/linux/utsname.h ）

```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;
        struct user_namespace *user_ns;
        struct ucounts *ucounts;
        struct ns_common ns;
};
```

- kref是一个引用计数器，它结构中只包含一个atomic_t类型的计数值，atomic_t是原子类型，对其操作都要求是原子执行的
- user_ns指示了拥有这个uts_namespace的user
- ns_common集合了namespace在proc中的所有抽象

UTS 源码分析

启用namespace后的gethostname( )函数（ kernel/sys.c ）

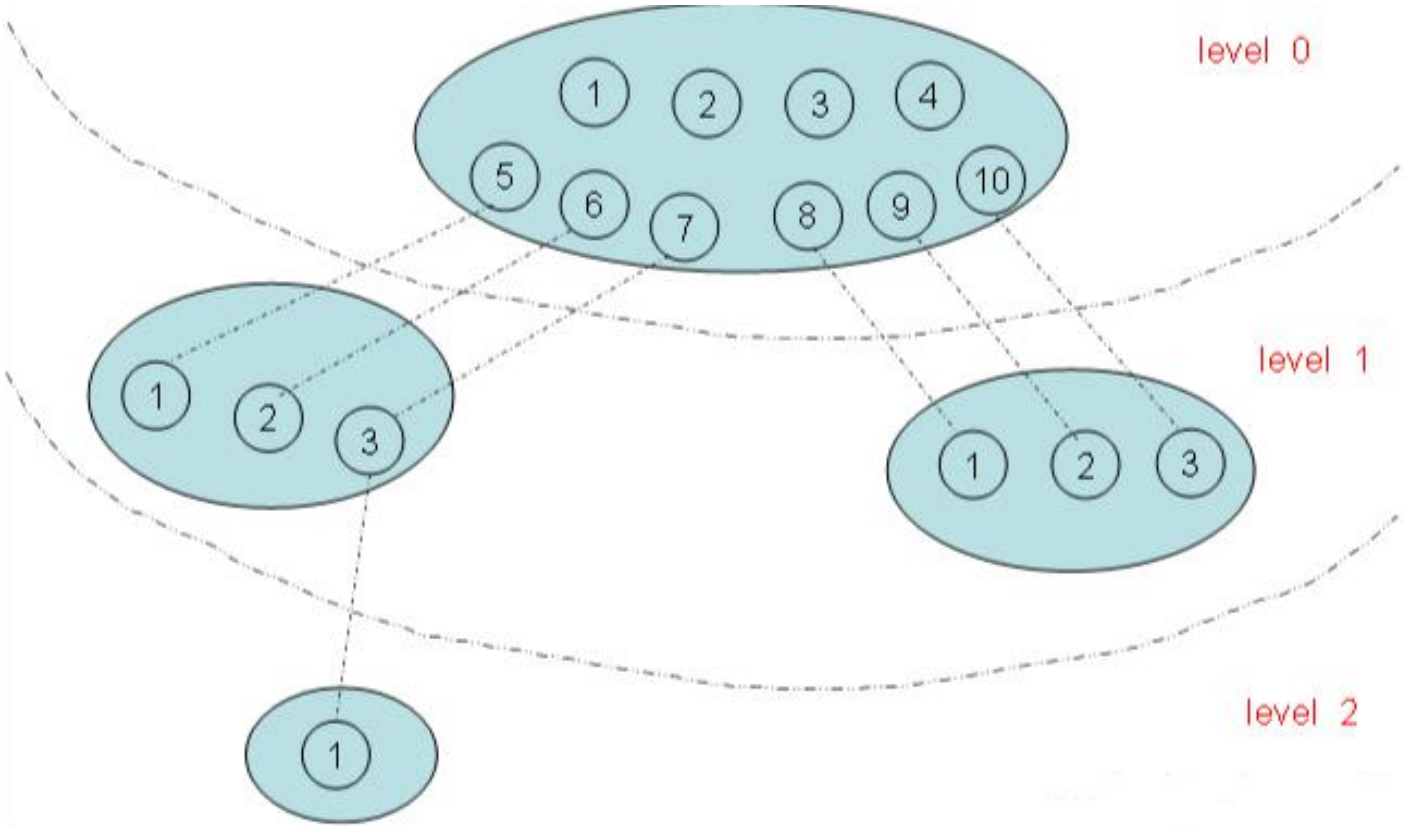
```
static inline struct new_utsname *utsname(void)
{
  //current指向当前进程的task结构体
  return &current->nsproxy->uts_ns->name;
}

SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
{
  struct new_utsname *u;
  ...
  u = utsname();
  if (copy_to_user(name, u->nodename, i)){
    errno = -EFAULT;
  }
  ...
}
```

PID namespace

- PID namespace隔离对进程PID重新标号，即两个不同namespace下的进程可以有同一个PID
- 内核为所有的PID namespace维护了一个树状结构，最顶层的是系统初始时创建的，可以称之为root namespace
- 父节点namespace可以看到子节点中的进程，并可以通过信号等方式对子节点中的进程产生影响，但是反过来不行

PID namespace



原始PID

NameSpace PID

PID namespace

修改hostname.c，加入PID隔离

```
int child_pid = clone(child_main, child_stack+STACK_SIZE,
        CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWUTS  | SIGCHLD,
          NULL)
```

```
adventural@ubuntu:~$ sudo ./hostname
程序开始:
在子进程中!
pid is:1
ppid is:0
root@test:~# echo $$
1
root@test:~# exit
exit
已退出
adventural@ubuntu:~$ echo $$
18680
adventural@ubuntu:~$
```

子进程内的PID编号
已经从1开始了

PID namespace

```
struct pid_namespace {
        struct kref kref;
        struct pidmap pidmap[PIDMAP_ENTRIES];
        struct rcu_head rcu;
        int last_pid;
        unsigned int nr_hashed;
        struct task_struct *child_reaper;
        struct kmem_cache *pid_cachep;
        unsigned int level;
        struct pid_namespace *parent;
#ifdef CONFIG_PROC_FS
        struct vfsmount *proc_mnt;
        struct dentry *proc_self;
        struct dentry *proc_thread_self;
        ……..
};
```

PID namespace

```
struct upid {
        /* Try to keep pid_chain in the same
cacheline as nr for find_vpid */
        int nr;                         //pid的数值
        struct pid_namespace *ns;  // 所在命名空间
        struct hlist_node pid_chain;   // 链表节点
};
```

Mount namespace

- Mount namespace通过隔离文件系统挂载点对隔离文件系统提供支持，隔离后，不同mount namespace中的文件结构发生变化也互不影响。
- 进程在创建mount namespace时，会把当前的文件结构复制给新的namespace，新namespace中的所有mount操作都只影响自身的文件系统
- 通过/proc/[pid]/mounts查看到所有挂载在当前namespace中的文件系统

Mount namespace

此时如果在子进程的shell中执行了ps aux/top之类的命令，发现还是可以看到所有父进程的PID

- 没有对文件系统进行隔离，ps/top之类的命令调用的是真实系统下的/proc文件内容

如果为了实现一个稳定安全的容器，PID namespace还需要进行一些额外的工作才能确保其中的进程运行顺利

Mount namespace

在子进程中重新挂载proc文件系统

mount -t proc proc /proc

使用ps aux列出的进程

```
程序开始:
在子进程中!
pid is:1
ppid is:0
root@test:~# mount -t proc proc /proc
root@test:~# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.1  22832  5148 pts/0    S    00:30   0:00 /bin/bash
root         24  0.0  0.0  37368  3300 pts/0    R+   00:31   0:00 ps aux
```

Mount namespace

fs/mount.h中定义了mnt_namespace结构体

```
struct mnt_namespace {
        atomic_t     count;
        struct ns_common      ns;
        struct mount *     root;
        struct list_head    list;
        struct user_namespace     *user_ns;
        u64     seq;
        wait_queue_head_t     poll;
        u64     event;
};
```

IPC namespace

- 容器中进程间通信采用的方法包括常见的信号量、消息队列和共享内存
- IPC Namespace主要针对的是SystemV IPC和Posix消息队列，其中System V IPC 对象包含共享内存、信号量和消息队列
- IPC namespace中实际上包含了系统IPC标识符以及实现POSIX消息队列的文件系统

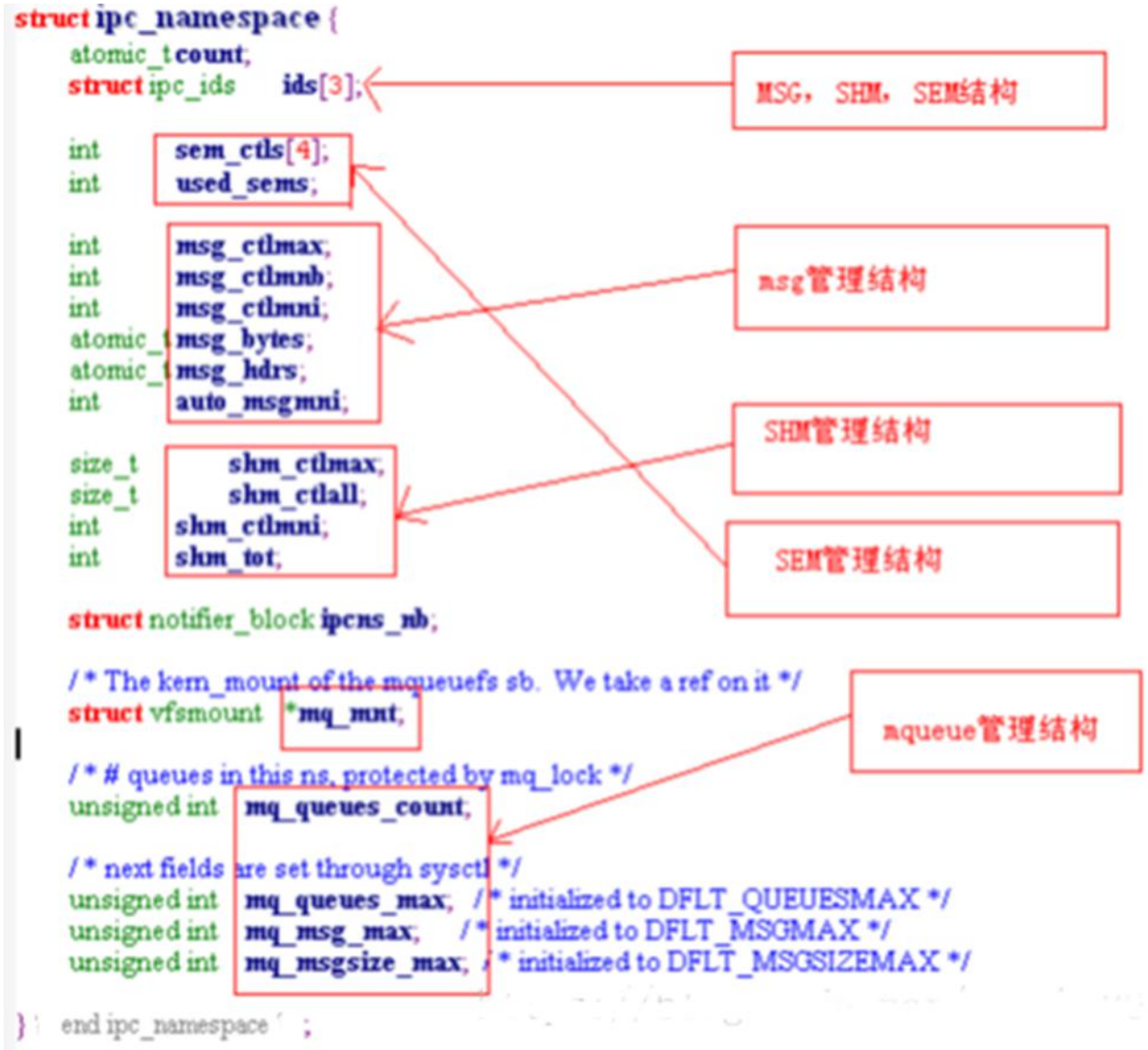
IPC namespace

可以使用CLONE_NEWIPC参数来创建一个IPC隔离的进程

```
int child_pid = clone(child_main, child_stack+STACK_SIZE,
        CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD, NULL)
```

目前使用IPC namespace机制的系统不多，其中比较有名的有PostgreSQL，Docker本身通过socket进行通信

IPC namespace



IPC namespace

```
struct ipc_namespace {

    struct ipc_ids ids[3];//三种通信方式

}
```

IPC NameSpace

```
#define IPC_SEM_IDS 0
#define IPC_MSG_IDS 1
#define IPC_SHM_IDS 2
```

```
#define sem_ids(ns) ((ns)->ids[IPC_SEM_IDS])
#define msg_ids(ns) ((ns)->ids[IPC_MSG_IDS])
#define shm_ids(ns) ((ns)->ids[IPC_SHM_IDS])
```

Network namespace

Network namespace主要提供了关于网络资源的隔离如：

- 网络设备
- IPv4和IPv6协议栈
- IP路由表
- /proc/net目录
- /sys/class/net目录
- 端口（socket）

…

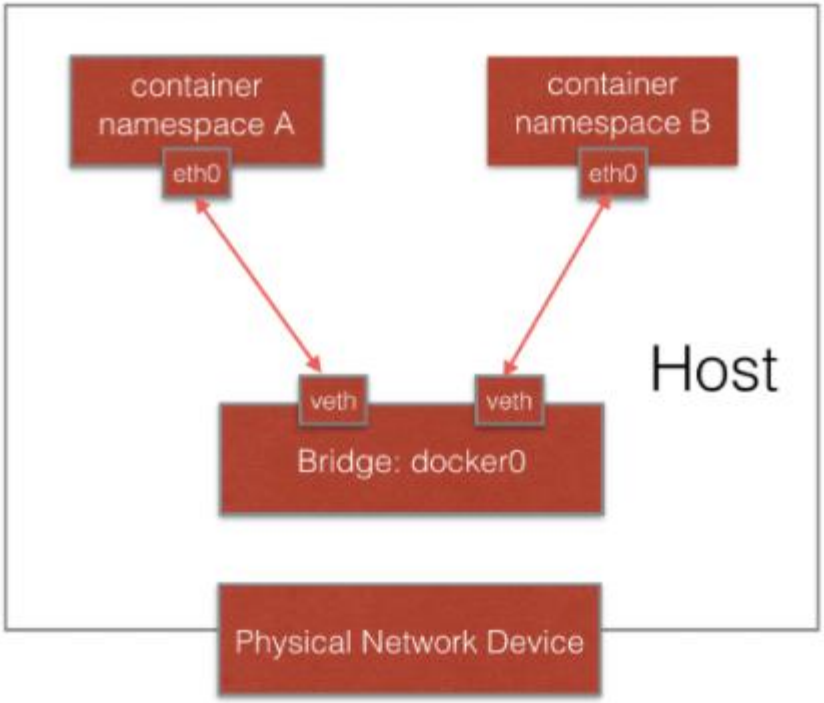
Network namespace

- network namespace有时指的未必是真正的网络隔离，而是把网络独立出来，给外部用户一种透明的感觉，仿佛跟另外一个网络实体在进行通信
- 容器的经典做法就是创建一个veth pair，一端放置在新的namespace中，通常命名为eth0，一端放在原先的namespace中连接物理网络设备，再通过网桥把别的设备连接进来或者进行路由转发，以此网络实现通信的目的

Network namespace

以Docker Daemon在启动容器dockerinit的过程为例

- Docker Daemon在宿主机上负责创建这个veth pair
- 建立的过程中，Docker Daemon和dockerinit就通过pipe进行通信
- dockerinit在管道的另一端循环等待，直到管道另一端传来Docker Daemon关于veth设备的信息，并关闭管道，然后启动eth0



User namespace

User namespace主要隔离了安全相关的标识符（identifiers）和属性（attributes）如：

- 用户ID
- 用户组ID
- root目录
- key（指密钥）
- 特殊权限

一个普通用户的进程通过clone()创建的新进程在新user namespace中可以拥有不同的用户和用户组

/proc/pid/ns

从3.8版本的内核开始，用户就可以在/proc/[pid]/ns文件下看到指向不同namespace号的文件（$$代表当前bash）

ll /proc/$$/ns

```
root@ubuntu:/# ll /proc/$$/ns
total 0
dr-x--x--x 2 root root 0 Nov 18 23:44 ./
dr-xr-xr-x 9 root root 0 Nov 18 23:44 ../
lrwxrwxrwx 1 root root 0 Nov 18 23:44 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 net -> 'net:[4026531993]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov 18 23:44 uts -> 'uts:[4026531838]'
root@ubuntu:/#
```

Cgroups

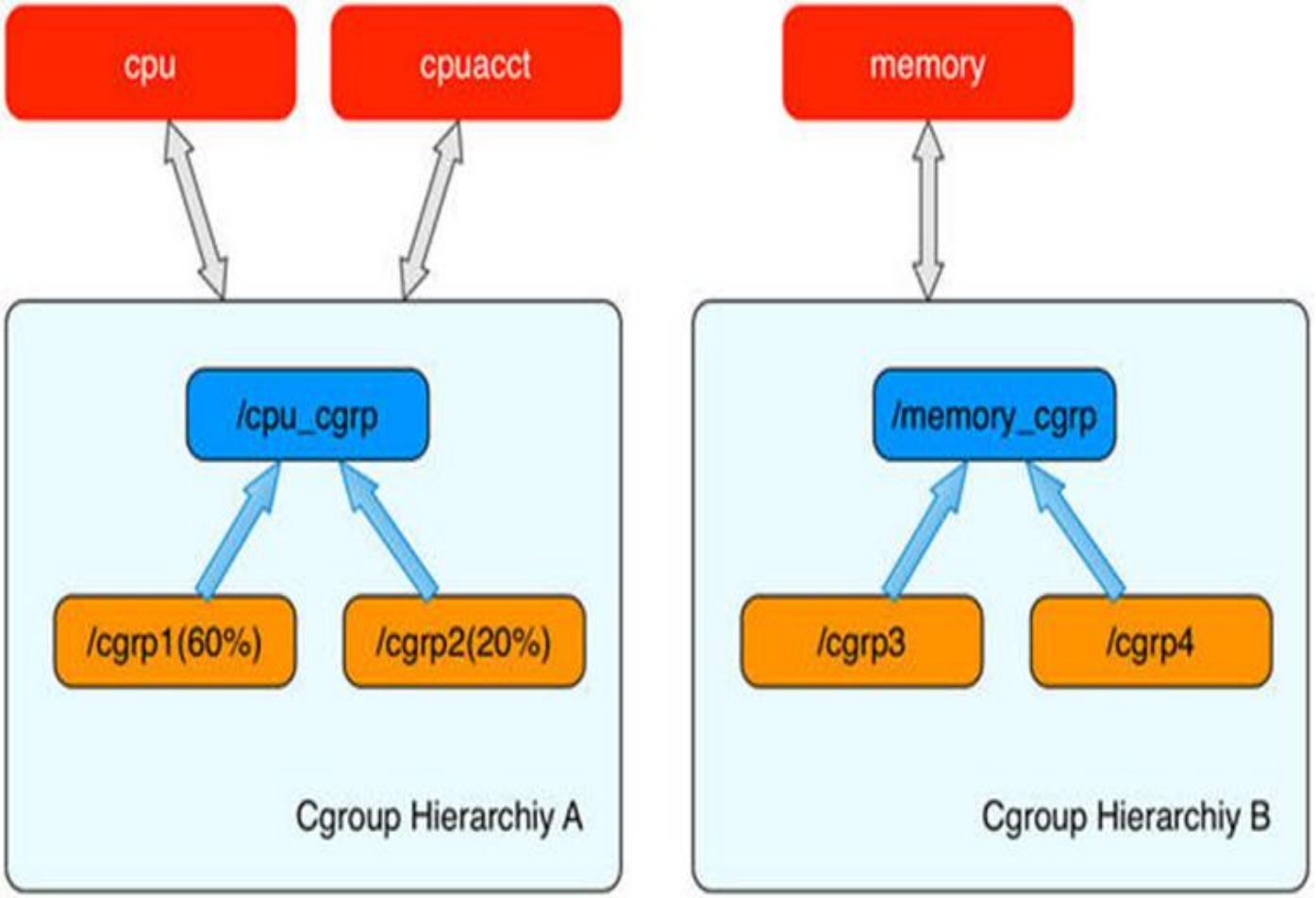
Cgroups全称Control Groups，是Linux内核提供的物理资源隔离机制，通过这种机制，可以实现对Linux进程或者进程组的资源限制、隔离和统计功能

- LXC(Linux Containers)和Docker容器所用到的资源隔离技术就是Cgroups

Cgroups相关概念

- 任务(task)::：在cgroup中，任务就是一个进程
- 控制组(control group)::：cgroup的资源控制是以控制组的方式实现，控制组指明了资源的配额限制，进程可以加入到某个控制组，也可以迁移到另一个控制组
- 层级(hierarchy)::：控制组有层级关系，类似树的结构，子节点的控制组继承父控制组的属性(资源配额、限制等)
- 子系统(subsystem)::：一个子系统其实就是一种资源的控制器，比如memory子系统可以控制进程内存的使用，子系统需要加入到某个层级，然后该层级的所有控制组均受到这个子系统的控制

Cgroups相关概念



Cgroups子系统分类

- cpu：限制进程的 cpu 使用率
- cpuacct 子系统：可以统计 cgroups 中的进程的 cpu 使用报告
- cpuset:：为cgroups中的进程分配单独的cpu节点或者内存节点
- memory:：限制进程的memory使用量
- blkio:：限制进程的块设备I/O

Cgroups子系统分类

- devices：控制进程能够访问某些设备
- net_cls:：标记cgroups中进程的网络数据包，然后可以使用tc模块（traffic control）对数据包进行控制
- net_prio： 限制进程网络流量的优先级
- huge_tlb：限制HugeTLB的使用
- freezer：挂起或者恢复cgroups中的进程
- ns：控制cgroups中的进程使用不同的namespace

CPU子系统简介

- 安装Cgroups
  - sudo apt install cgroup-bin
- 安装完成后，系统会出现/sys/fs/cgroup目录
  - cd /sys/fs/cgroup/cpu

```
adventural@ubuntu:/sys/fs/cgroup/cpu$ ll
total 0
dr-xr-xr-x  4 root root   0 Nov 20 00:29 ./
drwxr-xr-x 15 root root 380 Nov 20 00:29 ../
-rw-r--r--  1 root root   0 Nov 20 01:25 cgroup.clone_children
-rw-r--r--  1 root root   0 Nov 20 00:29 cgroup.procs
-r--r--r--  1 root root   0 Nov 20 01:25 cgroup.sane_behavior
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.stat
-rw-r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_all
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_percpu
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_percpu_sys
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_percpu_user
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_sys
-r--r--r--  1 root root   0 Nov 20 01:25 cpuacct.usage_user
-rw-r--r--  1 root root   0 Nov 20 00:29 cpu.cfs_period_us
-rw-r--r--  1 root root   0 Nov 20 00:29 cpu.cfs_quota_us
-rw-r--r--  1 root root   0 Nov 20 00:29 cpu.shares
-r--r--r--  1 root root   0 Nov 20 01:25 cpu.stat
-rw-r--r--  1 root root   0 Nov 20 01:25 notify_on_release
-rw-r--r--  1 root root   0 Nov 20 01:25 release_agent
drwxr-xr-x 64 root root   0 Nov 20 00:29 system.slice/
```

CPU子系统简介

目录下的文件大致可以分为两类：

1. 具有资源限制功能的参数：

- cpu.shares: cgroup对时间的分配。比如cgroup A设置的是1，cgroup B设置的是2，那么B中的任务获取cpu的时间，是A中任务的2倍
- cpu.cfs_period_us: 完全公平调度器的调整时间配额的周期
- cpu.cfs_quota_us: 完全公平调度器的周期当中可以占用的时间

CPU子系统简介

目录下的文件大致可以分为两类：

2. 不具备资源限制功能的参数（报告使用）

- cpuacct.usage: 该cgroup中所有任务总共使用的CPU时间（ns纳秒）
- cpuacct.stat: 该cgroup中所有任务总共使用的CPU时间，区分user和system时间
- cpuacct.usage_percpu: 该cgroup中所有任务使用各个CPU核数的时间

Cgroups限制CPU资源

在sys/fs/cpu目录下面新建一个test资源限制组

sudo mkdir test

创建成功后test目录下会自动出现相关的的文件

新建一个1.sh的文件，写入以下内容：

```
x=0
while [ True ];do
    x=$x+1
done;
```

Cgroups限制CPU资源

直接运行1.sh，cpu占用很快达到100%

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4015	adventu+	20	0	7000	3604	1596	R	100.0	0.1	0:33.11	sh
1872	adventu+	20	0	3674752	185624	81268	S	1.0	4.6	0:26.41	gnome-she+
1742	adventu+	20	0	489340	104780	47272	S	0.7	2.6	0:13.54	Xorg
2109	adventu+	20	0	892416	60540	47476	S	0.3	1.5	0:01.73	nautilus-+
2211	adventu+	20	0	816700	41480	30220	S	0.3	1.0	0:06.60	gnome-ter+
4016	adventu+	20	0	44212	3940	3232	R	0.3	0.1	0:00.08	top

将进程加入到Cgroups的cpu限制组内，具体操作为：

1. 配置cpu.cfs_period_us的值为100000
2. 配置cpu.cfs_quota_us的值为1000
3. 将1.sh这个进程pid写入tasks文件中

cpu占用率降到了1%

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4015	adventu+	20	0	13876	8840	1596	R	1.0	0.2	8:13.24	sh
1742	adventu+	20	0	489340	104780	47272	S	0.7	2.6	0:15.45	Xorg
1872	adventu+	20	0	3674752	185664	81276	S	0.7	4.6	0:29.45	gnome-she+
2211	adventu+	20	0	816700	41620	30264	S	0.7	1.0	0:08.17	gnome-ter+
555	root	0	-20	228292	7660	6612	S	0.3	0.2	0:06.52	vmtoolsd

Cgroups源码分析

- 每个进程的task_struct，都对应有一个css_set结构体
- css_set其实就是cgroup_subsys_state对象的集合
- 每个cgroup_subsys_state代表一个subsystem

```
struct task_struct
{
    #ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock */
    struct css_set __rcu *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock */
    struct list_head cg_list;
    #endif
}
```

Cgroups源码分析

/include/linux/cgroup-defs.h中css_set结构体

```
struct css_set {
    atomic_t refcount;
    struct hlist_node hlist;
    struct list_head tasks;
    struct list_head mg_tasks;
    struct list_head cgrp_links;
    struct cgroup *dfl_cgrp;
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    struct list_head mg_preload_node;
    struct list_head mg_node;
    struct cgroup *mg_src_cgrp;
    struct css_set *mg_dst_cset;
    struct list_head e_cset_node[CGROUP_SUBSYS_COUNT];
    struct list_head task_iters;
    bool dead;
    struct rcu_head rcu_head;
};
```

Cgroups源码分析

- refcount是该css_set的引用计数，一个css_set可以被多个进程共用
- hlist是嵌入的hlist_node，用于把所有的css_set组成一个hash表，这样内核可以快速查找特定的css_set
- tasks是将所有引用此css_set的进程连接成链表
- cgrp_links指向一个由struct cg_cgroup_link组成的链表
- subsys是一个指针数组，存储一组指向cgroup_subsys_state的指针，一个cgroup_subsys_state就是进程与一个特定的子系统相关的信息，通过这个指针，进程就可以获得相应的cgroups控制信息

Cgroups源码分析

cgroup_subsys_state结构

```
struct cgroup_subsys_state {
        struct cgroup *cgroup;
        atomic_t refcnt;
        unsigned long flags;
        struct css_id __rcu *id;
        struct work_struct dput_work;
};
```

- cgroup指针指向了一个cgroup结构，也就是进程属于的cgroup。进程受到子系统的控制，实际上是通过加入到特定的cgroup实现的，因为cgroup在特定的层级上，而子系统又是附加到层级上的
- 通过以上三个结构，task_struct就可以和cgroup 连接起来了：

  task_struct -> css_set -> cgroup_subsys_state -> cgroup

Cgroups源码分析

cgroup结构体：

```
struct cgroup {
        unsigned long flags;
        atomic_t count;
        int id;
        struct list_head sibling;
        struct list_head children;
        struct cgroup *parent;
        struct dentry *dentry;
        struct cgroup_name __rcu *name;
        struct cgroupfs_root *root;
        struct list_head css_sets;
        …
};
```

Cgroups源码分析

- sibling,children和parent三个嵌入的list_head负责将同一层级的cgroup连接成一颗cgroup树
- subsys是一个指针数组，存储一组指向cgroup_subsys_state的指针。
- root指向了一个cgroupfs_root的结构，即cgroup所在的层级对应的结构体。
- css_sets指向一个由struct cg_cgroup_link连成的链表，跟css_set中cg_links一样

思考题

- NameSpace是怎样做到hostname和PID隔离的？
- 为什么使用了mount namespace重新挂载/proc之后，在/proc下就看不到主机上的其它进程目录了？
- NameSpace是怎样实现网络隔离的？
- Cgroups的作用是什么？
- 系统是如何通过Cgroups控制资源使用的？其实现原理是怎样的？