**一.Namespace：#namespace是不是嵌套，如何嵌套，通过程序演示**

  Linux Namespaces机制提供一种资源隔离方案。PID,IPC,Network等系统资源不再是全局性的，而是属于特定的Namespace。每个

Namespace里面的资源对其他Namespace都是不可见的。要创建新的Namespace，只需要在调用clone时指定相应的flag。Linux

Namespaces机制为实现基于容器的虚拟化技术提供了很好的基础，容器正是利用这一特性实现了资源的隔离。不同container内的

进程属于不同的Namespace，彼此透明，互不干扰。

Linux很早就实现了一个系统调用chroot，该系统调用能够为进程提供一个限制的文件系统。chroot提供了一种简单的隔离模式：chroot

内部的文件系统无法访问外部的内容。Linux Namespace在此基础上，提供了对UTS、IPC、mount、PID、network的隔离机制＃document／namespace。txt

UTS: 包含了运行内核的名称，版本，底层体系结构的信息

IPC: 包含了所有与进程间通信有关的信息

PID: 就是进程ID

mount: 包含了文件系统的视图

NET: 网络访问

**1.task\_struct中的结构：**  
struct task\_struct {   
...   
struct nsproxy \*nsproxy;   
...   
};

将给定进程关联到所属的各个命名空间  
———————————————  
struct nsproxy {   
atomic\_t count;   
struct uts\_namespace \*uts\_ns;   
struct ipc\_namespace \*ipc\_ns;   
struct mnt\_namespace \*mnt\_ns;   
struct pid\_namespace \*pid\_ns;   
struct net \*net\_ns;   
};   
extern struct nsproxy init\_nsproxy;  
  
**2. 创建命名空间的方式**

2.1. clone创建新进程时，可以设置选项，使新进程与父进程共享命名空间，还是新进程创建一个独立的命名空间.

2.2. unshare系统调用，可以将进程的某些部分从父进程分离，其中也包括命名空间。

**3.PID**

PID命名空间按层次组织，在创建一个新的pid namespace，该命名空间中所有的pid都对父命名空间可见，但是子命名空间

看不到父命名空间的pid，因此进程在不同的pid namespace中具有不同的pid，只要能看到该进程的namespace都有一个PID。

  对于所有的进程来说，都有两种ID：一个是全局的ID（包含PID、TGID、PGRP、SID），保存在task\_struct->pid中；

一个是另局部的ID，即属于某个特定的命名空间的ID,对应task\_struct->pids数组，可以通过task\_struct->pids[pid\_type]->pid

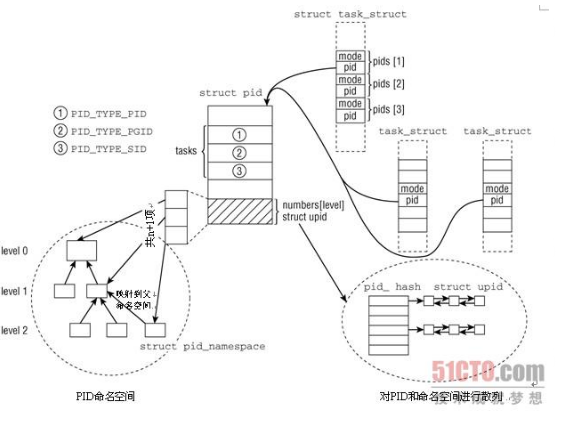
来找到对应的pid结构。pid\_type：PIDTYPE\_PID，PIDTYPE\_PGID，PIDTYPE\_SID，PIDTYPE\_MAX

struct **pid**   
{   
atomic\_t count; //计数  
unsigned int level; //对应多少namespace  
struct hlist\_head tasks[PIDTYPE\_MAX]; //指回task\_struct  
struct rcu\_head rcu; //rcu是将所有struct pid组织起来的辅助结构  
struct upid numbers[1]; //numbers成员中存储的是struct upid结构，该结构是pid与pid\_namespace相关联的结构。  
}

**upid**  
struct**upid** {   
int nr;   
struct pid\_namespace \*ns;   
struct hlist\_node pid\_chain; ＃内核链表  
};

所有的upid都保存在一个散列表中，通过upid->pid\_chain组织。

static struct hlist\_head \*pidhash;  
};  
**pid\_namespace**  
struct **pid\_namespace** {   
struct kref kref;   
struct pidmap pidmap[PIDMAP\_ENTRIES]; //保存该namespace中pid的分配情况  
int last\_pid; //保存上一个分配的pid  
struct task\_struct \*child\_reaper; //每个namespace都有一个进程来扮演Linux中init进程的角色,child\_reaper指向这个进程  
struct kmem\_cache \*pid\_cachep;   
unsigned int level; //表示该namespace在整个命名空间的层次  
struct pid\_namespace \*parent; //父namespace  
struct vfsmount \*proc\_mnt; ＃proc文件系统  
struct bsd\_acct\_struct \*bacct; ＃？  
};



**C代码实现：**

Pid namespace代码  
#define \_GNU\_SOURCE  
#include<sched.h>  
#include<unistd.h>  
#include<stdlib.h>  
#include<sys/wait.h>  
#include<string.h>  
#include<signal.h>  
#include<stdio.h>  
#include<limits.h>  
#include<sys/mount.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#define errExit(msg) do{ perror(msg);exit(EXIT\_FAILURE);} while (0)  
#define STACK\_SIZE (1024 \* 1024)  
  
static char child\_stack[STACK\_SIZE];  
  
static int childFunc(void \*arg){  
static int first\_call = 1;  
long level = (long) arg;  
  
if (!first\_call) {  
char mount\_point[PATH\_MAX];  
snprintf(mount\_point, PATH\_MAX,"/proc%c", (char) ('0' + level));   
mkdir(mount\_point, 0555); /\* 创建文件夹，mount\_point表示路径，0555表示权限 \*/  
  if (mount("proc",mount\_point, "proc", 0, NULL) == -1)  
errExit("mount");  
printf("Mounting procfs at %s\n",mount\_point);  
}  
first\_call = 0;  
if (level > 0) {  
level--;  
pid\_t child\_pid;  
child\_pid = clone(childFunc,  
child\_stack +STACK\_SIZE,  
CLONE\_NEWPID | SIGCHLD,(void \*) level);  
if (child\_pid == -1)  
errExit("clone");  
if (waitpid(child\_pid, NULL, 0) ==-1) /\* Wait for child \*/  
  errExit("waitpid");  
} else {  
printf("Final childsleeping\n");  
execlp("sleep","sleep", "1000", (char \*) NULL);  
errExit("execlp");  
}  
return 0;  
}  
  
int main(int argc,char \*argv[]){  
long levels;  
levels = (argc > 1) ? atoi(argv[1]) : 5;  
childFunc((void \*) levels);  
exit(EXIT\_SUCCESS);  
}  
PID命名空间是以父子关系层次嵌套的。在一个PID命名空间内，可以看同一个命名空间中的所有其它进程以及子进程。

运行结果如下图：

