namespace 命名空间-之应用

shichaog@126.com

由于各种原因，用户空间命名空间的实现算是一个里程碑了，首先是历时五年最复杂命名空间之一的 user 命名空间开发完结，其次是内核绝大多数命名空间已经实现了，当前的命名空间可以说是一个稳定版本了，但这并不意味着命名空间开发工作的完结，新的命名空间可能被添加进内核，也可能需要对当前的命名空间做一些扩展，例如内核日志的隔离。最后，就当前 user 命名空间的不同使用方法而言，可以说是“游戏规则”的改变：从 3.8 版本开始，非特权进程可以创建 namespace，并且其拥有特权权限，下面将会展示编程中如何使用命名空间的 API。

命名空间

目前 Linux 实现了六种类型的 namespace，每一个 namespace 是包装了一些全局系统资源的抽象集合，这一抽象集合使得在进程的命名空间中可以看到全局系统资源。命名空间的一个总体目标是支持轻量级虚拟化工具 container 的实现，container 机制本身对外提供一组进程，这组进程自己会认为它们就是系统唯一存在的进程。

在下面的讨论中，按命名空间实现的版本先后依次对其介绍，当提到命名空间的 API(clone(), ushare(),setns())时括号内的 CLONE_NEW*用于标识命名空间的类型。

mount 命名空间（CLONE_NEWS, Linux2.4.19）用于隔离一组进程看到的文件系统挂载点集合，即处于不同 mount 命名空间的进程看到的文件系统层次很可能是不一样的。mount()和 umount()系统调用的影响不再是全局的而只影响其调用进程指向的命名空间。

mount 命名空间的一个应用类似 chroot，然而和 chroot()系统调用相比，mount 命名空间在安全性和扩展性上更好。其它一些更复杂的应用如：不同的 mount 命名空间可以建立主从关系，这样可以让一个命名空间的事件自动传递到另一个命名空间。

mount 命名空间是 Linux 内核最早实现的命名空间，于 2002 年就开始了；这就是 CLONE_NEWS 的由来，当时没人想到其它不同的命名空间会被添加到内核。

UTS 命名空间（CLONE_NEWUTS， Linux2.6.19）隔离了两个系统变量，系统节点名和域名；uname()系统调用返回 UTS，名字使用 setnodename()和 setdomainname()系统调用设置。从容器的上下文看，UTS 赋予了每个容器各自的

主机名和网络信息服务名(NIS) (Network Information Service)，这使得初始化和配置脚本能够根据不同的名字进行裁剪。UTS 源于传递给 uname()系统调用的参数：struct utsname。该结构体的名字源于"UNIX Time-sharing System"。

IPC namespaces (CLONE_NEWIPC, Linux 2.6.19)隔离进程间通信资源，具体来说就是 System V IPC objects and (since Linux 2.6.30) POSIX message queues；这些机制的共同特点是由其特点而非文件系统路径名标识。每一个 IPC 命名空间尤其自己的 System V IPC 标识符和 POSIX 消息队列文件系统。

PID namespaces (CLONE_NEWPID, Linux 2.6.24)隔离进程 ID 号命名空间，话句话说就是位于不同进程 ID 命名空间的进程可以有相同的进程 ID 号，PID 命名空间的最大的好处是在主机之间移植 container 时，可以保留 container 内的 ID 号，PID 命名空间允许每个 container 拥有自己的 init 进程（ID=1），init 进程是所有进程的祖先，负责系统启动时的初始化和作为孤儿进程的父进程。

从特殊的角度来看 PID 命名空间，就是一个进程有两个 ID，一个 ID 号属于 PID 命名空间，一个 ID 号属于 PID 命名空间之外的主机系统，此外，PID 命名空间能够被嵌套。

Network namespaces (CLONE_NEWNET, Linux2.6.24 开始结束于 Linux 2.6.29)用于隔离和网络有关的资源，这就使得每个网络命名空间有其自己的网络设备、IP 地址、IP 路由表、/proc/net 目录、端口号等等。

从网络命名空间的角度看，每个 container 拥有其自己的网络设备（虚拟的）和用于绑定自己网络端口号的应用程序。主机上合适的路由规则可以将网络数据包和特定 container 相关的网络设备关联。例如，可以有多个 web 服务器，分别存在不同的 container 中，这就使得这些 web 服务器可以在其命名空间中绑定 80 端口号。

User namespaces (CLONE_NEWUSER, 起始于 Linux2.6.23 完成于 Linux 3.8) 隔离用户和组 ID 空间，换句话说，一个进程的用户和组 ID 在用户命名空间之外可以不同于命名空间之内的 ID，最有趣的是一个用户 ID 在命名空间之外非特权，而在命名空间内却可以是具有特权的。这就意味着在命名空间内拥有全部的特权权限，在命名空间之外则不是这样。

自 Linux3.8 开始，非特权进程可以创建用户命名空间，由于非特权进程在 user 命名空间内具有 root 权限，命名空间内非特权应用程序可以使用以前只有 root 能够使用的一些功能。

总结

自从第一个命名空间的实现到现在已有十年之久，命名空间的概念也发展为更通用的框架-隔离先前系统级的全局资源。结果使能命名空间能够提供完整的

轻量级虚拟化系统，呈现的形式就是 container。随着命名空间概念的扩展，与之相关的 clone()系统调用和一两个/proc 下的文件发展成许多其它的系统调用和/proc 下更多的文件。这些扩展后的 API 成为本文接下来讨论的主题。

系列博文索引

下面的列表给出了后续的系列博文。

Part 2: the namespaces API

demo_uts_s.c:示例了 UTS 命名空间的使用方法。

ns_exec.c: 使用 setns()关联一个命名空间并且执行该命令。

unshare.c: 停止命名空间的共享并执行命令。

Part 3: PID namespaces

pidns_init_sleep.c:证明 PID 命名空间

multi_pidns.c:嵌套的 PID 命名空间中创建一系列的子进程。

Part 4: more on PID namespaces

ns_child_exec.c: 创建一个子进程在新的命名空间中执行命令

simple_init.c: 简单的 init 类型的程序，用于在 PID 命名空间中使用。

orphan.c: 证明一个子进程变为孤儿进程后，将被 init 进程收留

ns_run.c: 使用 setns()关联一个或多个命名空间，并且在这些命名空间中执行一个命令，使用场景很可能是在子进程中。

Part 5: user namespaces

demo_userns.c: 创建命名空间的简单程序，并且显示了程序的权限和能力

userns_child_exec.c: 创建一个在一个新的命名空间执行 shell 命令的子进程，类似于 ns_child_exec.c，但是提供了用户命名空间的额外选项。

Part 6: more on user namespaces

userns_setns_test.c:从连个不同的 user 命名空间测试 setns()参数。

Part 7: network namespaces

命名空间 API

一个命名空间包含一些抽象化的全局系统资源，这些隔离的全局资源在命名空间将呈现给进程。命名空间被用于很多目的，最突出的就是轻量级虚拟化容器（container）了。

命名空间 API 包括三个系统调用：clone()、unshare()和 setns()，此外，还包括/proc 目录下的许多文件。本文将讨论上述系统调用以及/proc 目录下的一些文件。为了明确使用的命名空间的类型，三个系统调用使用先前提到的 CLONE_NEW*常量：

CLONE_NEWIPC, CLONE_NEWNS, CLONE_NEWNET, CLONE_NEWPID, CLONE_NEWUSER, and CLONE_NEWUTS。

创建一个新的命名空间： clone（）

创建一个命名空间的方法是使用 clone()系统调用，其会创建一个新的进程。为了说明创建的过程，给出 clone()的原型如下：

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

本质上，clone()是一个通用的 fork()版本，fork()的功能由 flags 参数控制。总的来说，约有超过 20 个不同的 CLONE_*标志控制 clone()提供不同的功能，包括父子进程是否共享如虚拟内存、打开的文件描述符、子进程等一些资源。如调用 clone 时设置了一个 CLONE_NEW*标志，一个与之对应的新的命名空间将被创建，新的进程属于该命名空间。可以使用多个 CLONE_NEW*标志的组合。

我们的例子(demo_uts_namespace.c)调用 clone()时设置了 CLONE_NEWUTS 标志以创建一个 UTS 命名空间。完整的 **demo_uts_namespaces.c**

```
/* demo_uts_namespaces.c
   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later
   Demonstrate the operation of UTS namespaces.
*/
#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/utsname.h>
#include <sched.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* A simple error-handling function: print an error message based
   on the value in 'errno' and terminate the calling process */
```

```
#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                                 } while (0)

static int                  /* Start function for cloned child */
childFunc(void *arg)
{
        struct utsname uts;

        /* Change hostname in UTS namespace of child */

        if (sethostname(arg, strlen(arg)) == -1)
                errExit("sethostname");

        /* Retrieve and display hostname */

        if (uname(&uts) == -1)
                errExit("uname");
        printf("uts.nodename in child:    %s\n", uts.nodename);

        /* Keep the namespace open for a while, by sleeping.
           This allows some experimentation--for example, another
           process might join the namespace. */

        sleep(100);

        return 0;               /* Terminates child */
}

#define STACK_SIZE (1024 * 1024)        /* Stack size for cloned child */

static char child_stack[STACK_SIZE];

int main(int argc, char *argv[])
{
        pid_t child_pid;
        struct utsname uts;

        if (argc < 2) {
                fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        /* Create a child that has its own UTS namespace;
           the child commences execution in childFunc() */
```

```
    child_pid = clone(childFunc,
                      child_stack + STACK_SIZE,     /* Points to start of
                                                       downwardly growing stack */
                      CLONE_NEWUTS | SIGCHLD, argv[1]);
    if (child_pid == -1)
        errExit("clone");
    printf("PID of child created by clone() is %ld\n", (long) child_pid);

    /* Parent falls through to here */

    sleep(1);                   /* Give child time to change its hostname */

    /* Display the hostname in parent's UTS namespace. This will be
       different from the hostname in child's UTS namespace. */

    if (uname(&uts) == -1)
        errExit("uname");
    printf("uts.nodename in parent: %s\n", uts.nodename);

    if (waitpid(child_pid, NULL, 0) == -1)          /* Wait for child */
        errExit("waitpid");
    printf("child has terminated\n");

    exit(EXIT_SUCCESS);
}
```

例程需要一个命令行参数，当运行时，其创建一个在新的 UTS 命名空间执行的子进程。在新的命名空间中，子进程使用获得的命令行参数更改主机名。

main 函数第一个比较重要的部分是创建子进程的 clone()调用。

```
    child_pid = clone(childFunc,
                      child_stack + STACK_SIZE,     /* Points to start of
                                                       downwardly growing stack */
                      CLONE_NEWUTS | SIGCHLD, argv[1]);
    printf("PID of child created by clone() is %ld\n", (long) child_pid);
```

新的子进程将会执行用户定义的函数 childFunc()；该函数会接收 clone()传递的最后一个参数 argv[1]，由于创建时使用了 CLONE_NEWUTS 标识，新的 UTS 命名空间会被创建。

主程序然后休眠一段时间。这是一个粗暴的方式以让子进程修改 UTS 命名空间内的主机名。那个程序然后使用 uname()获得命名空间内的主机名并且显示主机名。

```
    sleep(1);                   /* Give child time to change its hostname */
    if (uname(&uts) == -1)
```

```
        errExit("uname");
    printf("uts.nodename in child:    %s\n", uts.nodename);
```

与此同时，子进程执行的函数 childFunc()首先更改主机名为命令行输入的参数，然后显示修改后的主机名。

```
    if (uname(&uts) == -1)
        errExit("uname");
    printf("uts.nodename in child:    %s\n", uts.nodename);
```

在结束之前，子进程休眠一会。效果就是让子进程的 UTS 命名空间存在一段时间，这段时间使得我们能够做一些后面我们给出的实验。

执行如下命令：

```
$ su                # Need privilege to create a UTS namespace
Password:
# uname -n
antero
# ./demo_uts_namespaces bizarro
PID of child created by clone() is 27514
uts.nodename in child:   bizarro
uts.nodename in parent: antero
```

正如其它命名空间（user namespace 除外），创建一个 UTS 命名空间需要特权权限(CAP_SYS_ADMIN)。这可以避免 set-user-ID 类的应用程序因系统主机问题被误导而执行错误的事。

另外一个可能性是 set-user-ID 类应用可能使用主机名作为锁文件的一部分。如果一个非特权用户能够在一个具有任何主机名的 UTS 命名空间运行程序，这可能导致应用受到各种攻击。最简单的，这将使锁无效，在不同的 UTS 命名空间中引发应用实例被运行。另外，一个恶意用户可以在一个 UTS 命名空间运行一个 set-user-ID 应用程序来覆盖一个重要文件的锁。

The /proc/PID/ns 文件

每一个进程有一个/proc/PID/ns 目录，该目录下每一个命名空间对应一个文件。从 3.8 版本起，每一个这类文件都是一个特殊的符号链接。该符号链接提供在其命名空间上执行某个操作的某种方法。

```
$ ls -l /proc/$$/ns         # $$ is replaced by shell's PID
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 user -> user:[4026531837]
lrwxrwxrwx. 1 mtk mtk 0 Jan   8 04:12 uts -> uts:[4026531838]
```

这些符号链接可以用来判断两个命名空间是否在同一个命名空间。如果两个进程在同一个命名空间，内核会保证由/proc/PID/ns 导出的 inode 号将会是一样的。inode 号可以通过 stat()系统调用获得。

然而，内核同样为每个 /proc/PID/ns 构建了符号链接，以使其指向包含标识命名空间类型字符串（字符串以 inode 号结尾）的名字。我们可以通过 ls -l 或者 readlink 命令查看名字。

让我们回到上面 demo_uts_namespaces 运行的 shell 会话，通过查看父子进程的/proc/PID/ns 符号链接信息可以知道它们是否位于同一个命名空间。

```
^Z                          # Stop parent and child
[1]+  Stopped                 ./demo_uts_namespaces bizarro
# jobs -l                   # Show PID of parent process
[1]+ 27513 Stopped            ./demo_uts_namespaces bizarro
# readlink /proc/27513/ns/uts   # Show parent UTS namespace
uts:[4026531838]
# readlink /proc/27514/ns/uts   # Show child UTS namespace
uts:[4026532338]
```

正如我们看到的， /proc/PID/ns/uts 符号链接并不一样，表明它们位于不同的命名空间中。

/proc/PID/ns 同样服务于其它目的，如果我们随便打开一个文件，那么只要文件描述打开状态，那么命名空间将会保持存在而不论命名空间中的进程是否全部退出，相同的效果可以通过绑定其中一个符号链接到文件系统的其它地方获得。

```
# touch ~/uts                 # Create mount point
# mount --bind /proc/27514/ns/uts ~/uts
```

在 3.8 之前， /proc/PID/ns 下的文件是硬链接，并且只有 ipc、net 和 uts 文件是存在的。

关联一个存在命名空间：setns()

当一个命名空间没有进程时还保持其打开，这么做是为了后续添加进程到该命名空间。而添加这个功能这就是使用 setns()系统调用来完成了，这使得调用的进程能够和命名空间关联：

```
int setns(int fd, int nstype);
```

准确来说，setns()将调用的进程和一个特定的命名空间解除关系并将该进程和一个同类型的命名空间相关联。

fd 参数指明了关联的命名空间，其是指向了 /proc/PID/ns 目录下一个符号链接的文件描述符，可以通过打开这些符号链接指向的文件或者打开一个绑定到符号链接的文件来获得文件描述符（所谓的获得指的是引用计数加 1）。

nstype 参数运行调用者检查 fd 指向的命名空间的类型，如果这个参数等于

零，将不会检查。当调用者已经知道命名空间的类型时这会很有用。我们的示例程序(ns_exec.c)的 nstype 参数等于零，其适用于任何命名空间。当 nstype 被赋值为 CLONE_NEW*的常量时，内核会检查 fd 指向的命名空间的类型。

使用 setns()和 execve()（或者其它的 exec()函数）使得我们能够构建一个简单但是有用的工具，一个和特定命名空间关联的程序并且在命名空间中可以执行一个命令。

```
/* ns_exec.c

    Copyright 2013, Michael Kerrisk
    Licensed under GNU General Public License v2 or later

    Join a namespace and execute a command in the namespace
*/
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

/* A simple error-handling function: print an error message based
    on the value in 'errno' and terminate the calling process */

#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                                } while (0)

int
main(int argc, char *argv[])
{
      int fd;

      if (argc < 3) {
            fprintf(stderr, "%s /proc/PID/ns/FILE cmd [arg...]\n", argv[0]);
            exit(EXIT_FAILURE);
      }

      fd = open(argv[1], O_RDONLY);     /* Get descriptor for namespace */
      if (fd == -1)
            errExit("open");

      if (setns(fd, 0) == -1)               /* Join that namespace */
            errExit("setns");
```

```
    execvp(argv[2], &argv[2]);          /* Execute a command in namespace */
    errExit("execvp");
}
```

我们的应用程序接收两个命令行参数。第一个参数是/proc/PID/ns/* 符号链接的路径，剩下的参数是在和第一参数关联的命名空间中将要运行的程序名，程序中的关键步骤如下：

```
fd = open(argv[1], O_RDONLY);    /* Get descriptor for namespace */
setns(fd, 0);                    /* Join that namespace */
execvp(argv[2], &argv[2]);       /* Execute a command in namespace */
```

一个在命名空间中执行的有意思的程序当然是 shell 程序。我么可以使用先前创建的 UTS 命名空间以及 ns_exec 程序来在 demo_uts_namespaces.c 创建的新的 UTS 命名空间中执行一个 shell。

```
# ./ns_exec ~/uts /bin/bash      # ~/uts is bound to /proc/27514/ns/uts
My PID is: 28788
```

可以证明新的 UTS 命名空间创建的 shell 是 emo_uts_namespaces 的子进程，可以通过查看主机名或者比较 /proc/PID/ns/uts 的 inode 节点得到该结论。

```
# hostname
bizarro
# readlink /proc/27514/ns/uts
uts:[4026532338]
# readlink /proc/$$/ns/uts       # $$ is replaced by shell's PID
uts:[4026532338]
```

早期的内核版本，使用 setns()关联 mount、PID 和 user 命名空间是不可能的，但是从 3.8 开始支持所有的命名空间类型。

Leaving a namespace: unshare()

最后一种命名空间的系统调用是 unshare()：

```
int unshare(int flags);
```

unshare()系统调用提供类似 clone()的功能，但是作用于调用的进程。其会创建由 flags 参数中制定的 CLONE_NEW*命名空间，并且将调用者作为命名空间的一部分。unshare()的主要目的是消除命名空间的副作用而不需要创建新的进程或线程。

撇开 clone 系统调用的影响，调用的形式是：

```
clone(..., CLONE_NEWXXX, ....);
```

就命名空间术语来说，等价于下列顺序：

```
if (fork() == 0)
    unshare(CLONE_NEWXXX);      /* Executed in the child process */
```

unshare 的系统调用的一个例子是在命令行下使用 unshare 命令，其允许用户使用 shell 执行另一个命名空间的命令。该命令的通常形式如下：

```
unshare [options] program [arguments]
```

参数[arguments] 是传递给命令 program 的，options 传递给 unshare 指向的命名空间。

实现 unshare 命令的关键步骤很直接：

```
/* Code to initialize 'flags' according to command-line options
   omitted */
unshare(flags);
/* Now execute 'program' with 'arguments'; 'optind' is the index
   of the next command-line argument after options */
execvp(argv[optind], &argv[optind]);
```

一个简单的 unshare 命令的实现代码如下：

```
/* unshare.c
   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later
   A simple implementation of the unshare(1) command: unshare
   namespaces and execute a command.
*/

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

/* A simple error-handling function: print an error message based
   on the value in 'errno' and terminate the calling process */

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static void
```

```
usage(char *pname)
{
        fprintf(stderr, "Usage: %s [options] program [arg...]\n", pname);
        fprintf(stderr, "Options can be:\n");
        fprintf(stderr, "        -i      unshare IPC namespace\n");
        fprintf(stderr, "        -m      unshare mount namespace\n");
        fprintf(stderr, "        -n      unshare network namespace\n");
        fprintf(stderr, "        -p      unshare PID namespace\n");
        fprintf(stderr, "        -u      unshare UTS namespace\n");
        fprintf(stderr, "        -U      unshare user namespace\n");
        exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
        int flags, opt;
        flags = 0;

        while ((opt = getopt(argc, argv, "imnpuU")) != -1) {
                switch (opt) {
                case 'i': flags |= CLONE_NEWIPC;        break;
                case 'm': flags |= CLONE_NEWNS;         break;
                case 'n': flags |= CLONE_NEWNET;        break;
                case 'p': flags |= CLONE_NEWPID;        break;
                case 'u': flags |= CLONE_NEWUTS;        break;
                case 'U': flags |= CLONE_NEWUSER;       break;
                default:    usage(argv[0]);
                }
        }
        if (optind >= argc)
                usage(argv[0]);
        if (unshare(flags) == -1)
                errExit("unshare");
```

```
        execvp(argv[optind], &argv[optind]);
        errExit("execvp");
    }
```

在下面的 shell 会话中，我们使用 unshare.c 程序在另外一个 mount 命名空间中执行一个 shell。

```
    # echo $$                    # Show PID of shell
    8490
    # cat /proc/8490/mounts | grep mq     # Show one of the mounts in namespace
    mqueue /dev/mqueue mqueue rw,seclabel,relatime 0 0
    # readlink /proc/8490/ns/mnt          # Show mount namespace ID
    mnt:[4026531840]
    # ./unshare -m /bin/bash              # Start new shell in separate mount namespace
    # readlink /proc/$$/ns/mnt            # Show mount namespace ID
    mnt:[4026532325]
```

从上述 readlink 输出可以看到两个 shell 属于不同的命名空间，改变一个命名空间的挂载点，然后查看另外一个命名空间对上述改变是否可见即可分辨它们是否位于同一个命名空间。

```
    # umount /dev/mqueue                  # Remove a mount point in this shell
    # cat /proc/$$/mounts | grep mq       # Verify that mount point is gone
    # cat /proc/8490/mounts | grep mq     # Is it still present in the other namespace?
    mqueue /dev/mqueue mqueuerw,seclabel,relatime 0 0
```

从输出的最后两个参数看到， /dev/mqueue 挂载点一个命名空间可以看到而另一个命名空间看不到。

总结

本文我们查看了命名空间 API 以及它们的使用。接下来的文章，我们将更深入查看命名空间的，特别会深入查看 PID 和 user 命名空间。

PID 命名空间

被 PID 命名空间隔离的全局资源是进程 ID 号空间，这就意味着位于不同命名空间的进程的 ID 号可以相同，PID 命名空间被用来实现 container。

和传统 Linux 系统一样，在 PID 命名空间内的进程 ID 号是各不相等的，它们被从 1 开始分配进程 ID 号。同样的，ID 号等于 1 的 init 进程是一个特殊进程，它是命名空间中的第一个进程，它也命名空间提供一些管理工作。

初探

一个新的 PID 命名空间调用 clone(...CLONE_NEWPID...)创建，我们将展示一个简单的使用 clone 创建 PID 命名空间的例子，并且使用该例子阐释 PID 命名空间的基本概念，完整的 pidns_init_sleep.c 的源码如下：

```
/* pidns_init_sleep.c

   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later

   A simple demonstration of PID namespaces.
*/
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <signal.h>
#include <stdio.h>

/* A simple error-handling function: print an error message based
   on the value in 'errno' and terminate the calling process */

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static int                      /* Start function for cloned child */
childFunc(void *arg)
{
    printf("childFunc(): PID  = %ld\n", (long) getpid());
    printf("childFunc(): PPID = %ld\n", (long) getppid());
```

```
    char *mount_point = arg;

    if (mount_point != NULL) {
        mkdir(mount_point, 0555);           /* Create directory for mount point */
        if (mount("proc", mount_point, "proc", 0, NULL) == -1)
            errExit("mount");
        printf("Mounting procfs at %s\n", mount_point);
    }

    execlp("sleep", "sleep", "600", (char *) NULL);
    errExit("execlp");    /* Only reached if execlp() fails */
}

#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];        /* Space for child's stack */

int
main(int argc, char *argv[])
{
    pid_t child_pid;

    child_pid = clone(childFunc,
                      child_stack + STACK_SIZE,      /* Points to start of
                                                        downwardly growing stack */
                      CLONE_NEWPID | SIGCHLD, argv[1]);

    if (child_pid == -1)
        errExit("clone");

    printf("PID returned by clone(): %ld\n", (long) child_pid);

    if (waitpid(child_pid, NULL, 0) == -1)          /* Wait for child */
```

```
            errExit("waitpid");


        exit(EXIT_SUCCESS);
    }
```

主程序使用 clone()创建一个 PID 命名空间，并且显示了返回的 PID 号。

```
child_pid = clone(childFunc,
            child_stack + STACK_SIZE,   /* Points to start of
                                        downwardly growing stack */
            CLONE_NEWPID | SIGCHLD, argv[1]);
    printf("PID returned by clone(): %ld\n", (long) child_pid);
```

新的子进程开始执行 childFunc()函数，该函数接收 clone()调用的最后一个参数 argv[1]，该参数的作用后续讲解。 childFunc()函数显示进程和其父进程 ID，并且使用 sleep 函数结束。

```
    printf("childFunc(): PID = %ld\n", (long) getpid());
    printf("ChildFunc(): PPID = %ld\n", (long) getppid());
    ...
    execlp("sleep", "sleep", "1000", (char *) NULL);
```

使用 sleep 的主要价值在于其让我们区分子进程和父进程更简单。当我们允许该程序， 第一行的输出如下：

```
    $ su        # Need privilege to create a PID namespace
    Password:
    # ./pidns_init_sleep /proc2
    PID returned by clone(): 27656
    childFunc(): PID  = 1
    childFunc(): PPID = 0
    Mounting procfs at /proc2
```

pidns_init_sleep 前两行输出了从两个不同的 PID 命名空间查看子进程的 ID 号：clone()调用所在的 PID 命名空间和子进程存在的 PID 命名空间。即子进程有两个 PID：父进程命名空间中的为 27656，新创建的命名空间中的 ID 是 1。输出的下一行展示了子进程的父进程 ID。父进程 ID 等于 0 显示了命名空间实现的一点怪异之处。正如下面我们讨论的，命名空间形成一个层次结构：一个进程仅仅能够"看见"在其命名空间内和子命名空间内的进程 ID（子进程的子进程...均能看见），子进程看不见父进程命名空间的进程。由于由 clone()创建的子进程同父进程在不同的命名空间中，所有子进程“看”不到父进程；因此，getppid()返回的父

进程 ID 为 0.

为了解释最后一行，我们需要重新查看 childFunc() 中跳过的一些代码段。

/proc/PID 和 PID 命名空间

Linux 系统上的每一个进程都有一个/proc/PID 目录，该目录包括了描述进程的伪文件。这一机制可以直接得到 PID 的命名空间。在一个命名空间内部，/proc/PID 目录仅包含在该命名空间内和子命名空间。

然而，为了使和一个 PID 命名空间相关的/proc/PID 目录可见，proc 文件系统("procfs")需要在 PID 命名空间内被挂载。在一个命名空间内运行的 shell 上，我们可以使用 mount 命令完成：

```
#mount -t proc proc/mount_point
```

也可以使用 mount()系统调用完成，这里在 childFunc()里调用如下：

```
mkdir(mount_point, 0555);       /* Create directory for mount point */
mount("proc", mount_point, "proc", 0, NULL);
printf("Mounting procfs at %s\n", mount_point);
```

mount_pioint 参数在运行 pidns_init_sleep 时通过命令行参数给出。

在我们的例子中，shell 中运行 pidns_init_sleep，我们在/proc2 目录下挂着 procfs。在现实世界中，procfs 通常被挂载在/proc 目录下。然而，我们的例子在/proc2 目录下挂载 procfs，这避免了会给系统上其它进程带来问题：因为他们位于同样的挂载点，更改挂载在/proc 目录下的文件系统将使得 root PID 命名空间“看”不到/proc/PID 目录。

所以，在我们的 shell 会话中，/proc 目录下挂载的 procfs 将显示从父 PID 命名空间能够看到的进程的 PID 子目录，/proc2 则用于子进程命名空间。需要提醒的是虽然子进程 PID 命名空间的进程能够看见由/proc 挂载点导出的 PID 目录，但是这些 PID 目录是对于子进程 PID 命名空间的进程而言是无意义的，因为这些进程的系统调用只能看到它们所在命名空间的 PID。

如果我们想让像 ps 这样的工具在一个子进程中能够正确运行,那么在/proc 挂载点挂载一个 procfs 文件系统还是必要的。因为这些工具的信息源于/proc 目录。有两种方法在不影响父进程使用的 PID 命名空间前提下达到这个目标。其一，如果子进程使用 CLONE_NEWNS 标志创建，那么子进程将和系统的其它部分在不同的 mount 命名空间，在这种情况下，在/proc 目录下挂载 procfs 不会产生任何问题。另外，不采用 CLONE_NEWNS 的方法，子进程可以使用 chroot()并在/proc 目录下挂载 procfs。

让我们回到运行 pidns_init_sleep 程序的 shell 上，我们停止该程序并在 fu2 命名空间中使用 ps 检查父子进程的一些信息。

```
^Z                      Stop the program, placing in background
[1]+  Stopped                 ./pidns_init_sleep /proc2
# ps -C sleep -C pidns_init_sleep -o "pid ppid stat cmd"
  PID  PPID STAT CMD
27655 27090 T    ./pidns_init_sleep /proc2
27656 27655 S    sleep 600
```

PPID 的值为 27655，最后一行系显示了 sleep 是在父进程中执行的。

通过使用 readlink 命令查看父子进程的不同/proc/PID/ns/pid 符号链接信息，我们可以看到两个进程在不同的 PID 命名空间中：

```
# readlink /proc/27655/ns/pid
pid:[4026531836]
# readlink /proc/27656/ns/pid
pid:[4026532412]
```

到此，我们可以使用新挂载的 procfs 获得新 PID 命名空间的进程信息。我们可以使用下面的命令获得 PID 的列表：

```
# ls -d /proc2/[1-9]*
/proc2/1
```

正如我们看到的，PID 命名空间仅仅包括一个进程，它的进程 ID 号是 1。同样也可以使用/proc/PID/status 文件作为一个获得一个进程信息的不同方法。

```
# cat /proc2/1/status | egrep '^(Name|PP*id)'
Name:   sleep
Pid:    1
PPid:   0
```

PPid 是 0，符合前面 getppid()系统调用的返回的父进程的 ID 号。

嵌套的 PID 命名空间

如前面提到的，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后裔进程。这里，“看见”意思是能够在特定的进程 ID 号上使用系统调用，一个子 PID 命名空间不能看见父 PPID 的命名空间。

一个进程在 PID 命名空间的每一层都有一个进程 ID，存在的范围是该进程的 PID 命名空间一直到 root 命名空间。getpid()总是返回 PID 命名空间内的进程 ID。

我们可以使用 multi_pidns.c 来展示一个进程在不同的命名空间中拥有不同的进程 ID 号。为了简洁，我们简单阐述该程序都做了什么。

```
/* multi_pidns.c

   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later

   Create a series of child processes in nested PID namespaces.
*/
#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>

/* A simple error-handling function: print an error message based
   on the value in 'errno' and terminate the calling process */

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];    /* Space for child's stack */
                        /* Since each child gets a copy of virtual memory, this
                           buffer can be reused as each child creates its child */

/* Recursively create a series of child process in nested PID namespaces.
   'arg' is an integer that counts down to 0 during the recursion.
```

```
    When the counter reaches 0, recursion stops and the tail child
    executes the sleep(1) program. */

static int
childFunc(void *arg)
{
    static int first_call = 1;
    long level = (long) arg;

    if (!first_call) {

        /* Unless this is the first recursive call to childFunc()
           (i.e., we were invoked from main()), mount a procfs
           for the current PID namespace */

        char mount_point[PATH_MAX];

        snprintf(mount_point, PATH_MAX, "/proc%c", (char) ('0' + level));

        mkdir(mount_point, 0555);           /* Create directory for mount point */
        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) {

        /* Recursively invoke childFunc() to create another child in a
           nested PID namespace */

        level--;
        pid_t child_pid;
```

```
        child_pid = clone(childFunc,
                        child_stack + STACK_SIZE,      /* Points to start of
                                                          downwardly growing stack */
                        CLONE_NEWPID | SIGCHLD, (void *) level);

        if (child_pid == -1)
            errExit("clone");

        if (waitpid(child_pid, NULL, 0) == -1)    /* Wait for child */
            errExit("waitpid");

    } else {

        /* Tail end of recursion: execute sleep(1) */

        printf("Final child sleeping\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);
}
```

该程序递归创建一系列的子进程命名空间，命令行参数指明了子进程和 PID 命名空间的创建次数：

#./multi_pidns 5

除了递归创建子进程，每一个递归步骤还会在一个独一无二的挂载点挂载 procfs 文件系统。递归最后的子进程执行 sleep()系统调用。上面的命令行产生如下的输出：

```
Mounting procfs at /proc4
Mounting procfs at /proc3
Mounting procfs at /proc2
Mounting procfs at /proc1
Mounting procfs at /proc0
Final child sleeping
```

在每一个 procfs 下查看 PID，我们看到越是后创建的 procfs 包含的 PID 越少，表明每一个 PID 命名空间只显示其命名空间自身以及其后创建的命名空间的信息。

```
^Z                      Stop the program, placing in background
[1]+  Stopped           ./multi_pidns 5
# ls -d /proc4/[1-9]*       Topmost PID namespace created by program
/proc4/1  /proc4/2  /proc4/3  /proc4/4  /proc4/5
# ls -d /proc3/[1-9]*
/proc3/1  /proc3/2  /proc3/3  /proc3/4
# ls -d /proc2/[1-9]*
/proc2/1  /proc2/2  /proc2/3
# ls -d /proc1/[1-9]*
/proc1/1  /proc1/2
# ls -d /proc0/[1-9]*       Bottommost PID namespace
/proc0/1
```

一个 grep 命令使得我们能够看见递归最后的 PID。

 # grep -H 'Name:.*sleep' /proc?/[1-9]*/status

```
/proc0/1/status:Name:       sleep
/proc1/2/status:Name:       sleep
/proc2/3/status:Name:       sleep
/proc3/4/status:Name:       sleep
/proc4/5/status:Name:       sleep
```

换句话说，嵌套最深的 PID 命名空间(/proc0)，该进程执行 sleep 并且进程 ID 是 1，在最上面创建的 PID 命名空间是/proc4，进程的 PID 是 5。

如果你运行本文的例子，需要说明的是它们将残留下挂载点和挂载目录。在结束程序时，如下的 shell 命令行完成做够的清理工作：

 # umount /proc?

 # rmdir /proc?

总结

在本文，我们了解了一些 PID 命名空间的操作。在下文中，我们将讨论 PID 命名空间的 init 进程和一些其它的 PID 命名空间的 API。

深入 PID 命名空间

本文是对 PID 命名空间的更深入探讨。PID 命名空间的一个应用是打包一组进程（container），使打包的进程组自身就像一个操作系统。传统操作系统和这里的 container 一样的一个关键点是 init 进程。所以我们来看看 init 进程以及两种情况下都有哪些不同。按惯例，我们将看看适用于 PID 命名空间的其它一些细节。

PID 命名空间的 init 进程

在 PID 命名空间中创建的第一个进程的进程 ID 是 1，该进程的角色和传统操作系统的 init 进程一样；特别地，init 进程能够完成 PID 命名空间需要的初始化工作（这些工作很可能包括启动其它进程），其同样会是孤儿进程的父进程。

为了解释 PID 命名空间，我们将使用一些服务于目的的例程。第一个例程是 ns_child_exec.c，命令行运行的语法如下：

ns_child_exec[options] command [arguments]

```
/* ns_child_exec.c

   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later

   Create a child process that executes a shell command in new namespace(s).
*/
#define _GNU_SOURCE
```

```
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>

/* A simple error-handling function: print an error message based
     on the value in 'errno' and terminate the calling process */

#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                                 } while (0)

static void
usage(char *pname)
{
       fprintf(stderr, "Usage: %s [options] cmd [arg...]\n", pname);
       fprintf(stderr, "Options can be:\n");
       fprintf(stderr, "        -i      new IPC namespace\n");
       fprintf(stderr, "        -m      new mount namespace\n");
       fprintf(stderr, "        -n      new network namespace\n");
       fprintf(stderr, "        -p      new PID namespace\n");
       fprintf(stderr, "        -u      new UTS namespace\n");
       fprintf(stderr, "        -U      new user namespace\n");
       fprintf(stderr, "        -v      Display verbose messages\n");
       exit(EXIT_FAILURE);
}

static int                      /* Start function for cloned child */
childFunc(void *arg)
{
       char **argv = arg;

       execvp(argv[0], &argv[0]);
```

```
        errExit("execvp");
}

#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];        /* Space for child's stack */

int
main(int argc, char *argv[])
{
        int flags, opt, verbose;
        pid_t child_pid;

        flags = 0;
        verbose = 0;

        /* Parse command-line options. The initial '+' character in
            the final getopt() argument prevents GNU-style permutation
            of command-line options. That's useful, since sometimes
            the 'command' to be executed by this program itself
            has command-line options. We don't want getopt() to treat
            those as options to this program. */

        while ((opt = getopt(argc, argv, "+imnpuUv")) != -1) {
                switch (opt) {
                case 'i': flags |= CLONE_NEWIPC;        break;
                case 'm': flags |= CLONE_NEWNS;         break;
                case 'n': flags |= CLONE_NEWNET;        break;
                case 'p': flags |= CLONE_NEWPID;        break;
                case 'u': flags |= CLONE_NEWUTS;        break;
                case 'U': flags |= CLONE_NEWUSER;       break;
                case 'v': verbose = 1;                  break;
                default:    usage(argv[0]);
                }
```

```
        }

        child_pid = clone(childFunc,
                          child_stack + STACK_SIZE,
                          flags | SIGCHLD, &argv[optind]);
        if (child_pid == -1)
            errExit("clone");

        if (verbose)
            printf("%s: PID of child created by clone() is %ld\n",
                    argv[0], (long) child_pid);

        /* Parent falls through to here */

        if (waitpid(child_pid, NULL, 0) == -1)          /* Wait for child */
            errExit("waitpid");

        if (verbose)
            printf("%s: terminating\n", argv[0]);
        exit(EXIT_SUCCESS);
    }
```

ns_child_exec 程序使用 clone()系统调用来创建子进程；子进程然后执行命令行中的 command 命令，命令的参数是命令行中的 argument 参数。命令行中的 option 参数用于指定要创建的命名空间类型，该参数将传递给 clone()系统调用。例如 -p 选项将指导子进程创建新的 PID 命名空间，如下所示：

```
$ su              # Need privilege to create aPID namespace
    Password:
# ./ns_child_exec -p sh -c 'echo $$'
    1
```

上面的命令行在新的 PID 命名空间中创建了一个子进程，该子进程执行一个显示 shell 进程 ID 的 echo 命令。该 shell 进程 ID 是 1，当 shell 运行时其将是 PID 命名空间的 init 进程。

我们的下一个例程，simple_init.c 是一个我们要将其作为一个 PID 命名空间的 init 进程的程序。该程序用于证实 PID 命名空间的 init 进程的一些特性。

```
/* simple_init.c
```

```
    Copyright 2013, Michael Kerrisk
    Licensed under GNU General Public License v2 or later

    A simple init(1)-style program to be used as the init program in
    a PID namespace. The program reaps the status of its children and
    provides a simple shell facility for executing commands.
*/
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <wordexp.h>
#include <errno.h>
#include <sys/wait.h>

#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                                 } while (0)

static int verbose = 0;

/* Display wait status (from waitpid() or similar) given in 'status' */

/* SIGCHLD handler: reap child processes as they change state */

static void
child_handler(int sig)
{
    pid_t pid;
    int status;

    /* WUNTRACED and WCONTINUED allow waitpid() to catch stopped and
       continued children (in addition to terminated children) */

    while ((pid = waitpid(-1, &status,
                          WNOHANG | WUNTRACED | WCONTINUED)) != 0) {
        if (pid == -1) {
            if (errno == ECHILD)            /* No more children */
                break;
            else
                perror("waitpid");          /* Unexpected error */
        }
```

```
            if (verbose)
                printf("\tinit: SIGCHLD handler: PID %ld terminated\n",
                        (long) pid);
        }
}

/* Perform word expansion on string in 'cmd', allocating and
     returning a vector of words on success or NULL on failure */

static char **
expand_words(char *cmd)
{
        char **arg_vec;
        int s;
        wordexp_t pwordexp;

        s = wordexp(cmd, &pwordexp, 0);
        if (s != 0) {
            fprintf(stderr, "Word expansion failed\n");
            return NULL;
        }

        arg_vec = calloc(pwordexp.we_wordc + 1, sizeof(char *));
        if (arg_vec == NULL)
            errExit("calloc");

        for (s = 0; s < pwordexp.we_wordc; s++)
            arg_vec[s] = pwordexp.we_wordv[s];

        arg_vec[pwordexp.we_wordc] = NULL;

        return arg_vec;
}

static void
usage(char *pname)
{
        fprintf(stderr, "Usage: %s [-q]\n", pname);
        fprintf(stderr, "\t-v\tProvide verbose logging\n");

        exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char *argv[])
{
        struct sigaction sa;
#define CMD_SIZE 10000
        char cmd[CMD_SIZE];
        pid_t pid;
        int opt;

        while ((opt = getopt(argc, argv, "v")) != -1) {
                switch (opt) {
                case 'v': verbose = 1;                  break;
                default:    usage(argv[0]);
                }
        }

        sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
        sigemptyset(&sa.sa_mask);
        sa.sa_handler = child_handler;
        if (sigaction(SIGCHLD, &sa, NULL) == -1)
                errExit("sigaction");

        if (verbose)
                printf("\tinit: my PID is %ld\n", (long) getpid());

        /* Performing terminal operations while not being the foreground
           process group for the terminal generates a SIGTTOU that stops the
           process.    However our init "shell" needs to be able to perform
           such operations (just like a normal shell), so we ignore that
           signal, which allows the operations to proceed successfully. */

        signal(SIGTTOU, SIG_IGN);

        /* Become leader of a new process group and make that process
           group the foreground process group for the terminal */

        if (setpgid(0, 0) == -1)
                errExit("setpgid");;
        if (tcsetpgrp(STDIN_FILENO, getpgrp()) == -1)
                errExit("tcsetpgrp-child");

        while (1) {

                /* Read a shell command; exit on end of file */
```

```
printf("init$ ");
if (fgets(cmd, CMD_SIZE, stdin) == NULL) {
        if (verbose)
                printf("\tinit: exiting");
        printf("\n");
        exit(EXIT_SUCCESS);
}

if (cmd[strlen(cmd) - 1] == '\n')
        cmd[strlen(cmd) - 1] = '\0';            /* Strip trailing '\n' */

if (strlen(cmd) == 0)
        continue;                   /* Ignore empty commands */

pid = fork();                   /* Create child process */
if (pid == -1)
        errExit("fork");

if (pid == 0) {                 /* Child */
        char **arg_vec;

        arg_vec = expand_words(cmd);
        if (arg_vec == NULL)            /* Word expansion failed */
                continue;

        /* Make child the leader of a new process group and
           make that process group the foreground process
           group for the terminal */

        if (setpgid(0, 0) == -1)
                errExit("setpgid");;
        if (tcsetpgrp(STDIN_FILENO, getpgrp()) == -1)
                errExit("tcsetpgrp-child");

        /* Child executes shell command and terminates */

        execvp(arg_vec[0], arg_vec);
        errExit("execvp");                  /* Only reached if execvp() fails */
}

/* Parent falls through to here */

if (verbose)
```

```
                printf("\tinit: created child %ld\n", (long) pid);

            pause();                    /* Will be interrupted by signal handler */

            /* After child changes state, ensure that the 'init' program
               is the foreground process group for the terminal */

            if (tcsetpgrp(STDIN_FILENO, getpgrp()) == -1)
                errExit("tcsetpgrp-parent");
        }
    }
```

simple_init 程序实现 init 进程的两个主要功能，一个是系统初始化，大多数 init 程序是复杂的且多半是基于表的方法来初始化系统。我们的 simple_init 程序提供一个简单的 shell 工具，该工具使我们能够手动执行需要初始化命名空间的任何命令；该方法使我们能够随意执行 shell 命令以对命名空间做一些测试。另一个 simple_init 实现的功能是使用 waitpid()获得其子进程的退出状态。

所以可以联合使用 ns_child_exec 和 simple_init 来在一个新的 PID 命名空间中启动 init 进程：

```
# ./ns_child_exec -p ./simple_init
init$
```

init$提示符表明 simple_init 程序能够读取和执行 shell 命令。

我们将使用上面的两个以及下面的 orphan.c 示例来证实在 PID 命名空间中变为孤儿进程将被 PID 命名空间中的 init 进程收留，而不是系统的 init 进程收留。

```
/* orphan.c

   Copyright 2013, Michael Kerrisk
   Licensed under GNU General Public License v2 or later

   Demonstrate that a child becomes orphaned (and is adopted by init(1),
   whose PID is 1) when its parent exits.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    pid_t pid;

    pid = fork();
```

```
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if (pid != 0) {                 /* Parent */
            printf("Parent (PID=%ld) created child with PID %ld\n",
                    (long) getpid(), (long) pid);
            printf("Parent (PID=%ld; PPID=%ld) terminating\n",
                    (long) getpid(), (long) getppid());
            exit(EXIT_SUCCESS);
        }

        /* Child falls through to here */

        do {
            usleep(100000);
        } while (getppid() != 1);           /* Am I an orphan yet? */

        printf("\nChild    (PID=%ld) now an orphan (parent PID=%ld)\n",
                (long) getpid(), (long) getppid());

        sleep(1);

        printf("Child    (PID=%ld) terminating\n", (long) getpid());
        _exit(EXIT_SUCCESS);
    }
```

orphan 程序执行一个 fork()命令创建子进程。在进程继续执行时父进程会退出；当父进程退出时子进程变成孤儿进程。子进程执行一个循环直到其变为一个孤儿进程（getppid()的返回值是 1）；一旦子进程变成孤儿进程，它将退出。父子进程打印的信息使我们能够看见当两个子进程退出时刻以及何时子进程变成孤儿进程。

为了更清楚 simple_init 程序接受孤儿进程的哪些信息，我们使用-v 选项，该选项将产生子进程自己产生的各种信息。

```
# ./ns_child_exec -p ./simple_init -v
    init: my PID is 1
init$ ./orphan
    init: created child 2
Parent (PID=2) created child with PID 3
Parent (PID=2; PPID=1) terminating
```

```
    init: SIGCHLD handler: PID 2terminated
init$               # simple_init promptinterleaved with output from child
Child (PID=3) now an orphan (parent PID=1)
Child (PID=3) terminating
    init: SIGCHLD handler: PID 3terminated
```

上面的输出中，以 init：开始的信息由 simple_init 在启动 verbose 选项时打印出的。其它所有的信息（除了 init$提示符）由 orphan 程序打印。从输出来看，子进程（PID 是 3）在父进程（PID 是 2）退出时变成孤儿进程。变成孤儿进程的子进程（PID 是 3）被命名空间的 init 进程（PID 是 1）收留。

信号和 init 进程

传统的 init 进程对信号的处理有些特殊。能够发送给 init 进程的信号是那些已经有信号处理函数的信号。其它所有信号将被忽略。这阻止了 init 进程被任何用户意外的 kill 掉，init 进程的存在对于系统的稳定是至关重要的。

PID 命名空间使命名空间内的 init 进程实现了和传统 init 类似的一些行为。命名空间内的其它进程（即使特权进程）仅能够发送 init 进程建立了处理函数的信号。这防止了命名空间内的其它进程不经意地杀死了命名空间中具有特殊作用的 init 进程。然而，如同传统的 init 进程一样，在通常的环境中内核仍然能够为 PID 命名空间内的 init 进程产生信号（例如，硬件中断，终端产生的 SIGTOU 等信号，定时器超时）。

PID 命名空间内的祖先进程仍然能够发送信号给子 PID 命名空间内的 Init 进程。同样的，只有 init 设置了对应处理函数的信号才可以被发送给它，除了 SIGKILL 和 SIGSTOP 这两个特例。当一个位于祖先 PID 命名空间的进程发送上述两个特殊信号给 init 进程时，它们被强行发送。SIGSTOP 停止 init 进程；SIGKILL 终结 init 进程。由于 init 进程对于 PID 命名空间如此重要，以至于如果 init 进程被 SIGKILL 信号终结掉，内核将对该 PID 命名空间内的其它所有进程发送 SIGKILL 信号来终结所有进程。

通常，PID 命名空间在 init 进程终结时将被摧毁，然而，特例是：只要该命名空间内对应的/proc/PID/ns/pid 文件被打开或者绑定挂载点，命名空间将不会被摧毁。然而，不太可能使用 setns()和 fork()在新的命名空间中创建进程：在 fork()调用时会检测到缺少 init 进程，这将会返回 ENOMEM 错误码（就是 PID 不能够被分配）。换句话说，PID 命名空间仍然存在，但是变得不可用。

挂载一个 procfs 文件系统

在先前这个系列的文章中，PID 命名空间的/proc 文件系统 procfs 被挂载在不

同的挂载点（而不是在/proc 挂载点），这使得我们可以使用 shell 命令行载对应的/proc/PID 目录下查看每一个新的 PID 命名空间，同时也可以使用 ps 命令查看在 rootPID 命名空间可以看见的进程。

然而像 ps 之类内容依赖于挂载在/proc 目录下的 procfs 文件以获取它们需要的信息。因此，如果我们想让 ps 在命名空间中也运行正确，我们需要为命名空间挂载一个 procfs。由于 simple_init 程序允许我们执行 shell 命令行，我们可以再命令行下使用如下 mount 命令：

```
# ./ns_child_exec -p -m ./simple_init
init$ mount -t proc proc /proc
init$ ps a
  PID TTY      STAT   TIME COMMAND
    1 pts/8    S      0:00 ./simple_init
    3 pts/8    R+     0:00 ps a
```

ps 命令列出了通过/proc 命令能够存取的所有进程。在这种情况下，我们仅看见两个进程，表明命名空间只有两个进程在运行。

当运行上面的 ns_child_exec 命令，我们使用了了-m 参数，该参数用于将创建的子进程放在一个独立的 mount 命名空间。这使得 mount 命令并不会影响命名空间之外的进程看到的/proc 挂载点。

unshare（）和 setns（）

自从 3.8 版本，上面两个系统调用能够被 PID 命名空间使用，但是有些特殊的地方。

指定 CLONE_NEWPID 标志使用 unshare()创建新的 PID 命名空间，但是并没有将 unshare 的调用者放入命名空间中。而调用者创建的任何子进程将被放在新的命名空间中；第一个子进程将是该命名空间的 init 进程。

setns()系统调用现在支持 PID 命名空间：

```
setns(fd, 0);   /* Second argument can be CLONE_NEWPID toforce a
                   check that 'fd' refersto a PID namespace */
```

fd 参数是一个指明 PID 命名空间的文件描述符，该描述符是 PID 命名空间调用者的后裔；通过打开目标命名空间中一个进程的/proc/PID/ns/pid 文件来获得该文件描述符。unshare()、setns()不将调用者放在创建的命名空间中，而调用者后续创建的子进程将被放入命名空间中。

我们可以使用一个增强型 ns_exec.c 版本来验证 setns()的一些特性。

```
/* ns_exec.c

   Copyright 2013, Michael Kerrisk
```

```
    Licensed under GNU General Public License v2 or later

    Join a namespace and execute a command in the namespace
*/
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

/* A simple error-handling function: print an error message based
    on the value in 'errno' and terminate the calling process */

#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                            } while (0)

int
main(int argc, char *argv[])
{
    int fd;

    if (argc < 3) {
        fprintf(stderr, "%s /proc/PID/ns/FILE cmd [arg...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);     /* Get descriptor for namespace */
    if (fd == -1)
        errExit("open");

    if (setns(fd, 0) == -1)               /* Join that namespace */
        errExit("setns");

    execvp(argv[2], &argv[2]);          /* Execute a command in namespace */
    errExit("execvp");
}
```

新的 ns_run.c 程序如下：

```
/* ns_run.c

    Copyright 2013, Michael Kerrisk
    Licensed under GNU General Public License v2 or later

    Join one or more namespaces using setns() and execute a command in
```

```
        those namespaces, possibly inside a child process.

        This program is similar in concept to nsenter(1), but has a
        different command-line interface.
*/
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

/* A simple error-handling function: print an error message based
        on the value in 'errno' and terminate the calling process */

#define errExit(msg)        do { perror(msg); exit(EXIT_FAILURE); \
                                 } while (0)

static void
usage(char *pname)
{
        fprintf(stderr, "Usage: %s [-f] [-n /proc/PID/ns/FILE] cmd [arg...]\n",
                        pname);
        fprintf(stderr, "\t-f          Execute command in child process\n");
        fprintf(stderr, "\t-n          Join specified namespace\n");

        exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
        int fd, opt, do_fork;
        pid_t pid;

        /* Parse command-line options. The initial '+' character in
            the final getopt() argument prevents GNU-style permutation
            of command-line options. That's useful, since sometimes
            the 'command' to be executed by this program itself
            has command-line options. We don't want getopt() to treat
            those as options to this program. */

        do_fork = 0;
```

```
while ((opt = getopt(argc, argv, "+fn:")) != -1) {
        switch (opt) {

        case 'n':               /* Join a namespace */
            fd = open(optarg, O_RDONLY); /* Get descriptor for namespace */
            if (fd == -1)
                errExit("open");

            if (setns(fd, 0) == -1)           /* Join that namespace */
                errExit("setns");
            break;

        case 'f':
            do_fork = 1;
            break;

        default:
            usage(argv[0]);
        }
}

if (argc <= optind)
        usage(argv[0]);

/* If the "-f" option was specified, execute the supplied command
     in a child process. This is mainly useful when working with PID
     namespaces, since setns() to a PID namespace only places
     (subsequently created) child processes in the names, and
     does not affect the PID namespace membership of the caller. */

if (do_fork) {
        pid = fork();
        if (pid == -1)
            errExit("fork");

        if (pid != 0) {                         /* Parent */
            if (waitpid(-1, NULL, 0) == -1)         /* Wait for child */
                errExit("waitpid");
            exit(EXIT_SUCCESS);
        }

        /* Child falls through to code below */
}
```

```
    execvp(argv[optind], &argv[optind]);
    errExit("execvp");
}
```

ns_run [-f] [-n/proc/PID/ns/FILE]... command [arguments]

该程序使用 setns()关联由/proc/PID/ns（-n 选项指定）文件指定的命名空间。然后其执行给定的 command 命令并将参数 arguments 传递给它，如果指定了-f 参数，它使用 fork()创建的子进程执行上面的命令。

考虑一个场景，在一个终端窗口，我们在一个新的 PID 命名空间中启动 simple_init 程序，启动 verbose 选项记录日志以方便我们知道何时子进程被收留：

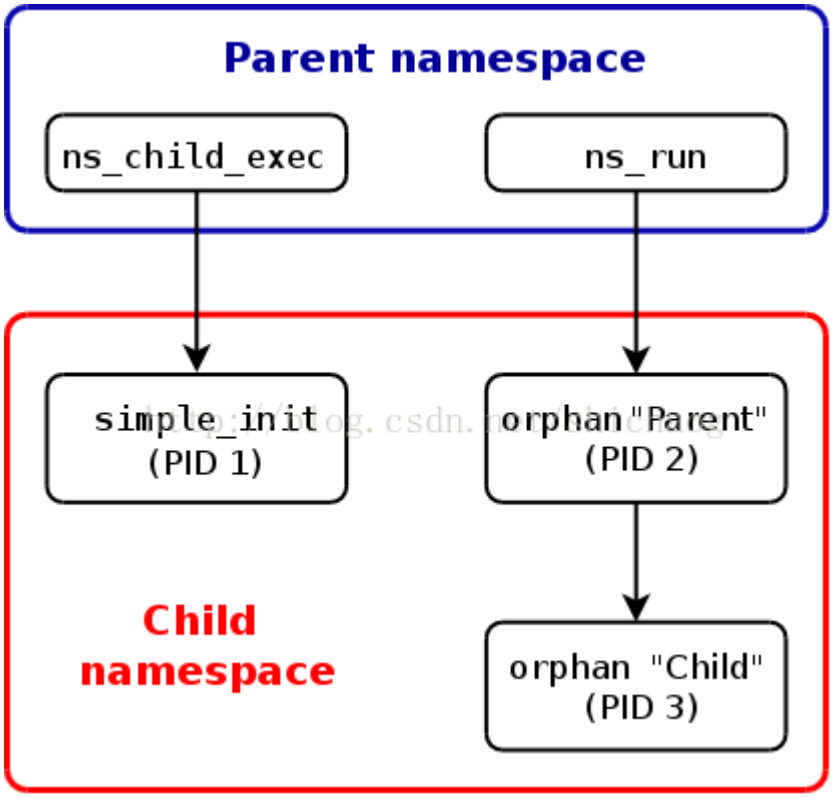
```
# ./ns_child_exec -p ./simple_init -v
    init: my PID is 1
init$
```

然后我们切到第二个终端，这里我们使用 ns_run 程序来执行我们的 orphan 程序。这个效果就是在 simple_init 管理的 PID 命名空间中创建两个子进程。

```
# ps -C sleep -C simple_init
 PID TTY          TIME CMD
9147 pts/8    00:00:00 simple_init
# ./ns_run -f -n /proc/9147/ns/pid./orphan
Parent (PID=2) created child with PID 3
Parent (PID=2; PPID=0) terminating
#
Child (PID=3) now an orphan (parent PID=1)
Child (PID=3) terminating
```

来看“父”进程（PID=2）的输出，可以知道其父进程 ID 是 0。这揭示了和启动 orphan 进程位于不同命名空间的事实。

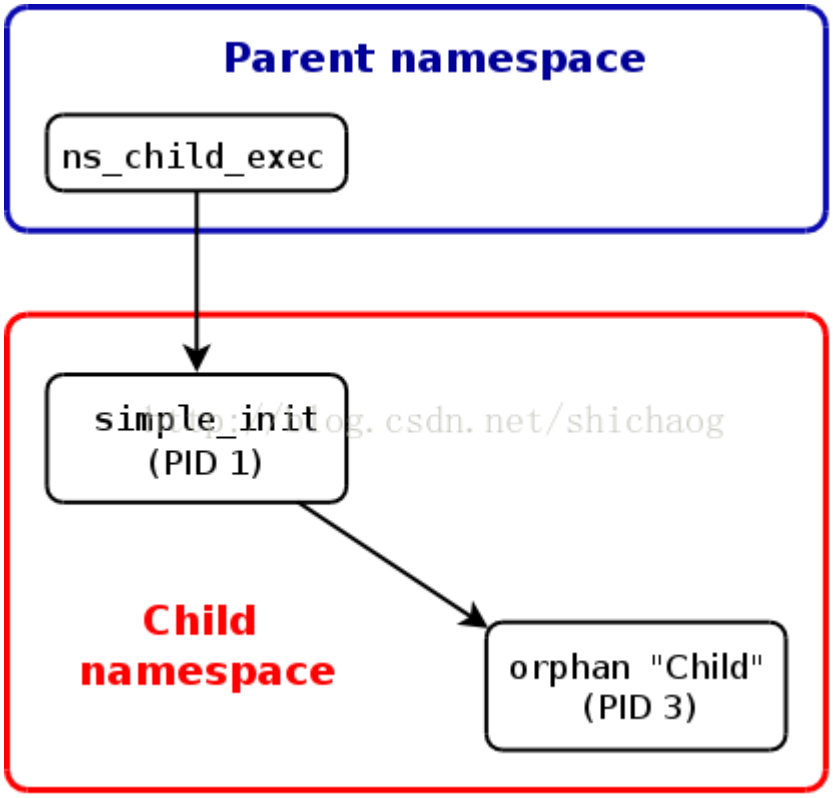
下图揭示了再 Oprah 的"父"进程终结之前的进程关系。箭头表明了进程间的父子关系。



回到创建 simple_init 程序的窗口，可以看到如下的输出：

init: SIGCHLD handler: PID 3 terminated

由 orphan 创建的子进程（PID 等于 3）由 simple_init 收留，但是其“父”进程（PID=2）并不是这样，这是因为“父”进程由在另一个命名空间中它自己的父进程 ns_run 收留。下图这展示了 orphan“父进程”结束但是“子”进程还未结束时的进程关系：



值得强调的是 setns()和 unshare()处理 PID 命名空间的方式不同。对其它类型的命名空间，这些系统调用确实改变了调用这些函数的进程的命名空间。不改变这些调用者的 PID 命名空间的原因是：成为其它 PID 命名空间的进程将导致进程自己的 PID 改变，这是由于 getpid()返回的是进程所在 PID 命名空间的进程 PID。许多用户空间的程序和库依赖一个进程的 PID 是确定的这一假设；如果进程的 PID 改变可能导致这些程序失败。换句话说，进程的 PID 命名空间在进程创建时确定，且后续不会改变。

总结

本文，我们查看了 PID 命名空间 init 进程，展示了为了让 ps 之类的工具可用而如何为一个 PID 命名空间挂载一个 procfs。此外，还查看了 unshare 和 setns 对 PID 命名空间特别的地方。

网络 namespaces

网络命名空间，正如从名字可以联想到的，网络命名空间分割了网络资源-设备、地址、端口、路由、防火墙规则等。网络命名空间出现在 2.6.24 内核版本，那几乎是五年前的事了；在一年前开发好了从那时起也被很多开发这忽略了。

网络命名空间的基本管理

正如其它命名空间，网络命名空间通过传递 CLONE_NEWNET 标志给 clone() 系统调用来创建。通过命令行工作 ip 很容易对网络命名空间进行相关的操作，例如：

```
# ip netns add netns1
```

上面的命令创建一个叫 netns1 的网络命名空间。当 ip 工具创建一个网络命名空间，其将在/var/run/netns 目录下创建一个绑定的 mount 点；这样即使命名空间中没有进程运行命名空间仍然存在，并且也简化的命名空间自身的操作。由于网络命名空间在可用前需要进行相当数量的配置工作，系统管理这真该感谢该特性。

“ip netns exec”命令能被用于在一个命名空间中执行网络管理命令：

```
# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

这个命令列出了命名空间可以看见的接口。一个网络命名空间的移除使用如下命令：

```
# ip netns delete netns1
```

该命令删除命名空间对应的挂载点。然而，只要命名空间内有进程在运行命名空间自身将持续存在。

网络命名空间配置

新的网络命名空间将仅有一个回环设备。除了回环设备，每一个网络设备（物理和虚拟接口，桥等）仅仅只能存在于一个网络命名空间。此外，除了 root 不能够将物理设备（没有连接到真实硬件）分配到命名空间。相反，虚拟网络设备（虚拟以太网或者 veth）能够被创建并分配到命名空间中。这些虚拟设备使得命名空间内的进程能够通过网络进行通信；如何通信则依赖于配置、路由等。

当第一次被创建时，在命名空间中的 lo 回环设备是 down 的状态，所以即使一个 ping 命令也会失败：

```
# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
```

启动该接口将允许 ping 回环设备的地址：

```
# ip netns exec netns1 ip link set dev lo up
# ip netns exec netns1 ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
```

```
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.051 ms
...
```

但这仍然不能是 netns1 命名空间和 root 命名空间通信。为了实现它们间通信的目的，虚拟网络设备需要被创建和配置：

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns netns1
```

第一个命令设置了连接的一对虚拟网络设备。发送给 veth0 数据包将会被 veth1 收到，反之亦然。第二个命令将 veth1 的网络命名空间设为 netns1。

```
# ip netns exec netns1 ifconfig veth1 10.1.1.1/24 up
# ifconfig veth0 10.1.1.2/24 up
```

然后，这两个命令设置两个设备的 IP 地址。

```
# ping 10.1.1.1
PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data.
64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.087 ms
...

# ip netns exec netns1 ping 10.1.1.2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.054 ms
...
```

两个方向的通信现在变得可行了，上述的 ping 命令已经验证了。

正如先前提到的，命名空间并不共享路由表或者防火墙规则，如在 netns1 中运行 route 和 iptables 命令。

```
# ip netns exec netns1 route
# ip netns exec netns1 iptables -L
```

第一个命令显示了数据包路由到 10.1.1 子网的路由，第二个命令则表明没有防火墙配置。所有的那些意味着从 netns1 送到网络上的数据包过大将会产生“网络不可达”错误。如果需要还有几个方法将命名空间连接上网。可以再在 root 命名空间和 netns1 命名空间为 veth 设备创建网桥。另外，IP 转发和网络地址转换（NAT）可以再 root 命名空间中被配置。这两种方法将允许从网络接收数据包，也运行 netns1 发送数据包到网络。

被分配到命名空间的非 root 进程（通过 clone、unshare、setns）仅能访问命名空间内已经设置的网络设备。当然，root 能够添加并配置新的网络设备。使用 ip netns 前缀的命令，有两种方法来寻址一个命名空间：通过名字，如 netns1，

或者通过进程 ID。因为 init 通常存在于 root 命名空间，可以使用如下的命令：

```
# ip link set vethX netns 1
```

这将在 root 命名空间中创建一个新的 veth 设备，并且该设备可以为任何其他命名空间 root 所用。在并不希望 root 拥有操作某个命名空间网络设备的权限时，PID 和 mount 命名空间可以用来完成此目的。

网络命名空间使用

正如我们看到的，一个命名空间的网络可以从不能进行网络通信到使用所有的的网络带宽进行通信。这导致了网络命名空间的许多不同的使用案例。

通过关闭一个命名空间的网络，管理员可以确保命名空间中运行的进程不能和命名空间之外的程序进行通信。即使一个进程牺牲一些安全性，其也不能够执行一些像关联僵尸网络或者发送垃圾邮件之类的动作。

即使处理网络流量的进程（web 服务器工作者进程或者 web 浏览渲染进程等）可以被放置于一个特定的命名空间。一旦和远端的连接建立，连接的文件描述符可以被由 clone()创建的新的网络命名空间内的子进程处理。子进程会继承父进程的文件描述符，这就可以存取连接的描述符了。另外一个可能的情况是父进程通过 UNIX 套接字将连接的文件描述符发送到一个给定的命名空间。不论哪种方式，命名空间中缺少合适的网络设备将使子进程或者工作进程的其它网络连接失败。

命名空间同样可以被用来在所有的网络命名空间中测试复杂的网络配置。在锁定的，限定防火墙的命名空间中运行敏感的服务同样也是一个使用案例。很明显，container 依赖网络命名空间为每一个 container 提供各自的网络视图。

命名空间提供了一个分割系统资源以及将进程组的进程与它们的资源隔离的方法。网络命名空间也是这样，但是由于网络是安全敏感的一个领域，提供各种类型的隔离具有很到的价值。当然，使用多种类型的命名空间可以实现超越安全和其它应用需要的隔离。

原文网址

http://lwn.net/Articles/531114/