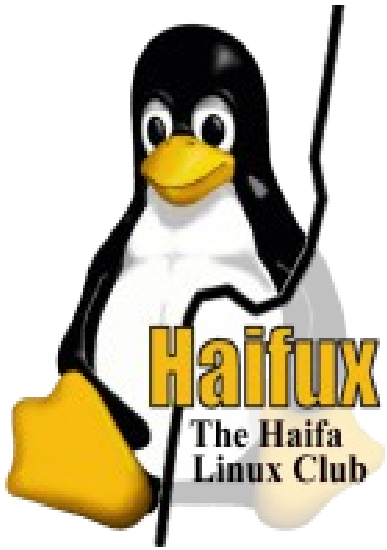
Resource management: Linux kernel Namespaces and cgroups



Rami Rosen
ramirose@gmail.com
Haifux, May 2013
www.haifux.org

TOC

Note: All code examples are from for_3_10 branch of cgroup git tree (3.9.0-rc1, April 2013)

General

The presentation deals with two Linux process resource management solutions: namespaces and cgroups.

We will look at:
- Kernel Implementation details.
- what was added/changed in brief.
- User space interface.
- Some working examples.
- Usage of namespaces and cgroups in other projects.
- Is process virtualization indeed lightweight comparing to Os virtualization ?
- Comparing to VMWare/qemu/scaleMP or even to Xen/KVM.

Namespaces

- Namespaces - lightweight process virtualization.
  - **Isolation:** Enable a process (or several processes) to have different views of the system than other processes.
  - 1992: “The Use of Name Spaces in Plan 9”
  - http://www.cs.bell-labs.com/sys/doc/names.html
    - Rob Pike et al, ACM SIGOPS European Workshop 1992.
  - Much like Zones in Solaris.
  - No hypervisor layer (as in OS virtualization like KVM, Xen)
  - Only one system call was added (***setns()***)
  - Used in Checkpoint/Restart
- Developers: Eric W. Biederman, Pavel Emelyanov, Al Viro, Cyrill Gorcunov, more.
  -

Namespaces - contd

There are currently 6 namespaces:

- mnt (mount points, filesystems)
- pid (processes)
- net (network stack)
- ipc (System V IPC)
- uts (hostname)
- user (UIDs)

Namespaces - contd

It was intended that there will be 10 namespaces: the following 4 namespaces are not implemented (yet):

- security namespace
- security keys namespace
- device namespace
- time namespace.
  - There was a time namespace patch – but it was not applied.
  - See: PATCH 0/4 - Time virtualization:
  - http://lwn.net/Articles/179825/
- see **ols2006**, "Multiple Instances of the Global Linux Namespaces" Eric W. Biederman

Namespaces - contd

- Mount namespaces were the first type of namespace to be implemented on Linux by Al Viro, appearing in 2002.
  - Linux 2.4.19.
- CLONE_NEWNS flag was added (stands for “**new namespace”;** at that time, no other namespace was planned, so it was not called new mount...)
- User namespace was the last to be implemented. A number of Linux filesystems are not yet user-namespace aware

Implementation details

•Implementation (partial):
 - 6 CLONE_NEW * flags were added:
(include/linux/sched.h)

• These flags (or a combination of them) can be used in ***clone()*** or ***unshare()*** syscalls to create a namespace.
•In ***setns()***, the flags are optional.

CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Implementation - contd

- Three system calls are used for namespaces:
- ***clone()*** **-** creates a **new process** and a **new namespace**; the process is attached to the new namespace.
  - Process creation and process termination methods, ***fork()*** and ***exit()*** methods, were patched to handle the new namespace CLONE_NEW* flags.
- ***unshare()*** **-** does not create a new process; creates a new namespace and attaches the current process to it.
  - ***unshare()*** was added in 2005, but not for namespaces only, but also for security.

    see “new system call, unshare” : http://lwn.net/Articles/135266/

- ***setns()*** **-** a new system call was added, for joining an existing namespace.

Nameless namespaces

From man (2) clone:

...

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

...

•Flags is the CLONE_* flags, including the namespaces CLONE_NEW* flags. There are more than 20 flags in total.

- See include/uapi/linux/sched.h

•There is **no** parameter of a namespace **name**.

• How do we know if two processes are in the same namespace ?

• **Namespaces do not have names.**

• Six entries (inodes) were added under **/proc/<pid>/ns** (one for each namespace) (in kernel 3.8 and higher.)

• Each namespace has a **unique** inode number.

•This inode number of a each namespace is created when the namespace is created.

Nameless namespaces

- **ls -al /proc/<pid>/ns**

```
lrwxrwxrwx 1 root root 0 Apr 24 17:29 ipc ->   ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Apr 24 17:29 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Apr 24 17:29 net ->  net:[4026531956]
lrwxrwxrwx 1 root root 0 Apr 24 17:29 pid ->   pid:[4026531836]
lrwxrwxrwx 1 root root 0 Apr 24 17:29 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Apr 24 17:29 uts ->   uts:[4026531838]
```

You can use also *readlink.*

Implementation - contd

- A member named nsproxy was added to the process descriptor , struct task_struct.
- A method named *task_nsproxy(struct task_struct *tsk)*, to access the nsproxy of a specified process. (include/linux/nsproxy.h)

- nsproxy includes 5 inner namespaces:
- **uts_ns, ipc_ns, mnt_ns, pid_ns, net_ns;**

  Notice that user ns is *missing* in this list,

- it is a member of the credentials object (struct cred) which is a member of the process descriptor, task_struct.

- There is an initial, default namespace for each namespace.

Implementation - contd

- Kernel config items:

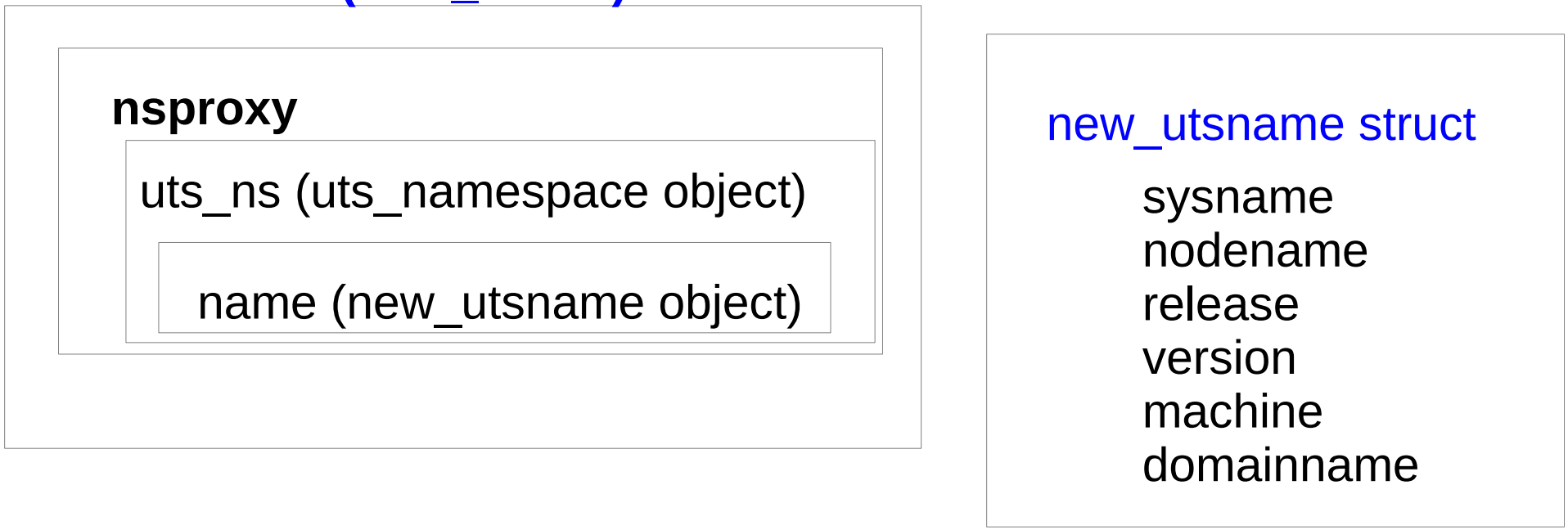
CONFIG_NAMESPACES
- CONFIG_UTS_NS
- CONFIG_IPC_NS
- CONFIG_USER_NS
- CONFIG_PID_NS
- CONFIG_NET_NS

- user space additions:
- IPROUTE package
- some additions like *ip netns add/ip netns del* and more.
- util-linux package
- *unshare* util with support for all the 6 namespaces.
- *nsenter* – a wrapper around *setns()*.

UTS namespace

- uts - (Unix timesharing)
  - Very simple to implement.

Added a member named uts_ns (uts_namespace object) to the nsproxy.



UTS namespace - contd

The **old** implementation of ***gethostname()***:

```
asmlinkage long sys_gethostname(char __user *name, int len)
{
...
        if (copy_to_user(name, system_utsname.nodename, i))
...                 errno = -EFAULT;
}
```

(system_utsname is a global)

kernel/sys.c, Kernel v2.6.11.5

UTS namespace - contd

A Method called ***utsname()*** was added:

```
static inline struct new_utsname *utsname(void)
{
    return &current->nsproxy->uts_ns->name;
}
```

The **new** implementation of ***gethostname()***:

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

Similar approach in ***uname()*** and ***sethostname()*** syscalls.

UTS namespace - Example

We have a machine where hostname is myoldhostname.
uname -n
myoldhostname

unshare -u /bin/bash

This create a UTS namespace by unshare() syscall and call ***execvp()*** for invoking bash.
Then:

hostname mynewhostname
uname -n
    mynewhostname

Now from a different terminal we will run *uname -n*, and we will see ***myoldhostname.***

UTS namespace - Example

nsexec

nsexec is a package by Serge Hallyn; it consists of a program called nsexec.c which creates tasks in new namespaces (there are some more utils in it) by clone() or by unshare() with fork().

https://launchpad.net/~serge-hallyn/+archive/nsexec

Again we have a machine where hostname is myoldhostname.

uname -n

myoldhostname

IPC namespaces

The same principle as uts , nothing special, more code.
Added a member named **ipc_ns** (ipc_namespace object) to the nsproxy.

- CONFIG_POSIX_MQUEUE or CONFIG_SYSVIPC must be set

Network Namespaces

- A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices.
- The network namespace is **struct net.** (defined in include/net/net_namespace.h)

Struct net includes all network stack ingredients, like:

- Loopback device.
- SNMP stats. (netns_mib)
- All network tables:routing, neighboring, etc.
- All sockets
- /procfs and /sysfs entries.

Implementations guidelines

• **A network device belongs to exactly one network namespace.**

• Added to struct net_device structure:

• struct net *nd_net;
for the Network namespace this network device is inside.

•Added a method: *dev_net(const struct net_device *dev)*
to access the nd_net namespace of a network device.

• **A socket belongs to exactly one network namespace.**

• Added **sk_net** to struct sock (also a pointer to struct net), for the Network namespace this socket is inside.

• Added *sock_net()* and *sock_net_set()* methods (get/set network namespace of a socket)

Network Namespaces - contd

- Added a system wide linked list of all namespaces: ***net_namespace_list,*** and a macro to traverse it ***(for_each_net())***
- The initial network namespace, **init_net (instance of struct net)**, includes the loopback device and all physical devices, the networking tables, etc.
- Each newly created network namespace includes only the loopback device.
- There are no sockets in a newly created namespace:

netstat -nl

Active Internet connections (only servers)

Proto Recv-Q Send-Q Local Address Foreign Address State

Active UNIX domain sockets (only servers)

Proto RefCnt Flags Type State I-Node Path

Example

- Create two namespaces, called "myns1" and "myns2":
- ***ip netns add myns1***
- ***ip netns add myns2***
  - ***(In fedora 18, ip netns is included in the iproute package).***
- This triggers:
- creation of */var/run/netns/myns1,/var/run/netns/myns2* empty **folders**
- *calling the **unshare()** system call with CLONE_NEWNET.*
  - ***unshare()** does not trigger cloning of a process; it does create a new namespace (a network namespace, because of the CLONE_NEWNET flag).*
- *see **netns_add()** in ipnetns.c (iproute2)*

- You can use the file descriptor of */var/run/netns/myns1* with the ***setns()*** system call.
- **From man 2 setns:**

  ...

  int setns(int fd, int nstype);

  DESCRIPTION

  Given a file descriptor referring to a namespace, reassociate the calling thread with that namespace.

  ...
- In case you pass 0 as nstype, no check is done about the fd.
- In case you pass some nstype, like CLONE_NEWNET of CLONE_NEWUTS, the method verifies that the specified nstype corresponds to the specified fd.

Network Namespaces - delete

- You delete a namespace by:
- ***ip netns del myns1***
  - This unmounts and removes /var/run/netns/myns1
  - see ***netns_delete()*** in ipnetns.c
  - Will not delete a network namespace if there is one or more processes attached to it.

- Notice that after deleting a namespace, all its **migratable** network devices are moved to the default network namespace;
- **unmoveable** devices (devices who have **NETIF_F_NETNS_LOCAL** in their features) and **virtual** devices are not moved to the default network namespace.
- (The semantics of migratable network devices and unmoveable devices are taken from ***default_device_exit()*** method, net/core/dev.c).

NETIF_F_NETNS_LOCAL

- NETIF_F_NETNS_LOCAL ia a network device feature
  - (a member of net_device struct, of type netdev_features_t)
- It is set for devices that are not allowed to move between network namespaces; sometime these devices are named "local devices".
- Example for local devices (where NETIF_F_NETNS_LOCAL is set):
  - Loopback, VXLAN, ppp, bridge.
  - You can see it with ethtool (by ***ethtool -k***, or ***ethtool –show-features***)
  - *ethtool -k p2p1*

    netns-local: off [fixed]

For the loopback device:

ethtool -k lo

netns-local: on [fixed]

VXLAN

- Virtual eXtensible Local Area Network.
- VXLAN is a standard protocol to transfer layer 2 Ethernet packets over UDP.
- Why do we need it ?
- There are firewalls which block tunnels and allow, for example, only TCP/UDP traffic.
- developed by Stephen Hemminger.
  - drivers/net/vxlan.c
  - IANA assigned port is 4789
  - Linux default is 8472 (legacy)

When trying to move a device with NETIF_F_NETNS_LOCAL flag, like VXLAN, from one namespace to another, we will encounter an error:

ip link add myvxlan type vxlan id 1
ip link set myvxlan netns myns1

We will get: RTNETLINK answers: Invalid argument

```
int dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat)
{
        int err;

        err = -EINVAL;
        if (dev->features & NETIF_F_NETNS_LOCAL)
                goto out;
...
}
```

- You list the network namespaces (which were added via “ ip netns add”)
- *ip netns list*
  - this simply reads the namespaces under:

    /var/run/netns
- You can find the pid (or list of pids) in a specified net namespace by:
  - ip netns pids namespaceName
- You can find the net namespace of a specified pid by:
  - ip/ip netns identify #pid

You can monitor addition/removal of network namespaces by:
ip netns monitor

- prints one line for each addition/removal event it sees

- Assigning p2p1 interface to myns1 network namespace:
- ***ip link set p2p1 netns myns1***
  - *This triggers changing the network namespace of the net_device to "myns1".*
  - *It is handled by **dev_change_net_namespace()**, net/core/dev.c.*
- Now, running:
- ***ip netns exec myns1 bash***
- will transfer me to myns1 network namespaces; so if I will run there:
- *ifconfig -a*
- I will see p2p1 (and the loopback device);
  - Also under /sys/class/net, there will be only p2p1 and lo folders.
- But if I will open a new terminal and type *ifconifg -a*, I will not see p2p1.

- Also, when going to the second namespace by running:
- ***ip netns exec myns2 bash***
- will transfer me to myns2 network namespace; but if we will run there:
- *ifconfig -a*
  - We will **not** see p2p1; we will only see the loopback device.
- We move a network device to the default, initial namespace by:

  ip link set p2p1 netns 1

- In that namespace, network application which look for files under /etc, will first look in /etc/netns/myns1/, and then in /etc.
- For example, if we will add the following entry "192.168.2.111 www.dummy.com"
- in /etc/netns/myns1/hosts, and run:
- ping www.dummy.com
- we will see that we are pinging 192.168.2.111.

veth

- You can communicate between two network namespaces by:
- creating a pair of network devices (veth) and move one to another network namespace.
- Veth (Virtual Ethernet) is like a pipe.
- unix sockets (use paths on the filesystems).

Example with veth:

Create two namesapces, myns1 and myns1:

ip netns add myns1

ip netns add myns2

veth

ip netns exec myns1 bash

- open a shell of myns1 net namespace

ip link add name if_one type veth peer name if_one_peer

- create veth interface, with if_one and if_one_peer
- ifconfig running in myns1 will show if_one and if_one_peer and lo (the loopback device)
- ifconfig running in myns2 will show only lo (the loopback device)

Run from myns1 shell:

ip link set dev if_one_peer netns myns2

move if_one_peer to myns2

- now ifconfig running in myns2 will show if_one_peer and lo (the loopback device)

- Now set ip addresses to if_one (myns1) and if_one_peer (myns2) and you can send traffic.

unshare util

- The **unshare** utility
- **Util-linux** recent git tree has the unshare utility with support for all six namespaces:

http://git.kernel.org/cgit/utils/util-linux/util-linux.git

./unshare –help

...

Options:

-m, --mount	unshare mounts namespace
-u, --uts	unshare UTS namespace (hostname etc)
-i, --ipc	unshare System V IPC namespace
-n, --net	unshare network namespace
-p, --pid	unshare pid namespace
-U, --user	unshare user namespace

- For example:
- Type:
- **./unshare --net bash**
  - A new network namespace was generated and the bash process was generated inside that namespace.
- Now run ***ifconfig -a***
- You will see only the loopback device.
  - With unshare util, no folder is created under /var/run/netns; also network application in the net namespace we created, do not look under /etc/netns
  - If you will kill this bash or exit from this bash, then the network namespace will be freed.
  -
  -

This is not the case as with ip netns exec myns1 bash; in that case, killing/exiting the bash does not trigger destroying the namespace.

For implementation details, look in
*put_net(struct net *net) and the reference count (named “count”) of the network namespace struct net.*

Mount namespaces

- Added a member named **mnt_ns** (mnt_namespace object) to the nsproxy.

- We copy the mount namespace of the calling process using generic filesystem method (see *copy_tree()* in *dup_mnt_ns()*).

- In the new mount namespace, all previous mounts will be visible; and from now on:
- mounts/unmounts in that mount namespace are invisible to the rest of the system.
- mounts/unmounts in the global namespace are visible in that namespace.
- pam_namespace module uses mount namespaces (with unshare(CLONE_NEWNS)) (modules/pam_namespace/pam_namespace.c)

mount namespaces: example 1

Example 1 (tested on Ubuntu):
Verify that /dev/sda3 is not mounted:

mount | grep /dev/sda3

should give nothing.

unshare -m /bin/bash

mount /dev/sda3 /mnt/sda3

now run mount | grep sda3
We will see:
/dev/sda3 on /mnt/sda3 type ext3 (rw)

readlink /proc/$$/ns/mnt
mnt:[4026532114]

From another terminal run

readlink /proc/$$/ns/mnt

mnt:[4026531840]

The results shows that we are in a different namespace.

Now run:

mount | grep sda3

/dev/sda3 on /mnt/sda3 type ext3 (rw)

Why ? We are in a different mount namespace?

We should have not see the mount which was done from another namespace!

The answer is simple: running mount is not good enough when working with mount namespaces. The reason is that mount reads **/etc/mtab**, which was updated by the mount command; mount command does not access the kernel structures.

What is the solution?

To access directly the kernel data structures, you should run:

cat /proc/mounts | grep sda3

(/proc/mounts is in fact symbolic link to /proc/self/mounts).

Now you will get no results, as expected.

mount namespaces: example 2

Example2: tested on Fedora 18

Verify that /dev/sdb3 is not mounted:

mount | grep sdb3

should give nothing.

unshare -m /bin/bash

mount /dev/sdb3 /mnt/sdb3

now run mount | grep sdb3

You will see:

/dev/sdb3 on /mnt/sdb3 type ext4 (rw,relatime,data=ordered)

readlink /proc/$$/ns/mnt

mnt:[4026532381]

From another terminal run:

readlink /proc/$$/ns/mnt

mnt:[4026531840]

This shows that we are in a different namespace.

Now run:

mount | grep sdb3

/dev/sdb3 on /mnt/sdb3 type ext4 (rw,relatime,data=ordered)

- We know now that we should use cat /proc/mounts (and not *mount*) to get the right answer when working with namespace; so:

cat /proc/mounts | grep sdb3

/dev/sdb3 /mnt/sdb3 ext4 rw,relatime,data=ordered 0 0

Why is it so ? We should have seen no results, as in previous example.

Answer: Fedora runs **systemd;**systemd uses the shared flag for mounting /.

From **systemd** source code: (*src/core/mount-setup.c*)

```
nt mount_setup(bool loaded_policy) {
  ...
  if (mount(NULL, "/", NULL, MS_REC|MS_SHARED, NULL) < 0)
        log_warning("Failed to set up the root directory for shared mount propagation: %m");
  ...
```

(MS_REC stands for recursive mount)

How do I know whether we have a shared flags ?

cat /proc/self/mountinfo | grep shared

We will see:

..

33 1 8:3 / / rw,relatime shared:1 - ext4 /dev/sda3 rw,data=ordered

..

What to do ?

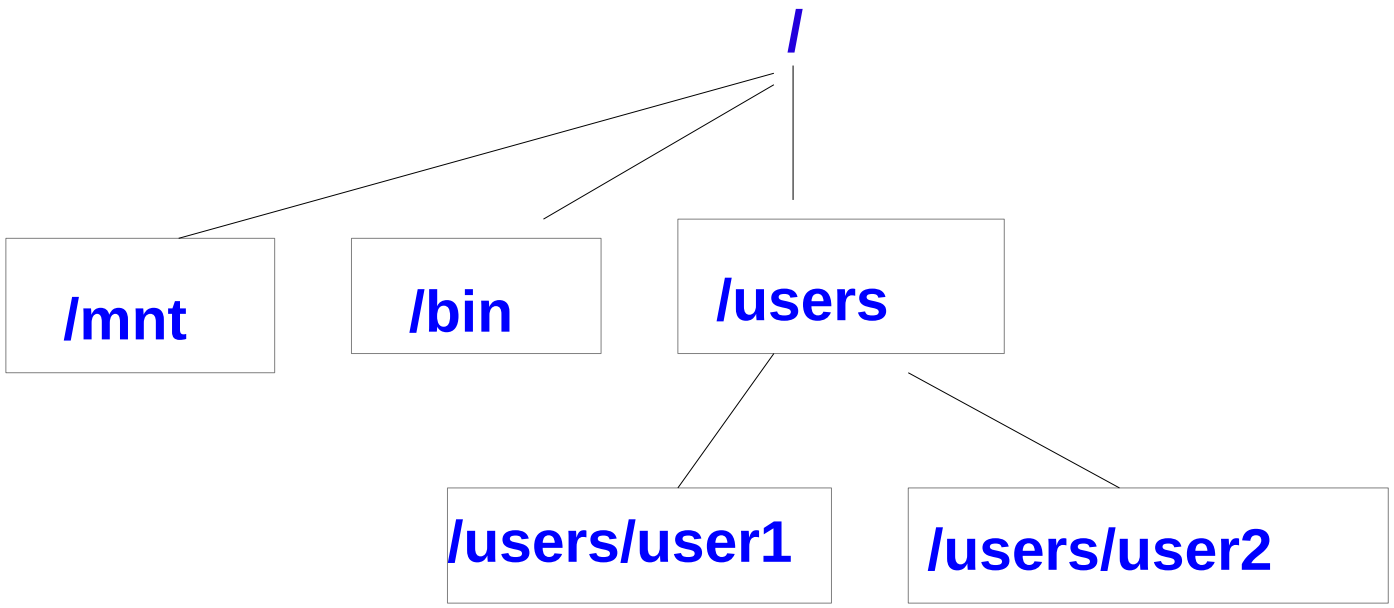
mount --make-rprivate -o remount / /dev/sda3

This changes the shared flag to private, recursively.

--make-rprivate – set the private flag recursively

Shared subtrees

By default, the filesysytem is mounted as **private**, unless the shared mount flag is set explicitly.

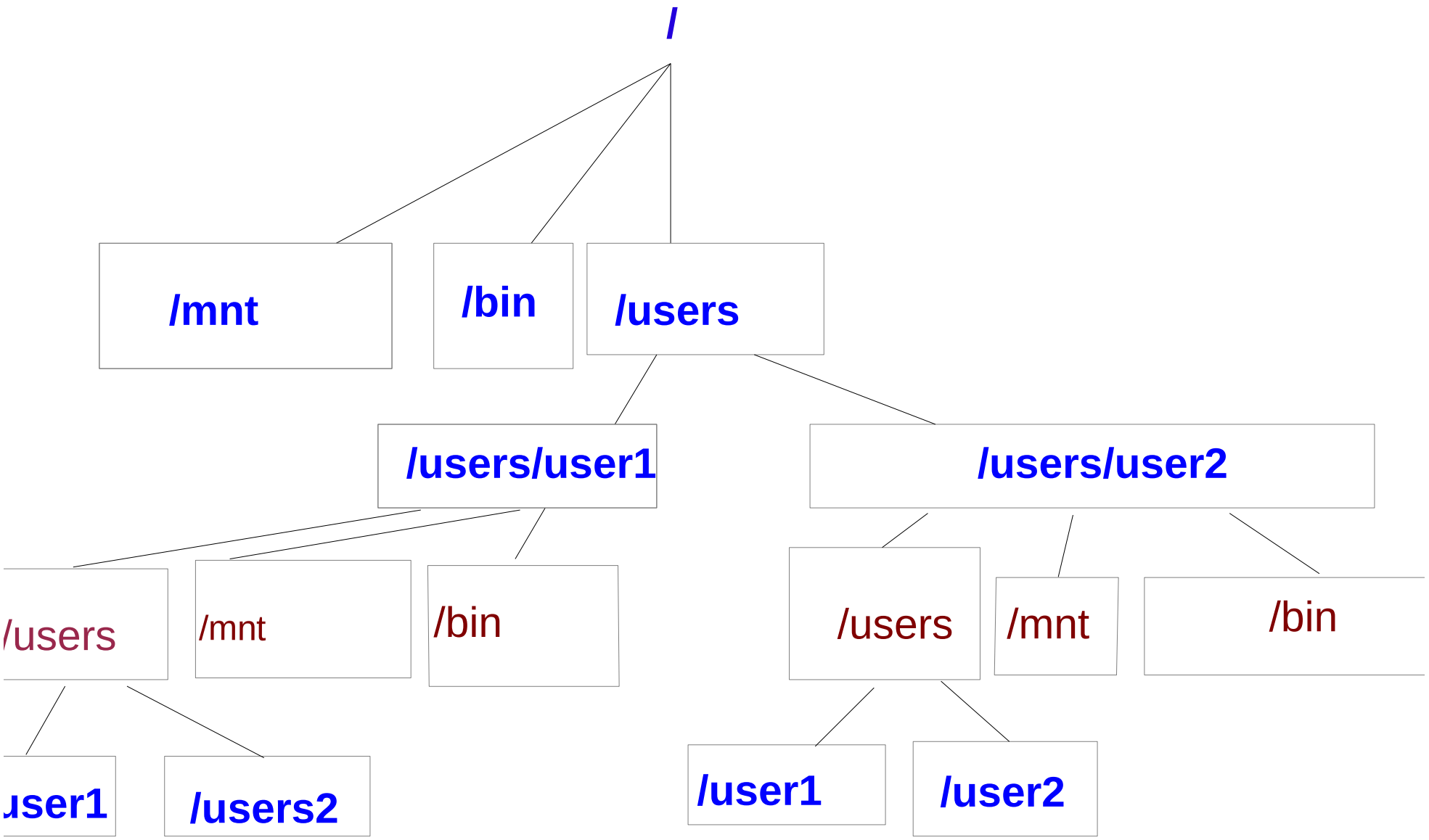


Now, we want that user1 and user2 folders will see the whole filesystem; we will run

mount –bind / /users/user1

mount –bind / /users/user2

Shared subtrees - contd



Shared subtrees – Quiz

Quiz:
Now, we mount a usb disk on key on /mnt/dok.

Will it be seen in /users/user1/mnt or /users/user2/mnt?

Shared subtrees - contd

The answer is no, since by default, the filesysytem is mounted as **private**. To enable that the dok will be seen also under /users/user1/mnt or /users/user2/mnt, we should mount the filesystem as shared:

mount / --make-rshared

And then mount the usb disk on key again.

The shared subtrees patch is from 2005 by Ram Pai.
It add some mount flags like –make-slave, --make-rslave, -make-unbindable, --make-runbindable and more. The patch added this kernel mount flags: MS_UNBINDABLE, MS_PRIVATE, MS_SLAVE and MS_SHARED

The shared flag is in use by the fuse filesystem.

PID namespaces

- Added a member named pid_ns (pid_namespace object) to the nsproxy.
- Processes in different PID namespaces can have the same process ID.
- When creating the first process in a new namespace, its PID is 1.
- Behavior like the “init” process:
  - When a process dies, all its orphaned children will now have the process with PID 1 as their parent (**child reaping**).
  - Sending **SIGKILL** signal does not kill process 1, regardless of which namespace the command was issued (initial namespace or other pid namespace).

PID namespaces - contd

- When a new namespace is created, we cannot see from it the PID of the parent namespace; running ***getppid()*** from the new pid namespace will return 0.
- But all PIDs which are used in this namespace are visible to the parent namespace.
- pid namespaces can be nested, up to 32 nesting levels. (MAX_PID_NS_LEVEL).
- See: multi_pidns.c, Michael Kerrisk, from http://lwn.net/Articles/532745/.
- When trying to run multi_pidns with 33, you will get:
  - clone: Invalid argument

User Namespaces

- Added a member named user_ns (user_namespace object) to the nsproxy.
- include/linux/user_namespace.h
- Includes a pointer named **parent** to the user_namespace that created it.
- *struct user_namespace *parent;*
- Includes the effective uid of the process that created it:
- *kuid_t owner;*

- A process will have distinct set of UIDs, GIDs and capabilities.

User Namespaces

Creating a new user namespace is done by passing **CLONE_NEWUSER** to ***fork()*** or ***unshare().***

Example:
Running from some user account

```
id -u
    1000  // 1000 is the effective user ID.
id -g
    1000 //  1000 is the effective group ID.
```

(usually the first user added gets uid/gid of 1000)

User Namespaces - example

Capbilties:
cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000001fffffffff

In order to create a user namespace and start a shell, we will run from that non-root account:

./nsexec -cU /bin/bash

- The c flag is for using clone
- The U flag is for using user namespace (CLONE_NEWUSER flag for clone())

User Namespaces - example -contd

Now from the new shell run

```
id -u
    65534
id -g
    65534
```

- These are default values for the eUID and eGUID In the new namespace.
- We will get the same results for effective user id and effective root id also when running ***/nsexec -cU /bin/bash as root.***
- ***The defaults can be changed by: /proc/sys/kernel/overflowuid, /proc/sys/kernel/overflowgid***
- ***In fact, the user namespace that was created had full capabilities, but the call to exec() with bash removed them.***

cat /proc/self/status | *grep Cap*

CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000001fffffffff

User Namespaces - contd

Now run:

echo $$ (get the bash pid)

Now, from a different **root** terminal, we set the uid_map:

First, we can see that uid_map is uninitialized by:

cat /proc/<pid>/uid_map

Then:

echo 0 1000 10 > /proc/<pid>/uid_map

(<pid> is the pid of the bash process from previous step).

Entry in uid_map is of the following format:

namespace_first_uid host_first_uid number_of_uids

So this sets the first uid in the new namespace (which correspond to uid 1000 in the outside world) to be 0; the second will be 1; and so forth, for 10 entries.

User Namespaces - contd

Note: you can set the uid_map only once for a specific process. Further attempts will fail.

run
id -u
    You will get 0.

whoami
root

•User namespace is the only namespace which can be created without CAP_SYS_ADMIN capability

cat /proc/self/status | grep Cap

```
CapInh:    0000000000000000
CapPrm:    0000001fffffffff
CapEff:    0000001fffffffff
CapBnd:    0000001fffffffff
```

The CapEff (Effective Capabilites) is 1fffffffff-> this is 37 bits of '1' , which means all capabilities.

Quiz: Will unshare --net bash work now ?

Answer: no
unshare --net bash
unshare: cannot set group id: Invalid argument

But after running, from a different terminal, as root:
echo 0 1000 10 > /proc/2429/gid_map
It will work.

ls /root will fail however:

ls /root/
ls: cannot open directory /root/: Permission denied

Short quiz 1:
I am a regular user, not root.
Will clone() with (CLONE_NEWNET) work ?

Short quiz 2:
Will clone() with (CLONE_NEWNET | CLONE_NEWUSER) work ?

•Quiz 1 : No.

• In order to use the CLONE_NEWNET we need to have CAP_SYS_ADMIN.

unshare --net bash

unshare: unshare failed: Operation not permitted

•Quiz 2: Yes.

namespaces code guarantees us that user namespace creation is the first to be created. For creating a user namespace we do'nt need CAP_SYS_ADMIN. The user namespace is created with full capabilities, so we can create the network namespace successfully.

./unshare --net --user /bin/bash

No errors!

Quiz 3:
If you run, from a non root user,
unsare –user bash

And then
cat /proc/self/status | *grep CapEff*
CapEff: 0000000000000000

This means no capabilities. So how was the net namespace, which needs CAP_SYS_ADMIN, created ?

Answer: we first do unshare;
It is first done with user namespace. This enables all capabilities. Then we create the namespace. Afterwards, we call exec for the shell; exec removes capabilities.

From unshare.c of util-linux:

```
if (-1 == unshare(unshare_flags))
        err(EXIT_FAILURE, _("unshare failed"));
...

    exec_shell();
```

Anatomy of a user namespaces vulnerability
By Michael Kerrisk, March 2013
About CVE 2013-1858 - exploitable security vulnerability
http://lwn.net/Articles/543273/

cgroups

- **cgroups** (control groups) subsystem is a Resource Management solution providing a generic process-grouping framework.
- This work was started by engineers at Google (primarily Paul Menage and Rohit Seth) in **2006** under the name "process containers; in **2007**, renamed to “Control Groups”.
  - Maintainers: Li Zefan (huawei) and Tejun Heo ;
  - The memory controller (memcg) is maintained separately (4 maintainers)
  - Probably the most complex.
  - Namespaces provide per process resource isolation solution.
  - Cgroups provide resource management solution (handling groups).
- Available in Fedora 18 kernel and ubuntu 12.10 kernel (also some previous releases).
  - Fedora systemd uses cgroups.
  - Ubuntu does not have systemd. Tip: do tests with Ubuntu and also make sure that cgroups are not mounted after boot, by looking with mount (packages such as cgroup-lite can exist)

- The implementation of cgroups requires a few, simple hooks into the rest of the kernel, none in performance-critical paths:
  - In boot phase (**init/main.c)** to preform various initializations.
  - In process creation and destroy methods, ***fork()*** and ***exit().***
  - A new file system of type "cgroup" (VFS)
  - Process descriptor additions (struct task_struct)
  - Add procfs entries:
    - For each process: /proc/pid/cgroup.
    - System-wide: /proc/cgroups

- The cgroup modules are **not** located in one folder but scattered in the kernel tree according to their functionality:
  - memory: mm/memcontrol.c
  - cpuset: kernel/cpuset.c.
  - net_prio: net/core/netprio_cgroup.c
  - devices: security/device_cgroup.c.
  - And so on.

cgroups and kernel namespaces

Note that the **cgroups** is not dependent upon namespaces; you can build cgroups **without** namespaces kernel support.

There was an attempt in the past to add "ns" subsystem (ns_cgroup, namespace cgroup subsystem); with this, you could mount a namespace subsystem by:

mount -t cgroup -ons.

This code it was removed in 2011 (by a patch by Daniel Lezcano).

See:

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a77aea92010acf54ad785047234418d5d68772e2

cgroups VFS

- Cgroups uses a Virtual File System
  - All entries created in it are not persistent and deleted after reboot.
- All cgroups actions are performed via filesystem actions (create/remove directory, reading/writing to files in it, mounting/mount options).
- For example:
  - cgroup inode_operations for cgroup mkdir/rmdir.
  - cgroup file_system_type for cgroup mount/unmount.
  - cgroup file_operations for reading/writing to control files.

Mounting cgroups

In order to use a filesystem (browse it/attach tasks to cgroups,etc) it must be mounted.

The control group can be mounted anywhere on the filesystem. Systemd uses /sys/fs/cgroup.

When mounting, we can specify with mount options (-o) which subsystems we want to use.

There are 11 cgroup subsystems (controllers) (kernel 3.9.0-rc4 , April 2013); **two** can be built as modules. (All subsystems are instances of **cgroup_subsys** struct)

cpuset_subsys - defined in kernel/cpuset.c.

freezer_subsys - defined in kernel/cgroup_freezer.c.

mem_cgroup_subsys - defined in mm/memcontrol.c; Aka memcg - memory control groups.

blkio_subsys - defined in block/blk-cgroup.c.

net_cls_subsys - defined in net/sched/cls_cgroup.c (can be built as a kernel module)

net_prio_subsys - defined in net/core/netprio_cgroup.c (can be built as a kernel module)

devices_subsys - defined in security/device_cgroup.c.

perf_subsys (perf_event) - defined in kernel/events/core.c

hugetlb_subsys - defined in mm/hugetlb_cgroup.c.

cpu_cgroup_subsys - defined in kernel/sched/core.c

cpuacct_subsys - defined in kernel/sched/core.c

Mounting cgroups – contd.

In order to mount a subsystem, you should first create a folder for it under /cgroup.

In order to mount a cgroup, you first mount some tmpfs root folder:

- *mount -t tmpfs tmpfs /cgroup*

Mounting of the memory subsystem, for example, is done thus:

- *mkdir /cgroup/memtest*
- *mount -t cgroup -o memory test /cgroup/memtest/*

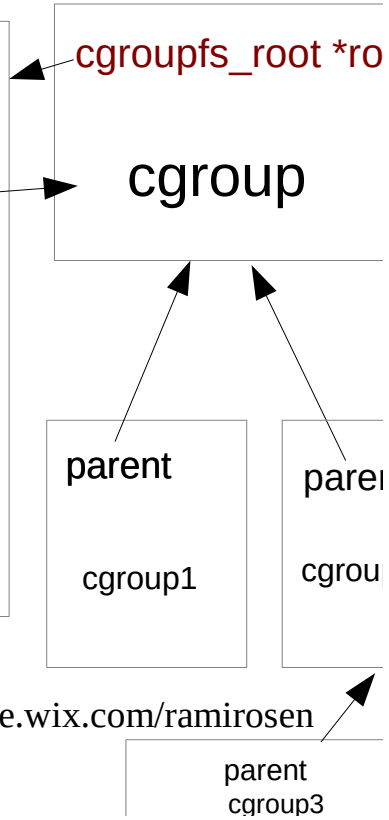
Note that instead “test” you can insert any text; this text is not handled by cgroups core. It's only usage is when displaying the mount by the **“mount”** command or by **cat /proc/mounts**.

Mounting cgroups – contd.

- Mount creates **cgroupfs_root** object + cgroup (**top_cgroup)** object
- mounting another path with the same subsystems - the same **subsys_mask**; the same **cgroupfs_root** object is reused.
- **mkdir** increments number_of_cgroups, **rmdir** decrements number_of_cgroups.
- cgroup1 - created by *mkdir /cgroup/memtest/cgroup1.*

struct **super_block *sb**
The super block being used. (in memory).
struct cgroup top_cgroup

unsigned long **subsys_mask**
bitmask of subsystems attached to this hierarchy
int **number_of_cgroups**



cgroupfs_root

Mounting a set of subsystems

From Documentation/cgroups/cgroups.txt:

If an active hierarchy with exactly the same set of subsystems already exists, it will be reused for the new mount.

If no existing hierarchy matches, and any of the requested subsystems are in use in an existing hierarchy, the mount will fail with -EBUSY.

Otherwise, a new hierarchy is activated, associated with the requested subsystems.

First case: Reuse

- mount -t tmpfs test1 /cgroup/test1
- mount -t tmpfs test2 /cgroup/test2
- mount -t cgroup -ocpu,cpuacct test1 /cgroup/test1
- mount -t cgroup -ocpu,cpuacct test2 /cgroup/test2
- This will work; the mount method recognizes that we want to use the same mask of subsytems in the second case.
  - (Behind the scenes, this is done by the return value of sget() method, called from ***cgroup_mount()***, found an already allocated superblock; the sget() makes sure that the mask of the sb and the required mask are identical)
  - Both will use the same **cgroupfs_root** object.
- This is exactly the first case described in Documentation/cgroups/cgroups.txt

Second case: any of the requested subsystems are in use

- mount -t tmpfs tmpfs /cgroup/tst1/
- mount -t tmpfs tmpfs /cgroup/tst2/
- mount -t tmpfs tmpfs /cgroup/tst3/
- mount -t cgroup -o freezer tst1 /cgroup/tst1/
- mount -t cgroup -o memory tst2 /cgroup/tst2/
- mount -t cgroup -o freezer,memory tst3 /cgroup/tst3
  - Last command will give an error. (-EBUSY).

    The reason: these subsystems (controllers) were been separately mounted.
- This is exactly the second case described in Documentation/cgroups/cgroups.txt

Third case - no existing hierarchy

no existing hierarchy matches, and none of the requested subsystems are in use in an existing hierarchy:

mount -t cgroup -o net_prio netpriotest /cgroup/net_prio/

Will succeed.

- under each new cgroup which is created, these 4 files are always created:
  - tasks
    - list of pids which are attached to this group.
  - cgroup.procs.
    - list of thread group IDs (listed by TGID) attached to this group.
  - cgroup.event_control.
    - Example in following slides.
  - notify_on_release (boolean).
    - For a newly generated cgroup, the value of **notify_on_release** in inherited from its parent; However, changing **notify_on_release** in the parent does not change the value in the children he already has.
    - Example in following slides.
- For the **topmost cgroup root object only**, there is also a **release_agent** – a command which will be invoked when the last process of a cgroup terminates; the *notify_on_release* flag should be set in order that it will be activated.

- Each subsystem adds specific control files for its own needs, besides these 4 fields. All control files created by cgroup subsystems are given a prefix corresponding to their subsystem name. For example:

cpuset subsystem

cpuset.cpus
cpuset.mems
cpuset.cpu_exclusive
cpuset.mem_exclusive
cpuset.mem_hardwall
cpuset.sched_load_balance
cpuset.sched_relax_domain_level
cpuset.memory_migrate
cpuset.memory_pressure
cpuset.memory_spread_page
cpuset.memory_spread_slab
cpuset.memory_pressure_enabled

devices subsystem

devices.allow
devices.deny
devices.list

cpu subsystem

cpu subsystem

cpu.shares	(only if CONFIG_FAIR_GROUP_SCHED is set)
cpu.cfs_quota_us	(only if CONFIG_CFS_BANDWIDTH is set)
cpu.cfs_period_us	(only if CONFIG_CFS_BANDWIDTH is set)
cpu.stat	(only if CONFIG_CFS_BANDWIDTH is set)
cpu.rt_runtime_us	(only if CONFIG_RT_GROUP_SCHED is set)
cpu.rt_period_us	(only if CONFIG_RT_GROUP_SCHED is set)

memory subsystem

memory.usage_in_bytes
memory.max_usage_in_bytes
memory.limit_in_bytes
memory.soft_limit_in_bytes
memory.failcnt
memory.stat
memory.force_empty
memory.use_hierarchy
memory.swappiness
memory.move_charge_at_immigrate
memory.oom_control

memory subsystem

up to 25 control files

memory.numa_stat	(only if CONFIG_NUMA is set)
memory.kmem.limit_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.usage_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.failcnt	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.max_usage_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.tcp.limit_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.tcp.usage_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.tcp.failcnt	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.tcp.max_usage_in_bytes	(only if CONFIG_MEMCG_KMEM is set)
memory.kmem.slabinfo	(only if CONFIG_SLABINFO is set)
memory.memsw.usage_in_bytes	(only if CONFIG_MEMCG_SWAP is set)
memory.memsw.max_usage_in_bytes	(only if CONFIG_MEMCG_SWAP is set)
memory.memsw.limit_in_bytes	(only if CONFIG_MEMCG_SWAP is set)
memory.memsw.failcnt	(only if CONFIG_MEMCG_SWAP is set)

blkio subsystem

blkio.weight_device
blkio.weight
blkio.weight_device
blkio.weight
blkio.leaf_weight_device
blkio.leaf_weight
blkio.time
blkio.sectors
blkio.io_service_bytes
blkio.io_serviced
blkio.io_service_time
blkio.io_wait_time
blkio.io_merged
blkio.io_queued
blkio.time_recursive
blkio.sectors_recursive
blkio.io_service_bytes_recursive
blkio.io_serviced_recursive
blkio.io_service_time_recursive
blkio.io_wait_time_recursive
blkio.io_merged_recursive
blkio.io_queued_recursive

blkio.avg_queue_size (only ifCONFIG_DEBUG_BLK_CGROUP is set)
blkio.group_wait_time (only ifCONFIG_DEBUG_BLK_CGROUP is set)
blkio.idle_time (only ifCONFIG_DEBUG_BLK_CGROUP is set)
blkio.empty_time (only ifCONFIG_DEBUG_BLK_CGROUP is set)
blkio.dequeue (only ifCONFIG_DEBUG_BLK_CGROUP is set)
blkio.unaccounted_time (only ifCONFIG_DEBUG_BLK_CGROUP is set)

blkio.throttle.read_bps_device
blkio.throttle.write_bps_device
blkio.throttle.read_iops_device
blkio.throttle.write_iops_device
blkio.throttle.io_service_bytes
blkio.throttle.io_serviced

netprio

net_prio.ifpriomap
net_prio.prioidx

Note the netprio_cgroup.ko should be insmoded so the mount will succeed. Moreover, rmmod will fail if netprio is mounted

- When mounting a cgroup subsystem (or a set of cgroup subsystems) , ***all*** *processes in the system belong to it (the top cgroup object).*

- *After mount -t cgroup -o memory test /cgroup/memtest/*
  - *you can see all tasks by: cat /cgroup/memtest/tasks*
  - When creating new child cgroups in that hierarchy, each one of them will **not** have any tasks at all initially.
  - Example:
  - mkdir /cgroup/memtest/group1
  - mkdir /cgroup/memtest/group2
  - cat /cgroup/memtest/group1/tasks
    - Shows nothing.
  - cat /cgroup/memtest/group2/tasks
    - Shows nothing.

•Any task can be a member of exactly one cgroup in a specific hierarchy.
•Example:
•*echo $$ > /cgroup/memtest/group1/tasks*
•*cat /cgroup/memtest/group1/tasks*
•*cat /cgroup/memtest/group2/tasks*
•Will show that task **only** in group1/tasks.
•After:
•*echo $$ > /cgroup/memtest/group2/tasks*
•The task was moved to group2; we will see that task it only in group2/tasks.

Removing a child group

Removing a child group is done by rmdir.
We cannot remove a child group in these two cases:
•When it has processes attached to it.
•When it has children.
We will get -EBUSY error in both cases.

Example 1 - processes attached to a group:
echo $$ > /cgroup/memtest/group1/tasks
rmdir /cgroup/memtest/group1
rmdir: failed to remove `/cgroup/memtest/group1': Device or resource busy

Example 2 - group has children:
mkdir /cgroup/memtest/group2/childOfGroup2
cat /cgroup/memtest/group2/tasks
- to make sure that there are no processes in group2.
rmdir /cgroup/memtest/group2/
rmdir: failed to remove `/cgroup/memtest/group2/': Device or resource busy

- Nesting is allowed:
  - *mkdir /cgroup/memtest/0/FirstSon*
  - *mkdir /cgroup/memtest/0/SecondSon*
  - *mkdir /cgroup/memtest/0/ThirdSon*
- *However, there are subsystems which will emit a kernel warning when trying to nest; in this subsystems, the .broken_hierarchy boolean member of cgroup_subsys is set explicitly to true.*

For example:

```
struct cgroup_subsys devices_subsys = {
.name = "devices",
...
.broken_hierarchy = true,
}
```

BTW, a recent patch removed it; in latest git for-3.10 tree, the only subsystem with broken_hierarchy is blkio.

broken_hierarchy example

- typing:
- *mkdir /sys/fs/cgroup/devices/0*
- Will omit no error, but if afterwards we will type:
- *mkdir /sys/fs/cgroup/devices/0/firstSon*
- We will see in the kernel log this warning:
- cgroup: mkdir (4730) created nested cgroup for controller "devices" which has incomplete hierarchy support. Nested cgroups may change behavior in the future.

- In this way, we can mount any one of the 11 cgroup subsystems (controllers) under it:
- *mkdir /cgroup/cpuset*
- *mount -t cgroup -ocpuset cpuset_group /cgroup/cpuset/*
- *Also here, the "cpuset_group" is only for the mount command,*
  - *So this will also work:*
  - *mkdir /cgroup2/*
  - mount -t tmpfs cgroup2_root /cgroup2
  - *mkdir /cgroup2/cpuset*
  - *mount -t cgroup -ocpuset mytest /cgroup2/cpuset*
  -

devices

- Also referred to as : devcg (devices control group)
- devices cgroup provides enforcing restrictions on opening and mknod operations on device files.
- 3 files: **devices.allow, devices.deny, devices.list.**
  - **devices.allow** can be considered as devices whitelist
  - **devices.deny** can be considered as devices blacklist.
  - **devices.list** available devices.
- Each entry is 4 fields:
  - **type**: can be a (all), c (char device), or b (block device).
    - All means all types of devices, and all major and minor numbers.
  - **Major number.**
  - **Minor number.**
  - **Access:** composition of 'r' (read), 'w' (write) and 'm' (mknod).

devices - example

/dev/null major number is 1 and minor number is 3 (You can fetch the major/minor number from Documentation/devices.txt)

mkdir /sys/fs/cgroup/devices/0

By default, for a new group, you have full permissions:

cat /sys/fs/cgroup/devices/0/devices.list

a *:* rwm

echo 'c 1:3 rmw' > /sys/fs/cgroup/devices/0/devices.deny

This denies rmw access from /dev/null deice.

echo $$ > /sys/fs/cgroup/devices/0/tasks

echo "test" > /dev/null

bash: /dev/null: Operation not permitted

echo a > /sys/fs/cgroup/devices/0/devices.allow

This adds the 'a *:* rwm' entry to the whitelist.

echo "test" > /dev/null

Now there is no error.

cpuset

- Creating a **cpuset** group is done with:
  - mkdir /sys/fs/cgroup/cpuset/0
    - You must be root to run this; for non root user, you will get the following error:
      - mkdir: cannot create directory ‘/sys/fs/cgroup/cpuset/0’: Permission denied
- **cpusets** provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks.

cpuset example

On Fedora 18, cpuset is mounted after boot on /sys/fs/cgroup/cpuset.

```
cd /sys/fs/cgroup/cpuset
mkdir test
cd test
/bin/echo 1 > cpuset.cpus
/bin/echo 0 > cpuset.mems
```

cpuset.cpus and cpuset.mems are not initialized; these two initializations are mandatory.

```
/bin/echo $$ > tasks
```

Last command moves the shell process to the new cpuset cgroup.

You cannot move a list of pids in a single command; you mush issue a separate command for each pid.

memcg (memory control groups)

Example:

mkdir /sys/fs/cgroup/memory/0

echo $$ > /sys/fs/cgroup/memory/0/tasks

echo 10M > /sys/fs/cgroup/memory/0/memory.limit_in_bytes

You can disable the out of memory killer with memcg:

echo 1 > /sys/fs/cgroup/memory/0/memory.oom_control

This disables the oom killer.

cat /sys/fs/cgroup/memory/0/memory.oom_control

oom_kill_disable 1

under_oom 0

- Now run some memory hogging process in this cgroup, which is known to be killed with oom killer in the default namespace.
- This process will **not** be killed.
- After some time, the value of under_oom will change to 1
- After enabling the OOM killer again:

```
echo 0 >   /sys/fs/cgroup/memory/0/memory.oom_control
```

You will get soon get the OOM “Killed” message.

Notification API

- There is an API which enable us to get notifications about changing status of a cgroup. It uses the ***eventfd()*** system call
- See man 2 eventfd
- It uses the fd of **cgroup.event_control**
- Following is a simple userspace app , “eventfd” (error handling was omitted for brevity)

Notification API – example

```
char buf[256];
int event_fd, control_fd, oom_fd, wb;
uint64_t u;
event_fd = eventfd(0, 0);
control_fd = open("cgroup.event_control", O_WRONLY);
oom_fd = open("memory.oom_control", O_RDONLY);
snprintf(buf, 256, "%d %d", event_fd, oom_fd);
write(control_fd, buf, wb);
close(control_fd);

for (;;) {
  read(event_fd, &u, sizeof(uint64_t));
  printf("oom event received from mem_cgroup\n");
}
```

Notification API – example (contd)

- Now run this program (eventfd) thus:
- From /sys/fs/cgroup/memory/0

  ./eventfd cgroup.event_control memory.oom_control

From a second terminal run:

cd /sys/fs/cgroup/memory/0/

echo $$ > /sys/fs/cgroup/memory/0/tasks

echo 10M > /sys/fs/cgroup/memory/0/memory.limit_in_bytes

Then run a memory hog problem.

When on OOM killer is invoked, you will get the messages from eventfd userspace program, **“oom event received from mem_cgroup”.**

release_agent example

- The release_agent is invoked when the last process of a cgroup terminates.
- The cgroup sysfs notify_on_release entry should be set so that release_agent will be invoked.
- A short script, /work/dev/t/date.sh:

```
#!/bin/sh
 date >> /work/log.txt
```

Run a simple process, which simply sleeps forever; let's say it's PID is pidSleepingProcess.

echo 1 > /sys/fs/cgroup/memory/notify_on_release

echo /work/dev/t/date.sh > /sys/fs/cgroup/memory/release_agent

mkdir /sys/fs/cgroup/memory/0/

echo pidSleepingProcess > /sys/fs/cgroup/memory/0/tasks

kill -9 pidSleepingProcess

This activates the release_agent; so we will see that the current time and date was written to /work/log.txt

Systemd and cgroups

• Systemd – developed by Lennart Poettering, Kay Sievers, others.
• Replacement for the Linux init scripts and daemon.
Adopted by Fedora (since Fedora 15), openSUSE , others.
• Udev was integrated into systemd.

• systemd uses control groups only for process grouping;
not for anything else like allocating resources like block io bandwidth, etc.

release_agent is a mount option on Fedora 18:
mount -a | grep systemd
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,**release_agent=/usr/lib/systemd/systemd-cgroups-agent**,name=systemd)

cgroup-agent is a short program (cgroups-agent.c) which all it does is send dbus message via the DBUS api.

dbus_message_new_signal()/dbus_message_append_args()/dbus_connection_send()

systemd Lightweight Containers new feature in Fedora 19: https://fedoraproject.org/wiki/Features/SystemdLightweightContainers

ls /sys/fs/cgroup/systemd/system

abrtd.service	crond.service	rpcbind.service
abrt-oops.service	cups.service	rsyslog.service
abrt-xorg.service	dbus.service	sendmail.service
accounts-daemon.service	firewalld.service	smartd.service
atd.service	getty@.service	sm-client.service
auditd.service	iprdump.service	sshd.service
bluetooth.service	iprinit.service	systemd-fsck@.service
cgroup.clone_children	iprupdate.service	systemd-journald.service
cgroup.event_control	ksmtuned.service	systemd-logind.service
cgroup.procs	mcelog.service	systemd-udevd.service
colord.service	NetworkManager.service	tasks
configure-printer@.service	notify_on_release	udisks2.service
console-kit-daemon.service	polkit.service	upower.service

We have here 34 services.

Example for bluetooth systemd entry:

ls /sys/fs/cgroup/systemd/system/bluetooth.service/

cgroup.clone_children cgroup.event_control cgroup.procs notify_on_release tasks

cat /sys/fs/cgroup/systemd/system/bluetooth.service/tasks
709

There are services which have more than one pid in the tasks control file.

- With fedora 18, default location of cgroup mount is: **/sys/fs/cgroup**
- **We have 9 controllers:**
- **/sys/fs/cgroup/blkio**
- **/sys/fs/cgroup/cpu,cpuacct**
- **/sys/fs/cgroup/cpuset**
- **/sys/fs/cgroup/devices**
- **/sys/fs/cgroup/freezer**
- **/sys/fs/cgroup/memory**
- **/sys/fs/cgroup/net_cls**
- **/sys/fs/cgroup/perf_event**
- **/sys/fs/cgroup/systemd**
- **In boot, systemd parses /sys/fs/cgroup and mounts all entries.**

/proc/cgroups

In Fedora 18, ***cat /proc/cgroups*** gives:

#subsys_name	hierarchy	num_cgroups	enabled
cpuset	2	1	1
cpu	3	37	1
cpuacct	3	37	1
memory	4	1	1
devices	5	1	1
freezer	6	1	1
net_cls	7	1	1
blkio	8	1	1
perf_event	9	1	1

Libcgroup

Libcgroup

libcgroup is a library that abstracts the control group file system in Linux.

libcgroup-tools package provides tools for performing cgroups actions.

Ubuntu:apt-get install cgroup-bin (tried on Ubuntu 12.10)

Fedora: yum install libcgroup

cgcreate creates new cgroup; **cgset** sets parameters for given cgroup(s); and **cgexec** runs a task in specified control groups.

Example:

cgcreate -g cpuset:/test

cgset -r cpuset.cpus=1 /test

cgset -r cpuset.mems=0 /test

cgexec -g* cpuset:/test *bash

One of the advantages of cgroups framework is that it is simple to add kernel modules which will work with. There are only two callback which we must implement, ***css_alloc()*** and ***css_free()***.
And there is no need to patch the kernel unless you do something special.
Thus, net/core/netprio_cgroup.c is only 322 lines of code and net/sched/cls_cgroup.c is 332 lines of code.

Checkpoint/Restart

Checkpointing is to the operation of a **Checkpointing** the state of a group of processes to a single file or several files.

Restart is the operation of restoring these processes at some future time by reading and parsing that file/files.

Attempts to merge Checkpoint/Restart in the Linux kernel failed:

Attempts to merge CKPT of openVZ failed:

Oren Laadan spent about three years for implementing checkpoint/restart in kernel; this code was not merged either.

Checkpoint and Restore In Userspace (CRIU)

- *A project of OpenVZ*
- *sponsored and supported by Parallels.*

Uses some kernel patches

http://criu.org/Main_Page

•**Workman**: (workload management)

It aims to provide high-level resource allocation and management implemented as a library but provides bindings for more languages (depends on the GObject framework ; allows all the library APIs to be exposed to non-C languages like Perl, Python, JavaScript, Vala).

https://gitorious.org/workman/pages/Home

•**Pax Controla Groupiana – a document:**
•Tries to define precautions that a software or user can take to avoid breaking or confusing other users of the cgroup filesystem.
http://www.freedesktop.org/wiki/Software/systemd/PaxControlGroups

• **aka "How to behave nicely in the cgroupfs trees"**

Note: in this presentation, we refer to two userspace package, iproute and util-linux. The examples are based on the most recent git source code of these packages.
You can check namespaces and cgroups support on your machine by running:
lxc-checkconfig
(from lxc package)

In Fedora 18 and Ubuntu 13.04, there is no support for User Namespaces though it is kernel 3.8

- On Android - Samsung Mini Galaxy:
  - *cat /proc/mounts | grep cgroup*

    none /acct cgroup rw,relatime,cpuacct 0 0

    none /dev/cpuctl cgroup rw,relatime,cpu 0 0

Links

Namespaces in operation series By Michael Kerrisk, January 2013:
part 1: namespaces overview
http://lwn.net/Articles/531114/

part 2: the namespaces API
http://lwn.net/Articles/531381/

part 3: PID namespaces
http://lwn.net/Articles/531419/

part 4: more on PID namespaces
http://lwn.net/Articles/532748/

part 5: User namespaces
http://lwn.net/Articles/532593/

part 6: more on user namespaces
http://lwn.net/Articles/540087/

Links - contd

Stepping closer to practical containers: "syslog" namespaces

http://lwn.net/Articles/527342/

- tree /sys/fs/cgroup/
- Devices implementation.
- Serge Hallyn nsexec

Capabilities - appendix

include/uapi/linux/capability.h

CAP_CHOWN
CAP_DAC_READ_SEARCH
CAP_FSETID
CAP_SETGID
CAP_SETPCAP
CAP_NET_BIND_SERVICE
CAP_NET_ADMIN
CAP_IPC_LOCK
CAP_SYS_MODULE
CAP_SYS_CHROOT
CAP_SYS_PACCT
CAP_SYS_BOOT
CAP_SYS_RESOURCE
CAP_SYS_TTY_CONFIG
CAP_LEASE
CAP_AUDIT_CONTROL
CAP_MAC_OVERRIDE
CAP_SYSLOG
CAP_BLOCK_SUSPEND
CAP_DAC_OVERRIDE
CAP_FOWNER
CAP_KILL
CAP_SETUID
CAP_LINUX_IMMUTABLE
CAP_NET_BROADCAST
CAP_NET_RAW
CAP_IPC_OWNER
CAP_SYS_RAWIO
CAP_SYS_PTRACE
CAP_SYS_ADMIN
CAP_SYS_NICE
CAP_SYS_TIME
CAP_MKNOD
CAP_AUDIT_WRITE
CAP_SETFCAP
CAP_MAC_ADMIN
CAP_WAKE_ALARM

See: man 8 setcap / man 8 getcap

Summary

- Namespaces
  - Implementation
  - UTS namespace
  - Network Namespaces
    - Example
  - PID namespaces
- cgroups
  - Cgroups and kernel namespaces
  - CGROUPS VFS
  - CPUSET
  - cpuset example
  - release_agent example
  - memcg
  - Notification API
  - devices
  - Libcgroup
- Checkpoint/Restart

Links

cgroups kernel mailing list archive:
http://blog.gmane.org/gmane.linux.kernel.cgroups

cgroup git tree:
git://git.kernel.org/pub/scm/linux/kernel/git/tj/cgroup.git

Thank you!