Multiple Instances of the Global Linux Namespaces

Eric W. Biederman
Linux Networx
ebiederman@lnxi.com

Abstract

Currently Linux has the filesystem namespace for mounts which is beginning to prove useful. By adding additional namespaces for process ids, SYS V IPC, the network stack, user ids, and probably others we can, at a trivial cost, extend the UNIX concept and make novel uses of Linux possible. Multiple instances of a namespace simply means that you can have two things with the same name.

For servers the power of computers is growing, and it has become possible for a single server to easily fulfill the tasks of what previously required multiple servers. Hypervisor solutions like Xen are nice but they impose a performance penalty and they do not easily allow resources to be shared between multiple servers.

For clusters application migration and preemption are interesting cases but almost impossibly hard because you cannot restart the application once you have moved it to a new machine, as usually there are resource name conflicts.

For users certain desktop applications interface with the outside world and are large and hard to secure. It would be nice if those applications could be run on their own little world to limit what a security breach could compromise.

Several implementations of this basic idea have been done succsessfully. Now the work is to create a clean implementation that can be merged into the Linux kernel. The discussion has begun on the *linux-kernel* list and things are slowly progressing.

1 Introduction

1.1 High Performance Computing

I have been working with high performance clusters for several years and the situation is painful. Each Linux box in the cluster is refered to as a node, and applications running or queued to run on a cluster are jobs.

Jobs are run by a batch scheduler and, once launched, each job runs to completion typically consuming 99% of the resources on the nodes it is running on.

In practice a job cannot be suspended, swapped, or even moved to a different set of nodes once it is launched. This is the oldest and most primitive way of running a computer. Given the long runs and high computation overhead of HPC jobs it isn't a bad fit for HPC environments, but it isn't a really good fit either.

Linux has much more modern facilities. What prevents us from just using them?

HPC jobs currently can be suspended, but that just takes them off the cpu. If you have sufficient swap there is a chance the jobs can even be pushed to swap but frequently these application lock pages in memory so they can be ensured of low latency communication.

The key problem is simply having multiple machines and multiple kernels. In general, how to take an application running under one Linux kernel and move it completely to another kernel is an unsolved problem.

The problem is unsolved not because it is fundamentally hard, but simply because it has not be a problem in the UNIX environment. Most applications don't need multiple machines or even big machines to run on (especially with Moore's law exponentially increasing the power of small machines). For many of the rest the large multiprocessor systems have been large enough.

What has changed is the economic observation that a cluster of small commodity machines is much cheaper and equally as fast as a large supercomputer.

The other reason this problem has not been solved (besides the fact that most people working on it are researchers) is that it is not immediately obvious what a general solution is. Nothing quite like it has been done before so you can't look into a text book or into the archives of history and know a solution. Which in the broad strokes of operating system theory is a rarity.

The hard part of the problem also does not lie in the obvious place people first look— how to save all of the state of an application. Instead, the problem is how do you restore a saved application so it runs successfully on another machine.

The problem with restoring a saved application is all of the global resources an application uses. Process ids, SYS V IPC identifiers, filenames, and the like. When you restore an application on another machine there is no guarantee that it can reuse the same global identifiers as another process on that machine may be using those identifiers.

There are two general approaches to solving this problem. Modifying things so these global machine identifiers are unique across all machines in a cluster, or modifying things so these machine global identifiers can be repeated on a single machine. Many attempts have been made to scale global identifiers cluster-wide—Mosix, OpenSSI, bproc, to name a few—and all of them have had to work hard to scale. So I choose to go with an implementation that will easily scale to the largest of clusters, with no communication needed to do so.

This has the added advantage that in a cluster it doesn't change the programming model presented to the user. Just some machines will now appear as multiple machines. As the rise of the internet has shown building applications that utilize multiple machines is not foreign to the rest of the computing world either.

To make this happen I need to solve the challenging problem of how to refactor the UNIX/Linux API so that we can have multiple instances of the global Linux namespaces. The Plan 9 inspired mount/filesystem namespace has already proved how this can be done and is slowly proving useful.

1.2 Jails

Outside the realm of high perfomance computing people have been restricting their server application to chroot jails for years. The problems with chroot jails have become well understood and people have begun fixing them. First BSD jails, and then Solaris containers are some of the better known examples.

Under Linux the open source community has not been idle. There is the linux-jail project, Vserver, Openvz, and related projects like SELinux, UML, and Xen.

Jails are a powerful general purpose tool useful for a great variety of things. In resource utilization jails are cheap, dynamically loading glibc is likely to consume more memory than the additional kernel state needed to track a jail. Jails allow applications to be run in simple stripped down environments, increasing security, and decreasing maintenance costs, while leaving system administrators with the familiar UNIX environment.

The only problem with the current general purpose implementation of jails under Linux is nothing has been merged into the mainline kernel, and the code from the various projects is not really mergable as it stands. The closest I have seen is the code from the linux-jail project, and that is simply because it is less general purpose and implemented completely as a Linux security module.

Allowing multiple instances of global names which is needed to restore a migrated application is a more constrained problem than that of simply implementing a jail. But a jail that ensures you can have multiple instances of all of the global names is a powerful general purpose jail that you can run just about anything in. So the two problems can share a common kernel solution.

1.3 Future Directions

A cheap and always available jail mechanism is also potentially quite useful outside the realm of high performance computing and server applications. A general purupose checkpoint/restart mechanism can allow desktop users to preserve all of their running applications when they log out. Vulnerable or untrusted applications like a web browser or an irc client could be contained so that if they are attacked all that is gained is the ability to browse the web and draw pictures in an X window.

There would finally be answer to the age old question: How do I preserve my user space while upgrading my kernel?

All this takes is an eye to designing the interfaces so they are general purpose and nestable. It should be possible to have a jail inside a jail inside a jail forever, or at least until there don't exist the resources to support it.

2 Namespaces

How much work is this? Looking at the existing patches it appears that 10,000 to 20,000 lines of code will ultimately need to be touched. The core of Linux is about 130,000 lines of code, so we will need to touch between 7% and 15% of the core kernel code. Which clearly indicates that one giant patch to do everything even if it was perfect would be rejected simply because it is too large to be reviewed.

In Linux there are multiple classes of global identifiers (i.e. process id, SYS V IPC keys, user ids). Each class of identifier can be thought of living in its own namespace.

This gives us a natural decomposition to the problem, allowing each namespace to be modified separately so we can support multiple instances of that namespace. Unfortunately this also increases the difficulty of the problem, as we need to modify the kernel's reporting and configuration interfaces to support multiple instances of a namespace instead of having them tightly coupled.

The plan then is simple. Maintain backwards compatibility. Concentrate on one namespace at a time. Focus on implementing the ability to have multiple objects of a given type, with the same name. Configure a namespace from the inside using the existing interfaces. Think of these things ultimately not as servers or virtual machines but as processes with peculiar attributes. As far as possible implement the namespaces so that an application can be given the capability bit for that allows full control over a namespace and still not be able to escape. Think in terms of a recursive implementation so we always keep in mind what it takes to recurse indefinitely.

What the system call interface will be to create a new instance of a namespace is still up for debate. The current contenders are a new `CLONE_` flag or individual system calls. I personally think a flag to `clone` and `unshare` is all that is needed but if arguments are actually needed a new system call makes sense.

Currently I have identified ten separate namespaces in the kernel. The filesystem mount namespace, uts namespace, the SYS V IPC namespace, network namespace, the pid namespace, the uid namespace, the security namespace, the security keys namespace, the device namespace, and the time namespace.

2.1 The Filesystem Mount Namespace

Multiple instances of the filesystem mount namespace are already implemented in the stable Linux kernels so there are few real issues with implementing it. There are still outstanding question on how to make this namespace usable and useful to unpriviledged processes, as well as some ongoing work to allow the filesystem mount namespace to allow bind mounts to have bind flags. For example, so the the bind mount can be restricted read only when other mounts of the filesystem are still read/write.

```
int uname(struct utsname *buf);
struct utsname {
        char sysname[];
        char nodename[];
        char release[];
        char version[];
        char machine[];
        char domainname[];
};
```

Figure 1: uname

`CAP_SYS_ADMIN` is currently required to modify the mount namespace, although there has been some discussion on how to relax the restrictions for bind mounts.

2.2 The UTS Namespace

The UTS namespace characterizes and identifies the system that applications are running on. It is characterizeed by the `uname` system call. `uname` returns six strings describing the current system. See Figure 1.

The returned `utsname` structure has only two members that vary at runtime `nodename` and `domainname`. `nodename` is the classic hostname of a system. `domainname` is the NIS domainname. `CAP_SYS_ADMIN` is required to change these values, and when the system boots they start out as `"(none)"`.

The pieces of the kernel that report and modify the `utsname` structure are not connected to any of the big kernel subsystems, build even when `CONFIG_NET` is disabled, and use `CAP_SYS_ADMIN` instead of one of the more specific capabilities. This clearly shows that the code has no affiliation with one of the larger namespaces.

Allowing for multiple instances of the UTS namespace is a simple matter of allocating a

new copy of `struct utsname` in the kernel for each different instance of this namespace, and modifying the system calls that modify this value to lookup the appropriate instance of `struct utsname` by looking at `current`.

2.3 The IPC Namespace

The SYS V interprocess communication namespace controls access to a flavor of shared memory, semaphores, message queues introduced in SYS V UNIX. Each object has associated with it a `key` and an `id`, and all objects are globally visible and exist until they are explicitly destroyed.

The `id` values are unique for every object of that type and assigned by the system when the object is created.

The `key` is assigned by the application usually at compile time and is unique unless it is specified as `IPC_PRIVATE`. In which case the `key` is simply ignored.

The ipc namespace is currently limited by the following universally readable and uid 0 setable `sysctl` values:

- `kernel.shmmax` The maximum shared memory segment size.
- `kernel.shmall` The maximum combined size of all shared memory segments.
- `kernel.shmni` The maximum number of shared memory segments.
- `kernel.msgmax` The maximum message size.
- `kernel.msgmni` The maximum number of message queues.
- `kernel.msgmnb` The maximum number of bytes in a message queue.
- `kernel.sem` An array of 4 control integers
  - `sc_semmsl` The maximum number of semaphores in a semaphore set.
  - `sc_semmns` The maximum number of semaphores in the system.
  - `sc_semopm` The maximum number of semaphore operations in a single system call.
  - `sc_semmni` The maximum number of semaphore sets in the system.

Operations in the ipc namespace are limited by the following capabilities:

- `CAP_IPC_OWNER` Allows overriding the standard ipc ownership checks, for the following operations: `shm_attach`, `shm_stat`, `shm_get`, `msgrcv`, `msgsnd`, `msg_stat`, `msg_get`, `semtimedop`, `sem_getall`, `sem_setall`, `sem_stat`, `sem_getval`, `sem_getpid`, `sem_getncnt`, `sem_getzcnt`, `sem_setval`, `sem_get`.

  For filesystems `namei.c` uses `CAP_DAC_OVERRIDE`, and `CAP_DAC_READ_SEARCH` to provide the same level of control.
- `CAP_IPC_LOCK` Required to control locking of shared memory segments in memory.
- `CAP_SYS_RESOURCE` Allows setting the maximum number of bytes in a message queue to exceed `kernel.msgmnb`
- `CAP_SYS_ADMIN` Allows changing the ownership and removing any ipc object.

Allowing for multiple instances of the ipc namespace is a straightforward process of duplicating the tables used for lookup by `key` and

`id`, and modifying the code to use `current` to select the appropriate table. In addition the `sysctl`s need to be modified to look at `current` and act on the corresponding copy of the namespace.

The ugly side of working with this namespace is the capability situation. `CAP_IPC_OWNER` trivially becomes restricted to the current ipc namespace. `CAP_IPC_LOCK` still remains dangerous. `CAP_DAC_OVERRIDE` and `CAP_DAC_READ_SEARCH` might be ok, it really depends on the state of the filesystem mount namespace. `CAP_SYS_RESOURCE` and `CAP_SYS_ADMIN` are still unsafe to give to untrusted applications.

2.4 The Network Namespace

By volume the code implementing the network stack is the largest of the subsystems that needs its own namespace. Once you look at the network subsystem from the proper slant it is straightforward to allow user space to have what appears to be multiple instances of the network stack, and thus you have a network namespace.

The core abstractions used by the network stack are processes, sockets, network devices, and packets. The rest of the network stack is defined in terms of these.

In adding the network namespace I add a few simple rules.

- A network device belongs to exactly one network namespace.
- A socket belongs to exactly one network namespace.

A packet comes into the network stack from the outside world through a network device. We can the look at that device to find the network namespace and the rules to process that packet by.

We generate a packet and feed it to the kernel through a socket. The kernel looks at the socket, finds the network namespace, and from there the rules to process the packet by.

What this means is that most of the network stack global variables need to be moved into the network namespace data structure. Hopefully we can write this so the extra level of indirection will not reduce the performance of the network stack.

Looking up the network stack global variables through the network namespace is not quite enough. Each instance of the network namespace needs its own copy of the loopback device, an interface needs to be added to move network devices between network namespaces, and a two headed tunnel device (a cousin of the loopback device) needs to be added so we can send packets between different network namespaces.

With these in place it is safe to give processes a separate network namespace: `CAP_NET_BIND_SERVICE`, `CAP_NET_BROADCAST`, `CAP_NET_ADMIN`, and `CAP_NET_RAW`. All of the functionality will work and working through the existing network devices there won't be an ability to escape the network namespace. Care should be given to giving an untrusted process access to real network devices, though, as hardware or software bugs in the implementation of that network device could be reduce the security.

The future direction of the network stack is towards Van Jackobson network channels, where more of the work is pushed towards process context, and happening in sockets. That work appears to be a win for network namespaces

in two ways. More work happening in process context and in well defined sockets means it is easier to lookup the network namespace, and thus cheaper. Having a lightweight packet classifier in the network drivers should allow a single network device to appear to user space as multiple network devices each with a different hardware address. Then based upon the destination hardware address the packet can be placed in one of several different network namespaces. Today to get the same semantics I need to configure the primary network device as a router, or configure ethernet bridging between the real network and the network interface that sends packets to the secondary network namespace.

2.5 The Process Id Namespace

The venerable process id is usually limited to 16bits so that the bitmap allocator is efficient and so that people can read and remember the ids. The identifiers are allocated by the kernel, and identifiers that are no longer in use are periodically reused for new processes. A single identifier value can refer to a process, a thread group, a process group and to a session, but every use starts life as a process identifier.

The only capability associated with process ids is `CAP_KILL` which allows sending signals to any process even if the normal security checks would not allow it.

Process identifiers are used to identify the current process, to identify the process that died, to specify a process or set of processes to send a signal to, to specify a process or set of processes to modify, to specify a process to debug, and in system monitoring tools to specify which process the information pertains to. Or in other words process identifiers are deeply entrenched in the user/kernel interface and are used for just about everything.

In a lot of ways implementing a process id namespace is straightforward as it is clear how everything should look from the inside. There should be a group of all of the processes in the namespace that `kill -1` sends signals to. Either a new pid hash table needs to be allocated or the key in the pid hash table needs to be modified to include the pid namespace. A new pid allocation bitmap needs to be allocated. `/proc/sys/pid_max` needs to be modified to refer to the current pid allocation bitmap. When the pid namespace exits all of the processes in the pid namespace need to be killed. Kernel threads need to be modified to never start up in anything except the default pid namespace. A process that has pid 1 must exist that will not receive any signals except for the ones it installs a signal handler for.

How a pid namespace should look from the outside is a much more delicate question. How should processes in a non default pid namespace be displayed in `/proc`? Should any of the process in a pid namespace show up in any other pid namespace? How much of the existing infrastructure that takes pids should continue to work?

This is one of the few areas where the discussion on the kernel list has come to a complete standstill, as an inexpensive technical solution to everyones requirements was not visible at the time of the conversation.

A big piece of what makes the process id namespace different is that processes are organized into a hierarchical tree. Maintaining the parent/child relationship between the process that initiates the pid namespace and the first process in the new pid namespace requires first process in the new pid namespace have two pids. A pid in the namespace of the parent and pid 1 in its own namespace. This results in namespaces that are hierarchical unlike most namespaces that are completely disjoint.

Having one process with two pids looks like a serious problem. It gets worse if we want that process to show up in other pid namespaces.

After looking deeply at the underlying mechanisms in the kernel I have started moving things away from `pid_t` to pointers to `struct pid`. The immediate gain is that the kernel becomes protected from pid wraparound issues.

Once all of the references that matter are `struct pid` pointers inside the kernel a different implementation becomes possible. We can hang multiple <pid namespace, `pid_t`> tuples off `struct pid` allowing us to have a different name for the same pid in several pid namespaces.

With processes in subordinate pid namespaces at least potentially showing up when we need them we can preserve the existing UNIX api for all functions that take pids and not need to reinvent pid namespace specific solutions.

The question yet to be answered in my mind is do we always map a process's `struct pid` into all of its ancestor's pid namespaces, or do we provide a mechanism that performs those mappings on demand?

2.6 The User and Group ID Namespace

In the kernel user ids are used for both accounting and for for performing security checks. The per user accounting is connected to the `user_struct`. Security checks are done against `uid`, `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid`, `fsgid`, processes capabilities, and variables maintained by a Linux security module. The kernel allows any of the uid/gid values to rotate between slots, or, if a process has `CAP_SETUID`, arbitrary values to be set into the filesystem uids.

With a uid namespace the security checks for equality of uids become checks to ensure the entire tuple <uid namespace, uid> is equal. Which means if two uids are in different namespaces the check will always fail. So the only permitted cases across uid namespaces will be when everyone is allowed to perform the action the process is trying to perform or when the process has the appropriate capability to perform the action on any process.

An alternative in some cases to modifying all of the checks to be against <namespace, uid> tuples is to modify some of the checks to be against `user_struct` pointers.

Since uid namespaces are not persistent, mapping of a uid namespace to filesystems requires some new mechanisms. The primary mechanism is to associate with the each `super_block` the uid namespace of the filesystem; probably moving that information into each `struct inode` in the kernel for speed and flexibility.

To allow sharing of filesystem mounts between different uid namespaces requires either using acls to tag inodes with non-default filesystem namespace information or using the key infrastructure to provide a mapping between different uid namespaces.

Virtual filesystems require special care as frequently they allow access to all kinds of special kernel functionality without any capability checks if the uid of a process equals 0. So virtual filesystems like proc and sysfs must specify the default kernel uid namespace in their `superblock` or it will be trivial to violate the kernel security checks.

There is a question of whether the change in rules and mechanisms should take place in the core kernel code, making it uid namespace aware, or in a Linux security module. A key of that decision is the uid hash table and `user_struct`. From my reading of the kernel code it appears that current Linux security

modules can only further restrict the default kernel permissions checks and there is not a hook that makes it possible to allocate a different `user_struct` depending on security module policies.

Which means at least the allocation of `user_struct`, and quite possibly making all of the uid checks fail if the uid namespaces are not equal, should happen in the core of the kernel with security modules standing in the background providing really advanced facilities.

With a uid namespace it becomes safe to give untrusted users `CAP_SETUID` without reducing security.

2.7 Security Modules and Namespaces

There are two basic approaches that can be pursued to implement multiple instances of user space. Objects in the kernel can be isolated by policy and security checks with security modules, or they can be isolated by making visible only the objects you are allowed to access by using namespaces.

The Linux Jail module (`http://sf.net/projects/linuxjail`) implemented by "Serge E. Hallyn" <serue@us.ibm.com> is a good example of what can be done with just a security module and isolating a group of processes with permission checks and policy rather than simply making the inaccessible parts of the system disappear.

Following that general principle Linux security modules have two different roles they can play when implementing multiple instances of user space. They can make up for any unimplemented kernel namespace by isolating objects with additional permission checks, which is good as a short term solution. Linux security modules modified to be container aware can also provide for enhanced security enforcement mechanisms in containers. In essence this second modification is the implementation of a namespace for security mechanisms and policy.

2.8 The Security Keys Namespace

Not long ago someone added to the kernel what is the frustration of anyone attempting to implementing namespaces to allow for the migration of user space. Another obscure and little known global namespace.

In this case each key on a key ring is assigned a global `key_serial_t` value. `CAP_SYS_ADMIN` is used to guard ownership and permission changes.

I have yet to look in detail but at first glance this looks like one of the easy cases, where we can just simply implement another copy of the lookup table. It appears the `key_serial_t` values are just used for manipulation of the security keys, from user space.

2.9 The Device Namespace

Not giving children in containers `CAP_SYS_MKNOD` and not mounting sysfs is sufficient to prevent them from accessing any device nodes that have not been audited for use by that container. Getting a new instance of the uid/gid namespace is enough to remove access from magic sysfs entries controlling devices although there is some question on how to bring them back.

For purposes of migration, unless all devices a set of processes has access to are purely virtual, pretending the devices haven't changed is nonsense. Instead it makes much more sense to explicitly acknowledge the devices have changed

and send hotplug remove and add events to the set of processes.

With the use of hotplug events the assumption that the global major and minor numbers that a device uses are constant is removed.

Equally as sensitive as `CAP_SYS_MKNOD`, and probably more important if mounting sysfs is allowed, is `CAP_CHOWN`. It allows changing the owner of a file. Since it would be required to change the sysfs owner before a sensitive file could be accessed.

So in practice managing the device namespace appears to be a user space problem with restrictions on `CAP_SYS_MKNOD` and `CAP_CHOWN` being used to implement the filter policy of which devices a process has access to.

2.10 The Time Namespace

The kernel provides access to several clocks `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, and `CLOCK_THREAD_CPUTIME_ID` being the primaries.

`CLOCK_REALTIME` reports the current wall clock time.

`CLOCK_MONOTONIC` is similar to `CLOCK_REALTIME` except it cannot be set and thus never backs up.

`CLOCK_PROCESS_CPUTIME_ID` reports how much aggregate cpu time the threads in a process have consumed.

`CLOCK_THREAD_CPUTIME_ID` reports how much cpu time an individual thread has consumed.

If process migration is not a concern none of these clocks except possibly `CLOCK_REALTIME` is interesting. In the context of process migration all of these clocks become interesting.

The thread and process clocks simply need an offset field so the amount of time spent on the previous machine can be added in. So that we can prevent the clocks from going backwards.

The monotonic timer needs an offset field so that we can guarantee that it never goes backwards in the presence of process migration.

The realtime clock matters the least but having an additional offset field for clock adds additional flexibility to the system and comes at practically no cost.

All of the clocks except for `CLOCK_MONOTONIC` support setting the clock with `clock_settime` so the existing control interfaces are sufficient. For the monotonic clock things are different, `clock_settime` is not allowed to set the clock, ensuring that the time will never run backwards, and there is a huge amount of sense in that logic.

The only reasonable course I can see is setting the monotonic clock when the time namespace is created. Which probably means we will need a syscall (and not a clone flag) to create the clone flag and we should provide it with minimum acceptable value of the monotonic clock.

3 Modifications of Kernel Interfaces

One of the biggest challenges with implementing multiple namespaces is how do we modify the existing kernel interfaces in a way that retains backwards compatibility for existing applications while still allowing the reality of the new situation to be seen and worked with.

Modifying existing system calls is probably the easiest case. Instead of reference global variables we reference variables through `current`.

Modifying `/proc` is trickier. Ideally we would introduce a new subdirectory for each class of namespace, and in that folder list each instance of that namespace, adding symbolic links from the existing names where appropriate. Unfortunately it is not possible to obtain a list of namespaces and the extra maintenance cost does not yet seem to just the extra complexity and cost of a linked list. So for proc what we are likely to see is the information for each namespace listed in `/proc/pid` with a symbolic link from the existing location into the new location under `/proc/self/`.

Modifying `sysctl` is fairly straightforward but a little tricky to implement. The problem is that `sysctl` assumes it is always dealing with global variables. As we put those variables into namespaces we can no longer store the pointer into a global variable. So we need to modify the implementation of `sysctl` to call a function which takes a `task_struct` argument to find where the variable is located. Once that is done we can move `/proc/sys` into `/proc/pid/sys`.

Modifying `sysfs` doesn't come up for most of the namespaces but it is a serious issue for the network namespace. I haven't a clue what the final outcome will be but assuming we want global visibility for monitor applications something like `/proc/pid` and `/proc/self` needs to be added so we can list multiple instances and add symbolic links from their old location in `sysfs`.

The `netlink` interface is the most difficult kernel interface to work with. Because control and query packets are queued and not necessarily processed by the application that sends the query, getting the context information necessary to lookup up the appropriate global variables is a challenge. It can even be difficult to figure out which port namespace to reply to. As long as the only users of netlink are part of the networking stack I have an implementation that solves the problems. However, as netlink has been suggested for other things, I can't count on just the network stack processing packets.

Resource counters are one of the more challenging interfaces to specify. Deciding if an interface should be per namespace or global is a challenge, and answering the question how does this work when we have recursive instances of a namespace. All of these concerns are exemplified when there are multiple untrusted users on the system. For starters we should be able to punt and implement something simple and require `CAP_SYS_RESOURCE` if there are any per namespace resource limits. Which should leave us with a simple and correct implementation. Then the additional concerns can be addressed from there.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin