# Adding Generic Process Containers to the Linux Kernel

Paul B. Menage*
*Google, Inc.*
menage@google.com

## Abstract

While Linux provides copious monitoring and control options for individual processes, it has less support for applying the same operations efficiently to related groups of processes. This has led to multiple proposals for subtly different mechanisms for process aggregation for resource control and isolation. Even though some of these efforts could conceptually operate well together, merging each of them in their current states would lead to duplication in core kernel data structures/routines.

The Containers framework, based on the existing cpusets mechanism, provides the generic process grouping features required by the various different resource controllers and other process-affecting subsystems. The result is to reduce the code (and kernel impact) required for such subsystems, and provide a common interface with greater scope for co-operation.

This paper looks at the challenges in meeting the needs of all the stakeholders, which include low overhead, feature richness, completeness and flexible groupings. We demonstrate how to extend containers by writing resource control and monitoring components, we also look at how to implement namespaces and cpusets on top of the framework.

## 1 Introduction

Over the course of Linux history, there have been and continue to be multiple efforts to provide forms of modified behaviour across sets of processes. These efforts have tended to fall into two main camps, resource control/monitoring, and namespace isolation (although some projects contain elements of both).

Technically resource control and isolation are related; both prevent a process from having unrestricted access to even the standard abstraction of resources provided by the Unix kernel. For the purposes of this paper, we use the following defintions:

*Resource control* is any mechanism which can do either or both of:

- tracking *how much* of a resource is being comsumed by a set of processes
- imposing quantative limits on that consumption, either absolutely, or just in times of contention.

Typically resource control is visible to the processes being controlled.

*Namespace isolation*[1] is a mechanism which adds an additional indirection or translation layer to the naming/visibility of some unix resource space (such as process ids, or network interfaces) for a specific set of processes. Typically the existence of isolation itself is invisible to the processes being isolated.

Resource Control and Isolation are both subsets of the general model of a subsystem which can apply behaviour differently depending on a process' membership of some specific group. Other examples could include:

- basic job control. A job scheduling system needs to be able to keep track of which processes are part of a given running "job," in the presence of `fork()` and `exit()` calls from processes in that job. This is the simplest example of the kind of

*With additional contributions by Balbir Singh and Srivatsa Vaddagiri, *IBM Linux Technology Center*, {balbir,vatsa}@in.ibm.com

[1] We avoid using the alternative term *virtualization* in this paper to make clear the distinction between lightweight in-kernel virtualization/isolation, and the much more heavyweight virtual machine hypervisors such as Xen which run between the kernel and the hardware.

process tracking system proposed in this paper, as the kernel need do nothing more than maintain the membership list of processes in the job.

- tracking memory pressure for a group of processes, and being able to receive notifications when memory pressure reaches some particular level (e.g. as measured by the scanning level reached in `try_to_free_pages()`), or reaches the OOM stage.

Different projects have proposed various basic mechanisms for implementing such process tracking. The drawbacks of having multiple such mechanisms include:

- different and mutually incompatible user-space and kernel APIs and feature sets.
- The merits of the underlying process grouping mechanisms and the merits of the actual resource control/isolation system become intertwined.
- systems that could work together to provide synergistic control of different resources to the same sets of processes are unable to do so easily since they're based on different frameworks.
- kernel structures and code get bloated with additional framework pointers and hooks.
- writers of such systems have to duplicate large amounts of functionally-similar code.

The aim of the work described in this paper is to provide a generalized process grouping mechanism suitable for use as a base for current and future Linux process-control mechanisms such as resource controllers; the intention is that writers of such controllers need not be concerned with the details of how processes are being tracked and partitioned, and can concentrate on the particular resource controller at hand, and the resource abstraction presented to the user. (Specifically, this framework *does not* attempt to prescribe any particular resource abstraction.) The requirements for a process-tracking framework to meet the needs of existing and future process-management mechanisms are enumerated, and a proposal is made that aims to satisfy these requirements.

For the purposes of this paper, we refer to a tracked group of processes as a *container*. Whether this is a suitable final name for the concept is currently the subject of debate on various Linux mailing lists, due to its use by other development groups for a similar concept in userspace. Alternative suggestions have included *partition* and *process set*.

## 2 Requirements

In this section we attempt to enumerate the properties required of a generic container framework. Not all mechanisms that depend on containers will require all of these properties.

### 2.1 Multiple Independent Subsystems

Clients of the container framework will typically be resource accounting/control systems and isolation systems. In this paper we refer to the generic client as a *subsystem*. The relationship between the container framework and a subsystem is similar to that between the Linux VFS and a specific filesystem—the framework handles many of the common operations, and passes notifications/requests to the subsystem.

Different users are likely to want to make use of different subsystems; therefore it should be possible to selectively enable different subsystems both at compile time and at runtime. The main function of the container framework is to allow a subsystem to associate some kind of policy/stats (referred to as the *subsystem state*) with a group of processes, without the subsystem having to worry in too much detail about how this is actually accomplished.

### 2.2 Mobility

It should be possible for an appropriately-privileged user to move a process between containers. Some subsystems may require that processes can only move into a container at the point when it is created (e.g. a virtual server system where the process becomes the new *init* process for the container); therefore it should be possible for mobility to be configurable on a per-subsystem basis.

### 2.3 Inescapability

Once a process has been assigned to a container, it shouldn't be possible for the process (or any of its children) to move to a different container without action by a privileged (i.e. root) user, or by a user to whom that capability has been delegated, e.g. via filesystem permissions.

### 2.4 Extensible User Interface

Different subsystems will need to present different configuration and reporting interfaces to userspace:

- a memory resource controller might want to allow the user to specify guarantees and limits on the number of pages that processes in a container can use, and report how many pages are actually in use.
- a memory-pressure tracker might want to allow the user to specify the particular level of memory pressure which should cause user notifications.
- the *cpusets* system needs to allow users to specify various parameters such as the bitmasks of CPUs and memory nodes to which processes in the container have access.

From the user's point of view it is simplest if there is at least some level of commonality between the configuration of different subsystems. Therefore the container framework should provide an interface that captures the common aspects of different subsystems but which still allows subsystems sufficient flexibility. Possible candidates include:

- A filesystem interface, where each subsystem can register files that the user can write (for configuration) and/or read (for reporting) has the advantages that it can be manipulated by many standard unix tools and library routines, has built-in support for permission delegation, and can provide arbitrary input/output formats and behaviour.
- An API (possibly a new system call?) that allows the user to read/write named properties would have the advantages that it would tend to present a more uniform interface (although possibly too restrictive for some subsystems) and potentially have slightly better performance than a filesystem-based interface.

### 2.5 Nesting

For some subsystems it is desirable for there to be multiple nested levels of containers:

- The existing cpusets system inherently divides and subdivides sets of memory nodes and CPUs between different groups of processes on the system.
- Nesting allows some fraction of the resources available to one set of processes to be delegated to a subset of those processes.
- Nested virtual servers may also find hierarchical support useful.

Some other subsystems will either be oblivious to the concept of nesting, or will actively want to avoid it; therefore the container system should allow subsystems to selectively control whether they allow nesting.

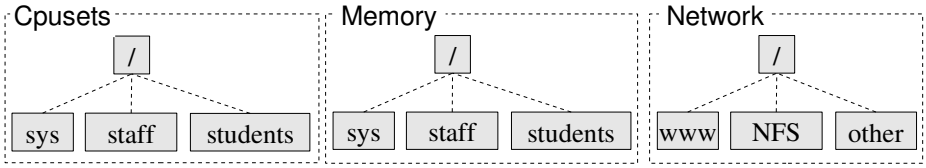
### 2.6 Multiple Partitions

The container framework will allow the user to partition the set of processes. Consider a system that is configured with the cpusets and beancounters subsystems. At any one time, a process will be in one and exactly one cpuset (a cpuset A may also be a child of some other cpuset B, in which case in a sense all processes in A are indirectly members of cpuset B as well, but a process is only a direct member of one cpuset). Similarly a process is only a direct member of one beancounter. So cpusets and beancounters are each a partition function on the set of processes.

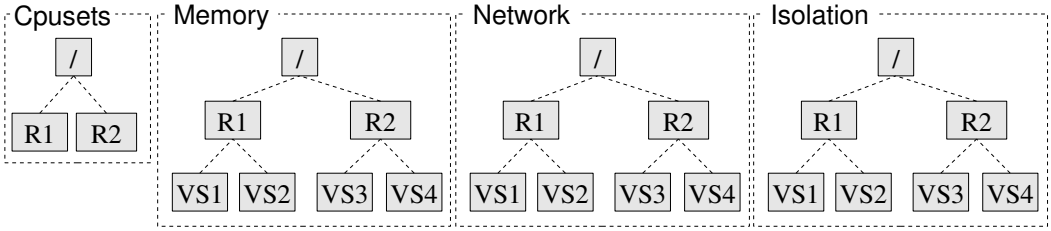
Some initial work on this project produced a system that simply used the same partition function for all subsystems, i.e. for every cpuset created there would also be a beancounter created, and all processes moved into the new cpuset would also become part of the new beancounter. However, very plausible scenarios were presented to demonstrate that this was too limiting.

A generic container framework should support some way of allowing different partitions for different subsystems. Since the requirements suggest that supporting hierarchical partitions is useful, even if not required/desired for all subsystems, we refer to these partitions, without loss of generality, as *hierarchies*.

For illustration, we consider the following examples of dividing processes on a system.




(a) University Server




(b) Hosting Server

Figure 1: Example container divisions with independent (single-subsystem) hierarchies

**A University Timesharing System**[2]

A university server has various users—students, professors, and system tasks. For CPU and memory, it is desired to partition the system according to the process owner's user ID, whereas for network traffic, it is desired to partition the system according to the traffic content (e.g. WWW browser related traffic across all users shares the same limit of 20%). Any single way of partitioning the system will make this kind of resource planning hard, potentially requiring creating a container for each tuple of the cross-product of the various configured resource subsystems.. Allowing the system to be partitioned in multiple ways, depending on resource type, is therefore a desirable feature.

**A Virtual Server System**

A large commercial hosting server has many NUMA nodes. Blocks of nodes are sold to resellers (R1, R2) to give guaranteed CPU/memory resources. The resellers then sell virtual servers (VS1–VS4) to end users, potentially overcommitting their resources but not affecting other resellers on the system.

The server owner would use cpusets to restrict each reseller to a set of NUMA memory nodes/CPUs; the reseller would then (via root-delegated capabilities) use a server isolation subsystem and a resource control subsystem to create virtual servers with various levels of resource guarantees/limits, within their own cpuset resources.

[2]Example contributed by Srivatsa Vaddagiri.

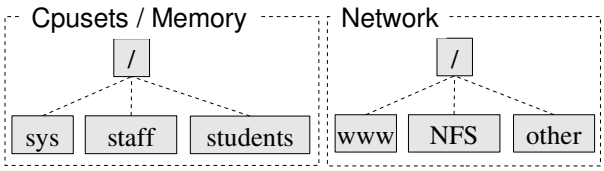
Two possible approaches to supporting these examples are given below; the illustrative figures represent a kernel with cpusets, memory, network, and isolation subsystems.

### 2.6.1 Independent hierarchies

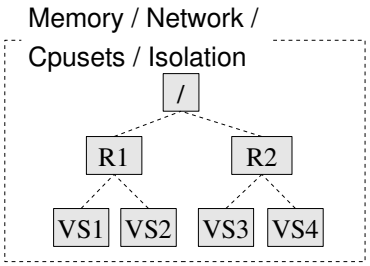
In the simplest approach, each hierarchy provides the partition for exactly one subsystem. So to use e.g. cpusets and beancounters on the same system, you would need to create a cpuset hierarchy and a beancounters hierarchy, and assign processes to containers separately for each subsystem.

This solution can be used to implement any of the other approaches presented below. The drawback is that it imposes a good deal of extra management work to userspace in the (we believe likely) common case when the partitioning is in fact the same across multiple or even all subsystems. In particular, moving processes between containers can have races—if userspace has to move a process as two separate actions on two different hierarchies, a child process forked during this operation might end up in inconsistent containers in the two hierarchies.

Figure 1 shows how containers on the university and hosting servers might be configured when each subsystem is an independent hierarchy. The cpusets and memory subsystems have duplicate configurations for the university server (which doesn't use the isolation subsystem), and the network, memory and isolation subsys-




(a) University Server




(b) Hosting Server

Figure 2: Example container divisions with multi-subsystem hierarchies

tems have duplicate configurations for the hosting server since they're the same for each virtual server.

#### 2.6.2 Multi-subsystem hierarchies

An extension to the above approach allows multiple subsystems to be bound on to the same hierarchy; so e.g. if you were using cpusets and beancounters, and you wanted the cpuset and beancounter assignments to be isomorphic for all processes, you could bind cpusets and beancounters to the same hierarchy of containers, and only have to operate on a single hierarchy when creating/destroying containers, or moving processes between containers.

Figure 2 shows how the support for multiple subsystems per hierarchy can simplify the configuration for the two example servers. The university server can merge the configurations for the cpusets and memory subsystems (although not for the network subsystem, since that's using an orthogonal division of processes). The hosting server can merge all four subsystems into a single hierarchy.

### 2.7 Non-process references to containers

Although the process is typically the object most universally associated with a container, it should also be possible to associate other objects—such as pages, file handles or network sockets—with a container, for accounting purposes or in order to affect the kernel's behaviour with respect to those objects.

Since such associations are likely to be subsystem-specific, the container framework needs primarily to be able to provide an efficient reference-counting mechanism, which will allow references to subsystem state objects to be made in such a way that they prevent the destruction of the associated container until the reference has been released.

### 2.8 Low overhead

The container framework should provide minimal additional runtime overhead over a system where individual subsystems are hard-coded into the source at all appropriate points (pointers in `task_struct`, additional `fork()` and `exit()` handlers, etc).

## 3 Existing/Related Work

In this section we consider the existing mechanisms for process tracking and control in Linux, looking at both those already included in the Linux source tree, and those proposed as bases for other efforts.

### 3.1 Unix process grouping mechanisms

Linux has inherited several concepts from classical Unix that can be used to provide some form of association be-

tween different processes. These include, in order of increasing specificity: group id (gid), user id (uid), session id (sid) and process group (pgrp).

Theoretically the gid and/or uid could be used as the identification portion of the container framework, and a mechanism for configuring per-uid/gid state could be added. However, these have the serious drawbacks that:

- Job-control systems may well want to run multiple jobs as the same user/group on the same machine.
- Virtual server systems will want to allow processes within a virtual server to have different uids/gids.

The sid and/or the pgrp might be suitable as a tracking base for containers, except for the fact that traditional Unix semantics allow processes to change their sid/pgrp (by becoming a session/group leader); removing this ability would be possible, but could resulting in unexpected breakage in applications. (In particular, requiring all processes in a virtual server to have the same sid or pgrp would probably be unmanageable).

### 3.2 Cpusets

The cpusets system is the only major container-like system in the mainline Linux kernel. Cpusets presents a pseudo-filesystem API to userspace with semantics including:

- Creating a directory creates a new empty cpuset (an analog of a container).
- Control files in a cpuset directory allow you to control the set of memory nodes and/or CPUs that tasks in that cpuset can use.
- A special control file, `tasks` can be read to list the set of processes in the cpuset; writing a pid to the `tasks` file moves a process into that cpuset.

### 3.3 Linux/Unix container systems

The Eclipse [3] and *Resource Containers* [4] projects both sought to add quality of service to Unix. Both supported hierarchical systems, allowing free migration of processes and threads between containers; Eclipse used the independent hierarchies model described in Section 2.6.1; Resource Containers bound all schedulers (subsystems) into a single hierarchy.

PAGG [5] is part of the SGI Comprehensive System Accounting project, adapted for Linux. It provides a generic container mechanism that tracks process membership and allows subsystems to be notified when processes are created or exit. A crucial difference between PAGG and the design presented in this paper is that PAGG allows a free-form association between processes and arbitrary containers. This results in more expensive access to container subsystem state, and more expensive `fork()`/`exit()` processing.

Resource Groups [2] (originally named CKRM – *Class-based Kernel Resource Management*) and BeanCounters [1] are resource control frameworks for Linux. Both provide multiple resource controllers and support additional controllers. ResGroups' support for additional controllers is more generic than the design proposed in this paper—we feel that the additional overheads that it introduces are unnecessary; BeanCounters is less generic, in that additional controllers have to be hard-coded into the existing BeanCounters source. Both frameworks also enforce a particular resource model on their resource controllers, which may be inappropriate for resource controllers with different requirements from those envisaged—for example, implementing cpusets with its current interface (or an equivalent natural interface) on top of either the BeanCounters or ResGroups abstractions would not be possible.

### 3.4 Linux virtual server systems

There have been a variety of virtual server systems developed for Linux; early commercial systems included Ensim's Virtual Private Server [6] and SWSoft's Virtuozzo [7]; these both provided various resource controllers along with namespace isolation, but no support for generic extension with user-provided subsystems.

More recent open-sourced virtualization systems have included VServer [8] and OpenVZ [9], a GPL'd subset of the functionality of Virtuozzo.

### 3.5 NSProxy-based approaches

Recent work on Linux virtual-server systems [10] has involved providing multiple copies of the various

namespaces (such as for IPC, process ids, etc) within the kernel, and having the namespace choice be made on a per-process basis. The fact that different processes can now have different IPC namespaces results in the requirement that these namespaces be reference-counted on `fork()` and released on `exit()`. To reduce the overhead of such reference counting, and to reduce the number of per-process pointers required for all these virtualizable namespaces, the `struct nsproxy` was introduced. This is a reference-counted structure holding reference-counted pointers to (theoretically) all the namespaces required for a task; therefore at `fork()`/`exit()` time only the reference count on the `nsproxy` object must be adjusted. Since in the common case a large number of processes are expected to share the same set of namespaces, this results in a reduction in the space required for namespace pointers and in the time required for reference counting, at the cost of an additional indirection each time one of the namespaces is accessed.

## 4 Proposed Design

This section presents a proposed design for a container framework based on the requirements presented in Section 2 and the existing work surveyed above. A prototype implementation of this design is available at the project website [11], and has been posted to the `linux-kernel@vger.kernel.org` mailing list and other relevant lists.

### 4.1 Overview

The proposed container approach is an extension of the process-tracking design used for cpusets. The current design favours the multiple-hierarchy approach described in Section 2.6.2.

### 4.2 New structure types

The container framework adds several new structures to the kernel. Figure 3 gives an overview of the relationship between these new structures and the existing `task_struct` and `dentry`.

#### 4.2.1 `container`

The `container` structure represents a container object as described in Section 1. It holds parent/child/sibling information, per-container state such as flags, and a set of subsystem state pointers, one for the state for each subsystem configured in the kernel. It holds no resource-specific state. It currently[3] holds no reference to a list of tasks in the container; the overhead of maintaining such a list would be paid whenever tasks `fork()` or `exit()`, and the relevant information can be reconstructed via a simple walk of the tasklist.

#### 4.2.2 `container_subsys`

The `container_subsys` structure represents a single resource controller or isolation component, e.g. a memory controller or a CPU scheduler.

The most important fields in a `container_subsys` are the callbacks provided to the container framework; these are called at the appropriate times to allow the subsystem to learn about or influence process events and container events in the hierarchy to which this subsystem is bound, and include:

**create** is called when a new container is created

**destroy** is called when a container is destroyed

**can_attach** is called to determine whether the subsystem wants to allow a process to be moved into a given container

**attach** is called when a process moves from one container to another

**fork** is called when a process forks a child

**exit** is called when a process exits

**populate** is called to populate the contents of a container directory with subsystem-specific control files

**bind** is called when a subsystem is moved between hierarchies.

[3] The possibility of maintaining such a task list just for those subsystems that really need it is being considered.

Apart from `create` and `destroy`, implemention of these callbacks is optional.

Other fields in `container_subsys` handle housekeeping state, and allow a subsystem to find out to which hierarchy it is attached.

Subsystem registration is done at compile time—subsystems add an entry in the header file `include/linux/container_subsys.h`. This is used in conjunction with pre-processor macros to statically allocate an identifier for each subsystem, and to let the container system locate the various `container_subsys` objects.

The compile-time registration of subsystems means that it is not possible to build a container subsystem purely as a module. Real-world subsystems are expected to require subsystem-specific hooks built into other locations in the kernel anyway; if necessary, space could be left in the relevant arrays for a compile-time configurable number of "extra" subsystems.

#### 4.2.3 `container_subsys_state`

A `container_subsys_state` represents the base type from which subsystem state objects are derived, and would typically be embedded as the first field in the subsystem-specific state object. It holds housekeeping information that needs to be shared between the generic container system and the subsystem. In the current design this state consists of:

**container** – a reference to the `container` object with which this state is associated. This is primarily useful for subsystems which want to be able to examine the tree of containers (e.g. a hierarchical resource manager may propagate resource information between subsystem state objects up or down the hierarchy of containers).

**refcnt** – the reference count of external non-process objects on this subsystem state object, as described in Section 2.7. The container framework will refuse to destroy a container whose subsystems have non-zero states, even if there are no processes left in the container.

To access its state for a given task, a subsystem can call `task_subsys_state(task, <subsys_id>)`. This function simply dereferences the given subsystem pointer in the task's `css_group` (see next Section).

#### 4.2.4 `css_group`

For the same reasons as described in Section 3.5, maintaining large numbers of pointers (per-hierarchy or per-subsystem) within the `task_struct` object would result in space wastage and reference-counting overheads, particularly in the case when container systems compiled into the kernel weren't actually used.

Therefore, this design includes the `css_group` (container subsystem group) object, which holds one `container_subsys_state` pointer for each registered subsystem. A reference-counted pointer field (called `containers`) to a `css_group` is added to `task_struct`, so the space overhead is one pointer per task, and the time overhead is one reference count operation per `fork()`/`exit()`. All tasks with the same set of container memberships across all hierarchies will share the same `css_group`.

A subsystem can access the per-subsystem state for a task by looking at the slot in the task's `css_group` indexed by its (statically defined) subsystem id. Thus the additional indirection is the only subsystem-state access overhead introduced by the `css_group`; there's no overhead due to the generic nature of the container framework. The space/time tradeoff is similar to that associated with `nsproxy`.

It has been proposed (by Srivatsa Vaddagiri and others) that the `css_group` should be merged with the `nsproxy` to form a single per-task object containing both namespace and container information. This would be a relatively straightforward change, but the current design keeps these as separate objects until more experience has been gained with the system.

### 4.3 Code changes in the core kernel

The bulk of the new code required for the container framework is that implementing the container filesystem and the various tracking operations. These are driven entirely by the user-space API as described in Section 4.5.

Changes in the generic kernel code are minimal and consist of:

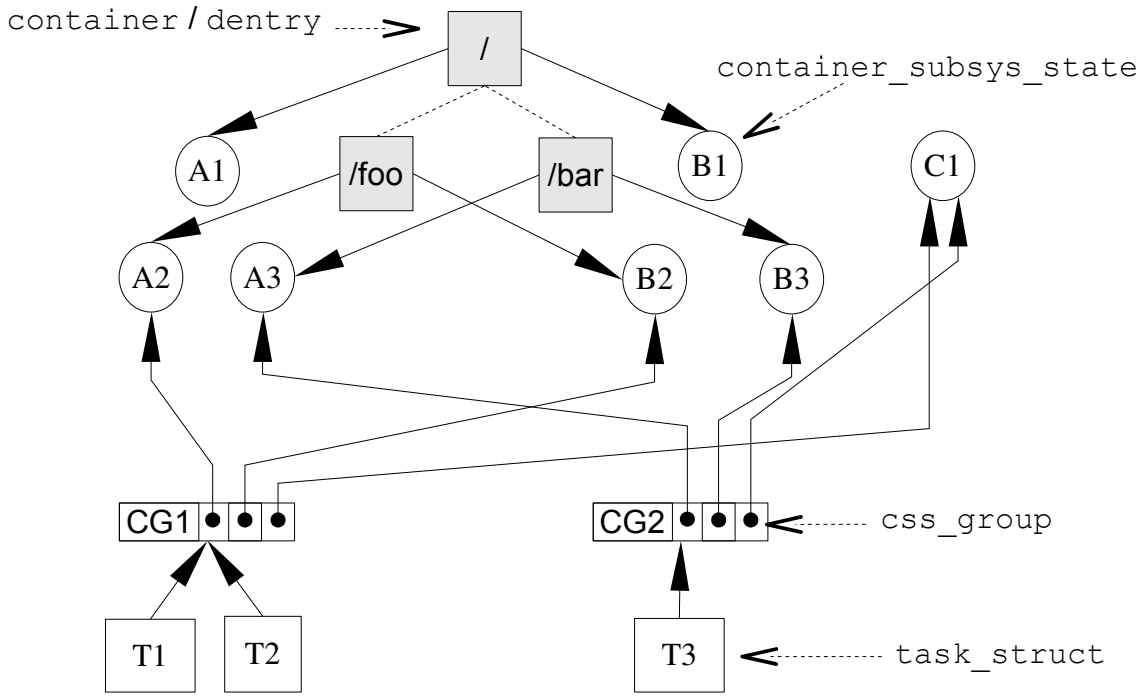



Figure 3: Three subsystems (A, B, C) have been compiled into the kernel; a single hierarchy has been mounted with A and B bound to it. Container directories `foo` and `bar` have been created, associated with subsystem states A2/B2 and A3/B3 respectively. Tasks T1 and T2 are in container `foo`; task T3 is in container `bar`. Subsystem C is unbound so all container groups (and hence tasks) share subsystem state C1.

- A hook early in the `fork()` path to take an additional reference count on the task's `css_group` object.
- A hook late in the `fork()` path to invoke subsystem-specific `fork` callbacks, if any are required.
- A hook in the `exit()` path to invoke any subsystem-specfic `exit` callbacks, if any, and to release the reference count on the task's `css_group` object (which will also free the `css_group` if this releases the last reference.)

### 4.4 Locking Model

The current container framework locking model revolves around a global mutex (`container_mutex`), each task's `alloc_lock` (accessed via `task_lock()` and `task_unlock()`) and RCU critical sections.

The `container_mutex` is used to synchronize per-container operations driven by the userspace API. These include creating/destroying containers, moving processes between containers, and reading/writing subsystem control files. Performance-sensitive operations should not require this lock.

When modifying the `containers` pointer in a task, the container framework surrounds this operation with `task_lock()`/`task_unlock()`, and follows with a `synchronize_rcu()` operation before releasing the `container_mutex`.

Therefore, in order for a subsystem to be sure that it is accessing a valid `containers` pointer, it suffices for *at least one* of the following three conditions to be true of the current task:

- it holds `container_mutex`
- it holds its own `alloc_lock`.
- it is in an RCU critical section.

The final condition, that of being in an RCU critical section, doesn't prevent the current task being concurrently moved to a different container in some hierarchy—it

simply tells you that the current task was in the specified set of containers at some point in the very recent past, and that any of the subsystem state pointers in that `css_group` object won't be candidates for deletion until after the end of the current RCU critical section.

When an object (such as a file or a page) is accounted to the value that has been read as the current subsystem state for a task, it may actually end up being accounted to a container that the task has just been moved from; provided that a reference to the charged subsystem state is stored somewhere in the object, so that at release time the correct subsystem state can be credited, this is typically sufficient. A subsystem that wants to reliably migrate resources between containers when the process that allocated those resources moves (e.g. cpusets, when the `memory_migrate` mode is enabled) may need additional subsystem-specific locking, or else use one of the two other locking methods listed above, in order to ensure that resources are always accounted to the correct containers.

The subsystem state pointers in a given `css_group` are immutable once the object has been created; therefore as long as you have a pointer to a valid `css_group`, it is safe to access the subsystem fields without additional locking (beyond that mandated by subsystem-specific rules).

### 4.5 Userspace API

The userspace API is very similar to that of the existing cpusets system.

A new hierarchy is created by mounting an instance of the `container` pseudo-filesystem. Options passed to the `mount()` system call indicate which of the available subsystems should be bound to the new hierarchy. Since each subsystem can only be bound to a single hierarchy at once, this will fail with `EBUSY` if the subsystem is already bound to a different hierarchy.

Initially, all processes are in the root container of this new hierarchy (independently of whatever containers they might be members of in other hierarchies).

If a mount request is made for a set of subsystems that exactly match an existing active hierarchy, the same superblock is reused.

At the time when a container hierarchy is unmounted, if the hierarchy had no child containers then the hierarchy is released and all subsystems are available for reuse. If the hierarchy still has child containers, the hierarchy (and superblock) remain active even though not actively attached to a mounted filesystem.[4]

Creating a directory in a container mount creates a new child container; containers may be arbitrarily nested, within any restrictions imposed by the subsystems bound to the hierarchy being manipulated.

Each container directory has a special control file, `tasks`. Reading from this file returns a list of processes in the container; writing a pid to this file moves the given process into the container (subject to successful `can_attach()` callbacks on the subsystems bound to the hierarchy).

Other control files in the container directory may be created by subsystems bound to that hierarchy; reads and writes on these files are passed through to the relevant subsystems for processing.

Removing a directory from a container mount destroys the container represented by the directory. If tasks remain within the container, or if any subsystem has a non-zero reference count on that container, the `rmdir()` operation will fail with `EBUSY`. (The existence of subsystem control files within a directory does not keep it busy; these are cleared up automatically.)

The file `/proc/PID/container` lists, for each active hierarchy, the path from the root container to the container of which process `PID` is a member.[5]

The file `/proc/containers` gives information about the current set of hierarchies and subsystems in use. This is primarily useful for debugging.

### 4.6 Overhead

The container code is optimized for fast access by subsystems to the state associated with a given task, and a fast `fork()`/`exit()` path.

Depending on the synchronization requirements of a particular subsystem, the first of these can be as simple as:

[4] They remain visible via a `/proc` reporting interface.

[5] An alternative proposal is for this to be a directory holding container path files, one for each subsystem.

```
struct task_struct *p = current;
rcu_read_lock()
struct state *st =
  task_subsys_state(p,
                    my_subsys_id);
...
<Do stuff with st>
...
rcu_read_unlock();
```

On most architectures, the RCU calls expand to no-ops, and the use of inline functions and compile-time defined subsystem ids results in the code being equivalent to `struct state *st = p->containers->subsys[my_subsys_id]`, or two constant-offset pointer dereferences. This involves one additional pointer dereference (on a presumably hot cacheline) compared to having the subsystem pointer embedded directly in the task structure, but has a reduced space overhead and reduced refcounting overhead at `fork()` / `exit()` time.

Assuming none of the registered subsystems have registered `fork()` / `exit()` callbacks, the overhead at `fork()` (or `exit()`) is simply a `kref_get()` (or `kref_put()`) on `current->containers->refcnt`.

## 5 Example Subsystems

The containers patches include various examples of subsystems written over the generic containers framework. These are primarily meant as demonstrations of the way that the framework can be used, rather than as fully-fledged resource controllers in their own right.

### 5.1 CPU Accounting Subsystem

The `cpuacct` subsystem is a simple demonstration of a useful container subsystem. It allows the user to easily read the total amount of CPU time (in milliseconds) used by processes in a given container, along with an estimate of the recent CPU load for that container.

The 250-line patch consists of:

- callback hooks added to the `account_*_time()` functions in `kernel/sched.c` to inform the subsystem when a particular process is being charged for a tick.
- declaration of a subsystem in `include/linux/container_subsys.h`
- Kconfig/Makefile additions
- the code in `kernel/cpuacct.c` to implement the subsystem. This can focus on the actual details of tracking the CPU for a container, since all the common operations are handled by the container framework.

Internally, the `cpuacct` subsystem uses a per-container spinlock to synchronize access to the usage/load counters.

### 5.2 Cpusets

Cpusets is already part of the mainline kernel; as part of the container patches it is adapted to use the generic container framework (the primary change involved removal of about 30% of the code, that was previously required for process-tracking).

Cpusets is an example of a fairly complex subsystem with hooks into substantial other parts of the kernel (particularly memory management and scheduler parameters). Some of its control files represent flags, some represent bitmasks of memory nodes and cpus, and others report usage values. The interface provided by the generic container framework is sufficiently flexible to accomodate the cpusets API.

Internally, cpusets uses an additional global mutex—`callback_mutex`—to synchronize container-driven operations (moving a task between containers, or updating the memory nodes for a container) with callbacks from the memory allocator or OOM killer.

For backwards compatibility the existing `cpuset` filesystem type remains; any attempt to mount it gets redirected to a mount of the `container` filesystem, with a subsystem option of `cpuset`.

### 5.3 ResGroups

ResGroups (formerly CKRM [2]) is a hierarchical resource control framework that specifies the resource limits for each child in terms of a fraction of the resources available to its parent. Additionally resources

may be borrowed from a parent if a child has reached its resource limits.

The abstraction provided by the generic containers framework is low-level, with free-form control files. As an example of how to provide multiple subsystems sharing a common higher-level resource abstraction, ResGroups is implemented as a container subsystem library by stripping out the group management aspects of the code and adding container subsystem callbacks. A resource controller can use the ResGroups abstraction simply by declaring a container subsystem, with the subsystem's `private` field pointing to a ResGroups `res_controller` structure with the relevant resource-related callbacks.

The ResGroups library registers control files for that subsystem, and translates the free-form read/write interface into a structured and typed set of callbacks. This has two advantages:

- it reduces the amount of parsing code required in a subsystem
- it allows multiple subsystems with similar (resource or other) abstractions to easily present the same interface to userspace, simplifying userspace code.

One of the ResGroups resource controllers (`numtasks`, for tracking and limiting the number of tasks created in a container) is included in the patch.

### 5.4 BeanCounters

BeanCounters [1] is a single-level resource accounting framework that aims to account and control consumption of kernel resources used by groups of processes. Its resource model allows the user to specify soft and hard limits on resource usage, and tracks high and low watermarks of resource usage, along with resource allocations that failed due to limits being hit.

The port to use the generic containers framework converts between the raw read/write interface and the structured get/store in a similar way to the ResGroups port, although presenting a different abstraction.

Additionally, BeanCounters allows the accounting context (known as a *beancounter*) to be overridden in particular situations, such as when in an interrupt handler or when performing work in the kernel on behalf of another process. This aspect of BeanCounters is maintained unchanged in the containers port—if a process has an override context set then that is used for accounting, else the context reached via the `css_group` pointer is used.

### 5.5 NSProxy / Container integration

A final patch in the series integrates the NSProxy system with the container system by making it possible to track processes that are sharing a given namespace, and (potentially in the future) create custom namespace sets for processes. This patch is a (somewhat speculative) example of a subsystem that provides an isolation system rather than a resource control system.

The namespace creation paths (via `fork()` or `unshare()`) are hooked to call `container_clone()`. This is a function provided by the container framework that creates a new sub-container of a task's container (in the hierarchy to which a specified subsystem is bound) and moves the task into the new container. The `ns` container subsystem also makes use of the `can_attach` container callback to prevent arbitrary manipulation of the process/container mappings.

When a task changes its namespaces via either of these two methods, it ends up in a fresh container; all of its children (that share the same set of namespaces) are in the same container. Thus the generic container framework provides a simple way to export to userspace the sets of namespaces in use, their hierarchical relationships, and the processes using each namespace set.

## 6 Conclusion and Future Work

In this paper we have examined some of the existing work on process partitioning/tracking in Linux and other operating systems, and enumerated the requirements for a generic framework for such tracking; a proposed design was presented, along with examples of its use.

As of early May 2007, several other groups have proposed resource controllers based on the containers framework; it is hoped that the containers patches can be trialled in Andrew Morton's `-mm` tree, with the aim of reaching the mainline kernel tree in time for 2.6.23.

## 7 Acknowledgements

Thanks go to Paul Jackson, Eric Biederman, Srivatsa Vaddagiri, Balbir Singh, Serge Hallyn, Sam Villain, Pavel Emelianov, Ethan Solomita, and Andrew Morton for their feedback and suggestions that have helped guide the development of these patches and this paper.

## References

[1] Kir Kolyshkin, *Resource management: the Beancounters*, Proceedings of the Linux Symposium, June 2007

[2] *Class-based Kernel Resource Management*, `http://ckrm.sourceforge.net`

[3] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz, *Resource management for QoS in Eclipse/BSD*. In Proceedings of FreeBSD'99 Conf., Berkeley, CA, USA, Oct. 1999

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. *Resource containers: A new facility for resource management in server systems*. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pages 45–58, 1999

[5] SGI. *Linux Process Aggregates (PAGG)*, `http://oss.sgi.com/projects/pagg/`

[6] Ensim Virtual Private Server, `http://www.ensim.com`

[7] SWSoft Virtuozzo, `http://www.swsoft.com/en/virtuozzo/`

[8] Linux VServer, `http://www.linux-vserver.org/`

[9] OpenVZ, `http://www.openvz.org/`

[10] Eric W. Biederman, *Multiple Instances of the Global Linux Namespaces*, Proceedings of the Linux Symposium, July 2006.

[11] Linux Generic Process Containers, `http://code.google.com/p/linuxcontainers`

# Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

## Conference Organizers

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

## Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*