An Abstract Model of Linux Control Groups and Subsystems

Glyn Normington Steve Powell

March 6, 2014

This document specifies precisely some of the features of *Linux control groups* and *subsystems* in order to support robust and reliable development of *Warden*, *Garden* and related products.

Contents

1 Introduction

In order to implement some features and improvements to *Warden* we sharpen our understanding of Linux control groups and simultaneously document their use by presenting an abstract model in Z[1].

Linux control groups are part of the basis for the current implementation of Warden[1][2] and Garden[2] as well as of Linux Containers (LXC), Docker, and Google's *lmctfy* ("let me contain that for you") project and so are important to the industry at large as well as to Cloud Foundry.

We intend to use our improved understanding to benefit:

- Linux Kernel documentation;
- Warden[1] and Garden[2] development and exploitation; and
- the wider development audience.

2 Overview of this document

We develop a model of a single *control group hierarchy* from the bottom up, in stages:

- Section 3 introduces some basic types and the notion of a named *control group* which consists of a collection of tasks. Basic operations for creating and destroying tasks are defined.
- Section 4 models a control group *hierarchy* as a collection of named control groups which partition the tasks in the system. Operations are defined for creating and deleting control groups and for moving a task between control groups.
- Section 5 introduces the notion of a *subsystem* which identifies collections of *resources* using control group names.
- Section 6 combines a *subsystem* with a *hierarchy*.
- Section 7 completes the model as a system of multiple subsystems associated with a shared hierarchy.

[1] See appendix A for a one-page summary of the Z notation.
[2] All references are listed in appendix D.

The further stage of modelling a system of multiple control group hierarchies is omitted since distinct hierarchies operate independently of each other.

The document concludes with section 8 which relates the model to kernel externals. Control group hierarchies are created using the `mount` command and monitored and managed using the mounted filesystem. The remaining control group hierarchy operations correspond to system actions, such as creating and destroying tasks.

There is a series of appendices which offer background material:

- Appendix A gives a one-page summary of most of the Z-notation used in this document.
- Appendix B outlines the sources we have used to help compile the model.
- Appendix C is a record of some experiments we performed to investigate control groups and subsystems properties.
- Appendix D contains references.

3 Fundamentals

We begin by describing some fundamental pieces of our model.

3.1 Given sets

There are fundamental sets of items we will not look into, for example $TASK$s[3]:

$$[TASK]$$

which appear in collections in each control group.

There are $RESOURCE$s, which can be allocated, managed and monitored:

$$[RESOURCE]$$

and which we will use to represent our abstract notion of a subsystem.

There are names, of which there are many types, here are two of them:

$$[SIMPLENAME, SSNAME]$$

the simple names out of which control group (path)names are formed, and the subsystem names.

[3] A *task* is the Linux kernel term for a process.

3.2 Sets of resources

A subsystem will be concerned with resources, and we will model that by a set associated with a control group. It is simpler to start with the resource sets by themselves.

There is a *ResourceSet* type:

$$ResourceSet == \mathbb{F}\, RESOURCE$$

a piece of state which holds one of them:

Resources

$$rs : ResourceSet$$

and these states begin by being empty:

$$InitResources \mathrel{\widehat=} [Resources' \mid rs' = \varnothing]$$

The basic operations not very interesting: simply adding and removing elements, which we expect happens in batches (sets):

AddResources

$$rs? : ResourceSet$$

$$\Delta Resources$$

$$rs' = rs \cup rs?$$

where we just stick 'em all in.

RemoveResources

$$rs? : ResourceSet$$

$$\Delta Resources$$

$$rs' = rs \setminus rs?$$

where we just take 'em all out.

These become more interesting when used in the context of a subsystem, for which see later.

3.3 A single control group

A single control group in isolation is just a (finite) collection of tasks, and we can document this:

$$\begin{array}{|l} \textit{CGroup} \\ \hline \textit{taskset} : \mathbb{F}\ \textit{TASK} \\ \hline \end{array}$$

As a convenience, we define the (global) function *tasks* which extracts the *taskset* from a *CGroup*, and the simple control group *EmptyCGroup* which has no tasks in it:

$$\textit{tasks} == (\lambda\ \textit{CGroup} \bullet \textit{taskset})$$

$$\textit{EmptyCGroup} == (\mu\ \textit{CGroup} \mid \textit{taskset} = \varnothing)$$

3.4 Control group path names

Control groups have names, but these have a structure—a file directory-like structure—so we define them as *sequences* of simple names (think of these as directory names), for which we use the shorthand *CGPath*:

$$\textit{CGPath} == \operatorname{seq} \textit{SIMPLENAME}$$

There is a special name for the first (initial) control group, the root of the structure. We define a constant for it:

$$\begin{array}{|l} \textit{ROOTCGROUPNAME} : \textit{CGPath} \\ \hline \textit{ROOTCGROUPNAME} = \langle\rangle \\ \hline \end{array}$$

There may be many *CGroup*s in the system, and a collection of *CGroup* with names must 'fill out' the tree-structure with *CGPath* names. We explain this with a *NamedCGroups* collection:

$$\begin{array}{|l} \textit{NamedCGroups} \\ \hline \textit{cg} : \textit{CGPath} \pfun \textit{CGroup} \\ \hline \textit{ROOTCGROUPNAME} \in \operatorname{dom} \textit{cg} \\ \forall\ s : \operatorname{dom} \textit{cg} \bullet \forall\ t : \textit{CGPath} \mid t \subset s \bullet t \in \operatorname{dom} \textit{cg} \\ \hline \end{array}$$

The constraints imply that there is at least one *CGroup* in the collection (the root) and that the names are 'prefix-closed'. The latter constraint is

equivalent to saying the names fit together in a tree-structure (as in a file-system). There is a *CGroup* for every node in the tree, 'named' by the sequence of simple names from the root to the node.

We have said nothing about the *tasks* in the *CGroup*s in the named collection. This comes next.

3.5 System tasks

In order to describe the relationship between named control groups and tasks, we need a minimal model of the tasks of the system. We first postulate a (fixed) initial task:

$$INITIALTASK : TASK$$

and then we say that the tasks of the system always include this initial task:

Tasks

$$systasks : \mathbb{F}\ TASK$$

$$INITIALTASK \in systasks$$

which has the (pleasing) consequence that there is always at least one task in the system.

3.5.1 Initialisation

When the system tasks are first created, then, this *INITIALTASK* must exist, which is to say that there is at least one task. We simplify it by saying there is exactly one (though we are not usually around to see this natal state).

InitTasks

$$Tasks'$$

$$\#systasks' = 1$$

We may not need to use this property directly.

3.5.2 Creating a new task—lowest level

We do need to speak about new task creation later, and so here is how we model it, from the point of view of *Tasks*:

CreateTask

$parent? : TASK$
$task! : TASK$

$\Delta Tasks$

$parent? \in systasks$
$task! \notin systasks$
$systasks' = systasks \cup \{task!\}$

The new task has to have a (designated) parent, which is just any existing system task; the new task is not already in the system and is inserted into the system, without disturbing any other tasks.

3.5.3 Destroying a task—lowest level

Task destruction is also needed later in the context of control groups, but from the point of view of *Tasks* we can say:

DestroyTask

$task? : TASK$

$\Delta Tasks$

$task? \in systasks$
$task? \neq INITIALTASK$
$systasks' = systasks \setminus \{task?\}$

The task being destroyed is already in the system; it must not be the system's initial task (this constraint could have been *derived* since $Tasks'$ already insists that *INITIALTASK* must remain in $systasks'$); the task is removed from the set of system tasks, without disturbing the other tasks.

Of course we know there is a parent/child relationship between tasks which the system knows and maintains. We don't think we need to know about that (except at create time) so we don't model it[4].

4 Control group hierarchies

Although it appears not possible to have a control group hierarchy without an attached subsystem (see section C.2) we find it easier to consider hierarchies

[4]If we need this it will become clear later when we cannot express a constraint, or cannot describe a change of state correctly.

in isolation first[5].

We combine the named control groups and system tasks and say how they should be related. At the same time we describe a function *cgroup* which maps each task to a control group in the collection.

We are defining a *hierarchy* of control groups:

CGHierarchy

$$
\begin{array}{l}
Tasks \\
NamedCGroups \\
\\
cgroup : TASK \pfun CGPath \\
\hline
tasks \circ cg \ \mathsf{partition} \ systasks \\
\\
\mathrm{dom}\ cgroup = systasks \\
\forall\, t : systasks \bullet t \in tasks(cg(cgroup\ t))
\end{array}
$$

Each task of the system is in precisely one control group, or, as expressed here, the control group tasks *partition* the system tasks; *and* there is a function (*cgroup*) which maps each system task to the (full) name (*CGPath*) of the control group in which it resides.

We note that the existence of the function *cgroup* is guaranteed by the partition requirement (on $tasks \circ cg$) that accompanies it. The function *cgroup* is 'derived state', and these assertions impose no extra constraints upon a hierarchy. In fact, the existence of *cgroup* is *equivalent* to the partition constraint.

It is legitimate to ask how a *CGHierarchy* is initialised. We describe this next along with other operations on a hierarchy.

4.1 Hierarchy operations

It is natural to consider state changes that might 'happen' to hierarchies. We will try to understand how a hierarchy is initialised, how new tasks enter or leave a hierarchy, and how tasks are moved between control groups in a hierarchy. Then we can consider how a new control group of a hierarchy is created or destroyed.

Describing each of these operations precisely challenges our state descriptions and constraints—which helps us to believe that they capture exactly what we want to say.

[5]This may reflect the behaviour of the re-architected *cgroups* function currently being developed.

4.1.1 Initialisation

We first consider initialisation of a hierarchy. This is a sort of degenerate operation, as is common in Z specifications.

InitCGHierarchy

$$
\begin{array}{l}
CGHierarchy' \\
systasks? : \mathbb{F}\ TASK \\
\hline
systasks' = systasks? \\
\exists\, CGroup' \bullet \\
(\ \ cg' = \{\, ROOTCGROUPNAME \mapsto \theta CGroup' \,\} \\
\land\ taskset' = systasks'\ )
\end{array}
$$

All existing system tasks are placed in the initial *CGroup*, with the special root name.

4.1.2 A single task operation skeleton

Since *Tasks* is part of our *CGHierarchy* state, it is natural to want to use our description of system task operations *CreateTask* and *DestroyTask* when describing the (extra) effects upon a hierarchy. The create and destroy operation on *Tasks* can be elaborated to explain what happens on the extended state. In general this is called "promotion" and has well-trodden precedents in Z. Our usage here is more of an extension.

We first say what all of these extensions have in common (expressed as a "promotion" schema), which in our case says that fundamental task operations only affect a *single task* in the hierarchy, while everything else remains fixed.

The *SingleTaskOp* explains what remains fixed when performing a system task operation in a hierarchy. It is a skeleton operation schema which we can adapt to particular cases.

SingleTaskOp

$$
\begin{array}{l}
task : TASK \\
\Delta CGHierarchy \\
\hline
systasks' \setminus \{task\} = systasks \setminus \{task\} \\
\{task\} \lhd cgroup' = \{task\} \lhd cgroup \\
\mathrm{dom}\ cg' = \mathrm{dom}\ cg
\end{array}
$$

This needs some explaining; the three clauses can be described thus:

Except possibly for *task*, the system tasks are unaffected (neither created nor destroyed). Except for this same *task*, the association between tasks and their control group path names (*cgroup*) is unaffected. The control group (directory) structure is likewise unaffected.

Note that, as we noted before, the map $cgroup'$, which is *derived state*, uniquely determines the partition ($tasks \circ cg$) except possibly for empty control groups. And, *vice versa*, the partition determines the *cgroup* map.

Simply saying what the new *cgroup* map should be and that no extra empty control groups arise (which is the force of $\mathrm{dom}\, cg = \mathrm{dom}\, cg'$) suffices to determine the rest of the new state, partition included.

With this skeleton[6] we can describe the next operations easily.

4.1.3 Create a new task in a hierarchy

Here we extend *CreateTask* as a *SingleTaskOp*eration on a hierarchy.

CreateTaskInCG

$$\begin{array}{l} CreateTask \\ SingleTaskOp[task!/task] \\ \hline cgroup'\ task! = cgroup\ parent? \end{array}$$

The particular task is renamed to be the *output* task (the one created). Everything else stays the same (see *SingleTaskOp*) and the new *task*! goes in the *parent*?'s control group.

4.1.4 Destroy an existing task in a hierarchy

Deleting a task from the system is even easier to explain:

DestroyTaskInCG

$$\begin{array}{l} DestroyTask \\ SingleTaskOp[task?/task] \end{array}$$

The particular task is renamed to be the *input* task (the one to be deleted). Everything else stays the same (see *SingleTaskOp*). No more needs to be

[6] We notice, for the first time, just how symmetric this is: reverse all the (state) operations (swapping all primed states with their un-primed states—for example $cgroup'$ and *cgroup* are exchanged) and we have the same promotion schema.

said, since after *DestroyTask* removes *task*? from *systasks*′, everything else is 'healed' by the constraints in the promotion.

We are heartened by this: *SingleTaskOp* captures exactly what we mean by a 'task-local' change, and is sufficient to precisely describe the effects on the *CGHierarchy* when such changes occur, excepting only when there is a hierarchy-specific piece of information to supply.

4.1.5 Move a task from one control group to another

Within a hierarchy we want to explain what happens if we move a task from one control group to another. The system tasks are not changed by this, and we need to preserve the partitioning nature of the hierarchy. We need to supply the task which we are moving (*task*?), and the (full name of the) control group we are moving it to (*dest*?). We do not need to supply the control group it 'moves from', because the hierarchy state has that information already.

This, it turns out, is another "single task operation", so we can use *SingleTaskOp* again to help to explain it.

MoveCGroupTask

$$
\begin{array}{l}
dest? : CGPath \\
\\
SingleTaskOp[task?/task] \\
\Xi Tasks \\
\hline
task? \in systasks \\
dest? \in \mathrm{dom}\ cg \\
\\
cgroup'\ task? = dest?
\end{array}
$$

The constraints on the inputs are that the *task*? exists already and the path (*dest*?) is the name of a known control group (of this hierarchy). The new map *cgroup*′ must map *task*? to the control group with path name *dest*?.

Amazingly, this is all we need to say. The last line implies that *task*? $\in$ dom *cgroup*′ and therefore *tasks*? $\in$ *systasks*′ too (from *CGHierarchy*′).

Ξ*Tasks* ensures that *task*? must also be in *systasks* before we start, so 'the task exists already' is implied by the last line, although we prefer to be explicit about it.

The *SingleTaskOp* constraints fix everything else for us.

It is instructive to consider what this operation specification implies if *dest*? = *cgroup task*?, or in other words, if we attempt to move a task to the

control group it is already in. It should be clear that nothing changes in this case.

4.1.6 Create a new *CGroup* in a hierarchy

How does a new control group arise (in a hierarchy)? Clearly we need to name it, and it must have an initial state, but there are also things to do with the hierarchy that have to be true, and we mustn't disturb the task structure of the system when creating a new control group, either:

CreateCGroup

$$path? : CGPath$$

$$\Delta CGHierarchy$$
$$\Xi Tasks$$

$$path? \notin \mathrm{dom}\, cg$$
$$path? \neq ROOTCGROUPNAME$$
$$cg' = cg \cup \{path? \mapsto EmptyCGroup\}$$
$$cgroup' = cgroup$$

The control group path name is not an existing path (it's not in the old state); it cannot be the root; the new path must point to the empty control group, and the path must 'fit' into the *cg* map (see below); the task partition is unaffected.

What does it mean for the path to 'fit'? It means that all its prefixes are in dom *cg* already. Equivalently, its immediate parent (the 'front' of the sequence) is there. The constraint which says '*path*? is not the root', is strictly redundant (it can be derived from the other constraints).

4.1.7 Delete a *CGroup* from a hierarchy

When a control group is deleted, what happens to its tasks or its children (in the hierarchy)? This is explained by saying that you can only destroy control groups that don't have tasks or children. Here is how this can be formalised:

DestroyCGroup

$$path? : CGPath$$

$$\Delta CGHierarchy$$
$$\Xi Tasks$$

$$path? \notin \operatorname{dom} cg'$$
$$path? \neq ROOTCGROUPNAME$$
$$cg = cg' \cup \{path? \mapsto EmptyCGroup\}$$
$$cgroup' = cgroup$$

The constraints say: this is not a new path (it's not in the new state); it cannot be the root; the path identifies the empty control group (in the old state); the task partition is unaffected.

Notice that the path *used to* 'fit' in cg, and if removing it leaves the cg' structured correctly, this means it cannot have had any children beforehand.

The constraint which says '$path?$ is not the root', is *also* redundant. It is harder to see why in this case, but removing the root would produce a $CGHierarchy'$ with an empty map cg'. The constraints on $CGHierarchy'$ (in particular from the $NamedCGroups'$ schema) prohibit this.

The constraints for *Create...* and *Destroy...* are deliberately symmetric. Doing it this way helped us to (write and) understand them.

5 Subsystems

A *Subsystem* is a collection of resource sets named by *CGPath* names. We will eventually marry this with a *GCHierarchy*, but we consider it in isolation first.

Subsystem

$$res : CGPath \twoheadrightarrow ResourceSet$$

$$pinned : \mathbb{F}\, CGPath$$

$$pinned = \operatorname{dom}(res \rhd \{\varnothing\})$$

Resource sets "named by control group path names" are represented by the map *res*; *pinned* is derived state (completely determined by *res*) and consists of the control group path names which refer to non-empty resource sets[7].

[7] $\operatorname{dom}(res \rhd \{\varnothing\})$ is a technical jewel, meaning the domain of the relation which is that subset of *res* consisting of pairs that do not relate names to the empty resource set.

5.1 Subsystem operations

We consider initialisation and basic resource update operations, which we lift from the *Resources* operations we had before.

5.1.1 Initialisation

We cannot say everything about the initial subsystem (until the *CGHierarchy* is available), but we might try to say:

$$pinned' = \varnothing$$

which doesn't tie down the domain of *res*, but at least says that there are no resources in its range.

This is a mistake. There is *no* guarantee that a subsystem initially uses *no* resources—in fact it is extremely likely that *some* resources will be used by the tasks of the system which, remember, appear in the root of a hierarchy when it is created. So we should not say anything about an *initial Subsystem*. For completeness we will define the name though:

$$InitSubsystem \mathrel{\widehat=} Subsystem'$$

letting it 'flap' completely (except for the usual subsystem state constraints, of course).

5.1.2 Lifting the resource set operations

We want to add and remove resources from a particular resource set. In general we can say how we lift an operation on resource sets to subsystems:

SubsystemResourcesOp

$$cgp? : CGPath$$

$$\Delta Subsystem$$
$$\Delta Resources$$

$$cgp? \in \operatorname{dom} res$$
$$\{cgp?\} \lhd res = \{cgp?\} \lhd res'$$
$$res\ cgp? = rs$$
$$res'\ cgp? = rs'$$

There is an input path name (*cgp*?) which must be a known name of the subsystem; the resource sets (other than the one named *cgp*?) are not changed; the resource set named *cgp*? is changed in precisely the way that the *Resources* operation changes its resource set (*rs* goes to *rs*′).

5.1.3 Add and remove resources

When we combine *SubsystemResourcesOp* with operations on *Resources* (which include the signature $\Delta Resources$) we get a lifted operation definition on *Subsystem* except for the $\Delta Resources$ signature, which we can then hide.

$$AddSSResources \mathrel{\widehat=} \exists \Delta Resources \bullet (AddResources \land SubsystemResourcesOp)$$

$$RemoveSSResources \mathrel{\widehat=} \exists \Delta Resources \bullet (RemoveResources \land SubsystemResourcesOp)$$

The $\exists \Delta Resources \bullet$ prefix is just a clumsy way of hiding the *Resources* and $Resources'$ states which we only needed to expose to explain the lifting, and should not be exposed on the subsystem operation.

Are these *really* operations on *Subsystem*?

We can demonstrate that they are by deriving the "signature" of, for example, *AddSSResources*, and it works out as[8]:

$$rs? : ResourceSet$$
$$cgp? : CGPath$$
$$\Delta Subsystem$$

which is clearly an operation with inputs *rs*?, a set of resources, and *cgp*?, a path name, as we would expect.

6 Attached subsystems

*Subsystem*s are pretty anæmic because they are not yet 'attached' to a hierarchy. This is what we describe next.

AttachedSubsystem

$$Subsystem$$
$$CGHierarchy$$
$$\hline$$
$$\mathrm{dom}\, res = \mathrm{dom}\, cg$$

The control group paths that the subsystem tracks are precisely the ones in the attached hierarchy.

[8] Not really, for we have preserved some constraints in the declarations to make it easier to relate to our other operations.

6.1 Attached subsystem operations

The operations on *CGHierarchy* and those on *Subsystem* can now be lifted to *AttachedSubsystem* by similar lifting mechanisms we used before. The constraints (aka preconditions) imposed by *CGHierarchy* and *Subsystem* operations combine via the *AttachedSubsystem* constraint, to enable us to derive many (but not all) of the preconditions of the lifted operations.

6.1.1 Initialisation

The initially attached subsystem ought to be a simple combination of initial subsystem and initial hierarchy; and it is:

$$\begin{aligned} &\mathit{InitAttachedSubsystem} \mathrel{\widehat=} \\ &\quad \mathit{AttachedSubsystem}' \land \mathit{InitSubsystem} \land \mathit{InitCGHierarchy} \end{aligned}$$

Notice that the resources used by the initial (root) control group might be anything—we had to allow this when explaining *InitSubsystem*.

6.1.2 Operations lifted from *CGHierarchy*

In *CGHierarchy* operations we created a new task, destroyed an existing task, moved a task and created and destroyed a control group. All of these occur when a subsystem is attached to the hierarchy, and we can explain the *extra* constraints easily enough.

In each case the hierarchy changes are already stipulated, we only have to say what happens to the subsystem that is tracking it.

Create a task: We have to explain what parts of the subsystem may change when a new task is created:

CreateTaskInAttachedCGH

$$\begin{aligned} &\Delta \mathit{AttachedSubsystem} \\ &\mathit{CreateTaskInCG} \\ &\hline \\ &\textbf{let } \mathit{cgp} == \mathit{cgroup}'\ \mathit{task}! \bullet \\ &\quad \{\mathit{cgp}\} \lhd \mathit{res} = \{\mathit{cgp}\} \lhd \mathit{res}' \land \\ &\quad \mathit{res}\ \mathit{cgp} \subseteq \mathit{res}'\ \mathit{cgp} \end{aligned}$$

Here we've introduced a local shorthand *cgp* for $\mathit{cgroup}'\ \mathit{task}!$, the name of the control group which contains the new task ($\mathit{task}!$); the resource-set map,

res, remains the same on all control groups except this one; on this one the resource set usage may increase (but it cannot decrease).

This is intentionally non-deterministic because we cannot say what resources a new task might require or what its parent might require. In general, then, we might add resources into the control group.

Note that it is possible for the *pinned* set to increase but only by the addition of the single control group named *cgp*.

Destroy a task: We have to explain what parts of the subsystem may change when a task is destroyed:

DestroyTaskInAttachedCGH

$$\Delta AttachedSubsystem$$
$$DestroyTaskInCG$$

$$\textbf{let } cgp == cgroup\ task? \bullet$$
$$\quad \{cgp\} \lhd res = \{cgp\} \lhd res' \land$$
$$\quad res'\ cgp \subseteq res\ cgp$$

Shorthand again; the resource-set map, *res*, remains the same on all control groups except this one; on this one the resource set usage may *decrease* (but it cannot increase).

Here the *pinned* set might reduce by one control group, but only one.

Move a task: We have to explain what parts of the subsystem may change when a task is moved from one control group to another.

MoveTaskInAttachedCGH

$$\Delta AttachedSubsystem$$
$$MoveCGroupTask$$

$$\textbf{let } src == cgroup\ task? \bullet$$
$$\quad \{src, dest?\} \lhd res = \{src, dest?\} \lhd res' \land$$
$$\quad res'\ src \subseteq res\ src \land$$
$$\quad res\ dest? \subseteq res'\ dest?$$

This time we introduce local shorthand for the source control group (*src*); the *res* map remains unchanged on all control groups except possibly for the source and destination control groups; the source control group resources might decrease; the destination control group resources might increase.

We are tempted to say that the resources transfer from one control group to the other; but we cannot say this. In general the transfer may involve extra resources, or it may reduce the resources used. Particular subsystems might be able to say more about this.

Again it is instructive to consider what happens if the source and destination are the same control group. The specification implies that this is a no-op: the resources used are unchanging. It remains to be seen if this is the case, but we would be surprised if it wasn't. It is nice to see this falling out of a general description.

Create a control group: We have to explain what parts of the subsystem may change when a new control group is created:

$$
\begin{array}{|l}
\textit{CreateCGroupInAttachedCGH} \\
\Delta \textit{AttachedSubsystem} \\
\textit{CreateCGroup} \\
\hline
res' \; path? = \varnothing \\
res = \{path?\} \lhd res' \\
\hline
\end{array}
$$

The new control group uses no resources (and has no tasks you may recall from *CreateCGroup*); the resources for all the other control groups remain the same. Notice how the Δ*AttachedSubsystem* constraints save us from having to say stuff like 'no other control groups magically appear'.

Destroy a control group: We have to explain what parts of the subsystem may change when a control group is destroyed:

$$
\begin{array}{|l}
\textit{DestroyCGroupInAttachedCGH} \\
\Delta \textit{AttachedSubsystem} \\
\textit{DestroyCGroup} \\
\hline
res \; path? = \varnothing \\
res' = \{path?\} \lhd res \\
\hline
\end{array}
$$

The control group being destroyed *must not be using any resources*; the resources for all the other control groups remain the same.

Not using any resources tallies with the *CGHierarchy* operation constraint that it must have no tasks when it is destroyed, although not having tasks doesn't imply it is not using resources).

We also get the parent-child restrictions for free (ultimately from *NamedCGroups*).

6.1.3 Operations lifted from *Subsystem*

Adding or removing resources from a particular control group has no effect on the structure or the tasks in a hierarchy.

We can therefore lift the *Add* and *Remove* resources operations directly as follows:

$$AddResourcesInAttachedSubsystem \mathrel{\widehat=} AddSSResources \land \Delta AttachedSubsystem \land \Xi CGHierarchy$$

$$RemoveResourcesInAttachedSubsystem \mathrel{\widehat=} RemoveSSResources \land \Delta AttachedSubsystem \land \Xi CGHierarchy$$

where $\Xi CGHierarchy$ documents that the hierarchy remains unchanged.

7 Many subsystems

The aim of this section is to move away from considering just one subsystem and its affect upon a control group hierarchy, to the more general case where two or more subsystems are attached to the same hierarchy.

We model this as a collection of *AttachedSubsystem*s, where the hierarchies are identical. A function which extracts just the control group hierarchy from an *AttachedSubsystem* is then useful:

$$cgHierarchy == (\lambda\, AttachedSubsystem \bullet \theta CGHierarchy)$$

and we can use this function to ensure that the hierarchies spoken of in the *AttachedSubsystem* collection are all the same.

Multiple subsystems attached to the same hierarchy consists of a set of subsystem names, a map from this set to *AttachedSubsystem*s, and the control group hierarchy which is common to them all.

MultiSubsystems

$$subNames : \mathbb{F}_1\, SSNAME$$
$$subs : SSNAME \twoheadrightarrow AttachedSubsystem$$
$$commonCGH : CGHierarchy$$

$$subNames = \operatorname{dom} subs$$
$$\forall\, sn : subNames \bullet cgHierarchy(subs\ sn) = commonCGH$$

The domain of the map is exactly the set of subsystem names, and every named subsystem's control group hierarchy is identical to the top-level one.

There are two points to be made: the set of subsystem names (*subNames*) is *non-empty*, ensuring at least one *AttachedSubsystem* exists; and the common hierarchy is really only there to tie them together, but we will see advantages to giving it a separate, subsystem-independent name later.

7.1 Many subsystems operations

Previously we have used Δ to indicate the before and after states of a schema, but in this case we want to impose an additional constraint on all of our operations. After a *MultiSubsystem* is created we want to insist that the set of subsystems remains fixed. This reflects reality where the act of `mount`ing the subsystems creates the hierarchy at the same time, and there is no API for partially *un*mounting a subsystem, or adding one to an existing mount point.

We will define what is allowed to change for every *MultiSubsystems* operation:

$$
\begin{array}{|l}
\hline
\textit{MultiSubsystemsChange} \\
\Delta \textit{MultiSubsystems} \\
\hline
\textit{subNames}' = \textit{subNames} \\
\hline
\end{array}
$$

Every operation leaves the named subsystems unchanged.

7.1.1 Initialisation

Since the *subNames* cannot change, when are they initialised? When the first *MultiSubsystems* state is created:

$$
\begin{array}{|l}
\hline
\textit{InitMultiSubsystems} \\
\textit{subNames}? : \mathbb{F}_1\ \textit{SSNAME} \\
\textit{systasks}? : \mathbb{F}\ \textit{TASK} \\
\textit{MultiSubsystems}' \\
\hline
\textit{subNames}' = \textit{subNames}? \\
\forall\, \textit{sn} : \textit{subNames}? \bullet \\
\quad \exists\, \textit{AttachedSubsystem}' \mid \textit{subs}'\ \textit{sn} = \theta \textit{AttachedSubsystem}' \bullet \\
\quad\quad \textit{InitAttachedSubsystem} \\
\hline
\end{array}
$$

We need to supply the subsystems names (at least one of them), and the system tasks currently extant, and the attached subsystems are each initialised appropriately. The *MultiSubsystems*$'$ constraints imply that the hierarchies match.

7.1.2 Lifting attached subsystem operations

The operations to be lifted are: create, destroy and move a task; create and destroy a control group; and add and remove resources, which should act on an identified subsystem.

These groups of operations are lifted in two different ways, reflecting the fact that four of them affect *all* the subsystems, and two of them (the resource ones) affect only *one* subsystem.

The first way is where all the subsystems change together. We describe what one among many changes might look like:

OneSubsystemChange

$$
\begin{array}{l}
sn : SSNAME \\
MultiSubsystemsChange \\
\Delta AttachedSubsystem \\
\hline
subs\ sn = \theta AttachedSubsystem \\
subs'\ sn = \theta AttachedSubsystem'
\end{array}
$$

crucially, we do not insist here that all the other subsystems remain fixed (which is what a 'point operation' promotion would do).

The second way is when we wish to insist that *only* one subsystem is changed (normally the subsystem name is input).

We augment *OneSubsystemChange* to say this:

OnlySubsystemChange

$$
\begin{array}{l}
OneSubsystemChange[sn?/sn] \\
\hline
\{sn?\} \lhd subs = \{sn?\} \lhd subs'
\end{array}
$$

which adds the further restriction that all the other subsystems remain the same in this case.

We use these two types of 'lifting' for our lifted operations.

Create a task: When a new task is created all attached subsystems change simultaneously. We expose the output *task*! and the input *parent*? as usual:

$$
\begin{array}{|l}
\textit{CreateTaskInMultiSs} \\
\hline
\textit{MultiSubsystemsChange} \\
\textit{parent}?, \textit{task}! : \textit{TASK} \\
\hline
\forall\, \textit{sn} : \textit{subNames} \bullet \\
\quad \exists\, \Delta \textit{AttachedSubsystem} \mid \textit{OneSubsystemChange} \bullet \\
\qquad \textit{CreateTaskInAttachedCGH} \\
\hline
\end{array}
$$

Each attached subsystem in *subs* changes as described in *CreateTaskInAttachedCGH*.

Notice that all the inputs and outputs to the individual operations are identical and that the hierarchies (all being identical) can only change in the same way—however, the resources may change in different ways for the different subsystems as permitted by *CreateTaskInAttachedCGH*.

Destroy a task: When a task is destroyed all attached subsystems change simultaneously. We expose the input *task*? as usual:

$$
\begin{array}{|l}
\textit{DestroyTaskInMultiSs} \\
\hline
\textit{MultiSubsystemsChange} \\
\textit{task}? : \textit{TASK} \\
\hline
\forall\, \textit{sn} : \textit{subNames} \bullet \\
\quad \exists\, \Delta \textit{AttachedSubsystem} \mid \textit{OneSubsystemChange} \bullet \\
\qquad \textit{DestroyTaskInAttachedCGH} \\
\hline
\end{array}
$$

Each attached subsystem changes as described in *DestroyTaskInAttachedCGH*.

Here the resources are depleted in each subsystem, but the resources involved in each subsystem need not be the same.

Move a task: Moving a task to a control group operates in a similar way. We expose the inputs dest? and task? as before:

$$
\begin{array}{|l}
\textit{MoveTaskInMultiSs} \\
\hline
\textit{MultiSubsystemsChange} \\
\textit{dest}? : \textit{CGPath} \\
\textit{task}? : \textit{TASK} \\
\hline
\forall\, \textit{sn} : \textit{subNames} \bullet \\
\quad \exists\, \Delta \textit{AttachedSubsystem} \mid \textit{OneSubsystemChange} \bullet \\
\qquad \textit{MoveTaskInAttachedCGH} \\
\hline
\end{array}
$$

Each attached subsystem manages the move in the same way, but may involve quite different resources in each case.

Create a control group: A new control group is created in each attached subsystem in the same way:

$$
\begin{array}{|l}
\textit{CreateCGroupInMultiSs} \\
\textit{MultiSubsystemsChange} \\
\textit{path?} : \textit{CGPath} \\
\hline
\forall\, \textit{sn} : \textit{subNames} \bullet \\
\quad \exists\, \Delta \textit{AttachedSubsystem} \mid \textit{OneSubsystemChange} \bullet \\
\quad\quad \textit{CreateCGroupInAttachedCGH} \\
\hline
\end{array}
$$

Each attached subsystem creates the new control group (*path*?) with no resources.

Destroy a control group: A control group is removed from each attached subsystem:

$$
\begin{array}{|l}
\textit{DestroyCGroupInMultiSs} \\
\textit{MultiSubsystemsChange} \\
\textit{path?} : \textit{CGPath} \\
\hline
\forall\, \textit{sn} : \textit{subNames} \bullet \\
\quad \exists\, \Delta \textit{AttachedSubsystem} \mid \textit{OneSubsystemChange} \bullet \\
\quad\quad \textit{DestroyCGroupInAttachedCGH} \\
\hline
\end{array}
$$

Each attached subsystem destroys a control group in the same way.

Notice that *every* subsystem must satisfy the constraints for this operation to apply. Any resource for this control group *in any subsystem* will prevent this.

Add and remove resources: Adding and removing resources applies to *only* one subsystem.

$$\textit{AddResourcesInMultiSs} \mathrel{\widehat=} \exists\, \Delta \textit{AttachedSubsystem} \bullet \textit{AddResourcesInAttachedSubsystem} \land \textit{OnlySubsystemChange}$$

$$\textit{RemoveResourcesInMultiSs} \mathrel{\widehat=} \exists\, \Delta \textit{AttachedSubsystem} \bullet \textit{RemoveResourcesInAttachedSubsystem} \land \textit{OnlySubsystemChange}$$

Here we use standard pointwise lifting.

The subsystem name input (*sn*?) appears in *OnlySubsystemChange* and the other inputs appear in the ...*ResourcesInAttachedSubsystem* operations.

8 Relating the model to `cgroups` function

Each operation on the *MultiSubsystems* state is related to a kernel command or action.

8.1 *InitMultiSubsystems*

The signature of the initialisation in the model is:

$$systasks? : \mathbb{F}\ TASK$$
$$subNames? : \mathbb{F}\ SSNAME$$
$$MultiSubsystems'$$

The `mount` command creates a *MultiSubsystems* state, that is, it attaches named subsystems to a (newly created) hierarchy.

```
mount -t cgroup <mountName> -o<subNames> <h-path>
```

The `<mountName>` is ignored (except perhaps for messages), the option (following `-o`) is a non-empty list[9] of subsystem names (*subNames*?), and `<h-path>` is a location in a filesystem where the state of the multiple subsystems and the hierarchy are surfaced as a virtual filesystem (see further API).

The input *systasks*? is the set of extant tasks (processes) in the system when the subsystems and hierarchy are created and is an implicit parameter to `mount`.

Directories in the control group filesystem at `<h-path>` correspond to control groups, and each subsystem contributes (pseudo) files to those directories.

Notice that if any filesystem is mounted on an existing mount point (`<h-path>` above) then the first `mount` is overwritten. A consequence of this is that subsystems may not be detached or attached to hierarchies that are already mounted.

8.2 *CreateTaskInMultiSs*

The signature of the task creation operation is:

[9] If the option is omitted, the list defaults to all the subsystems defined to the kernel.

$$
\begin{array}{l}
parent? : TASK \\
task! : TASK \\
\\
\Delta MultiSubsystems
\end{array}
$$

A task created by any process automatically causes this operation to occur for all attached subsystems.

The implicit input *parent*? is the new task's parent task, and the output *task*! is the task which is created.

As the subsystem state changes so the filesystem rooted in `<h-path>` reflects these updates for the relevant control group.

8.3 *DestroyTaskInMultiSs*

Similarly, the signature of the task destruction operation is:

$$
\begin{array}{l}
task? : TASK \\
\\
\Delta MultiSubsystems
\end{array}
$$

This operation occurs when a task dies.

The input *task*? is the dying task.

As the subsystem state changes so the filesystem rooted in `<h-path>` reflects these updates for the relevant control group.

8.4 *MoveTaskInMultiSs*

Moving a task from one control group to another has this signature:

$$
\begin{array}{l}
task? : TASK \\
dest? : CGPath \\
\\
\Delta MultiSubsystems
\end{array}
$$

This is an operation explicitly initiated by a control group action, for example when the process id (`pid`) referring to the task (*task*?) is written into a (pseudo) file in the `cgroup` filesystem (rooted at `<h-path>`), in a directory corresponding to the destination `cgroup` (path *dest*?).

For example:

```
echo 1234 >/h-path/dest/cg1/tasks
```

will move the task with process id `1234` from whatever control group it is currently in to the control group `dest/cg1`.

Again, as the subsystem state changes so the filesystem rooted in `<h-path>` reflects these updates; in this case this might involve two control groups.

Whether or not *resources* move with the task is determined by subsystem settings, and is not prohibited nor mandated by our model operation.

8.5 *CreateCGroupInMultiSs*

Creating a new control group in a hierarchy has this signature:

$$path? : CGPath$$
$$\Delta MultiSubsystems$$

A control group is created when a new directory is created (for example with `mkdir`) in the `cgroup` filesystem.

The input *path?* is the relative path of the directory from the `<h-path>` root.

Each subsystem automatically contributes (pseudo) files to the new directory. There are initially *no* tasks associated with the new control group.

8.6 *DestroyCGroupInMultiSs*

Destroying a control group has the same signature as creating one:

$$path? : CGPath$$
$$\Delta MultiSubsystems$$

A control group is deleted when its directory is removed (with `rmdir`, or equivalent) from the `cgroup` filesystem.

Before this will succeed (as the model describes), there must be no tasks and no resources associated with that control group, and there must be no subdirectories in the control group directory.

A consequence of these restrictions means that the root control group can never be deleted by this operation.

8.7 *Add* and *Remove ResourcesInMultiSs*

The model signatures for adding and removing resources managed by a single subsystem are the same:

$$
\begin{array}{|l}
rs? : ResourceSet \\
cgp? : CGPath \\
sn? : SSNAME \\
\\
\Delta MultiSubsystems \\
\hline
\end{array}
$$

These are the set of resources to be added or removed ($rs?$), the control group where these resource changes are to take place ($cgp?$), and the (name of the) subsystem which is managing these resources ($sn?$).

These operations describe only the most basic features of subsystems' resource management. They are limited to one subsystem's resources in one control group.

They are instigated by a subsystem as a result of some system state change rather than by an explicit command or action by a process.

A Z Notation

Numbers:

$\mathbb{N}$ `Natural numbers {0,1,...}`

Propositional logic and the schema calculus:

$\ldots \land \ldots$	And	$\langle\!\langle \ldots \rangle\!\rangle$	Free type injection
$\ldots \lor \ldots$	Or	$[\ldots]$	Given sets
$\ldots \Rightarrow \ldots$	Implies	$', ?, !, {}_0 \ldots {}_9$	Schema decorations
$\forall .. \mid .. \bullet ..$	For all	$\ldots \vdash \ldots$	theorem
$\exists .. \mid .. \bullet ..$	There exists	$\theta \ldots$	Binding formation
$\ldots \setminus \ldots$	Hiding	$\lambda \ldots$	Function definition
$\ldots \mathrel{\widehat=} \ldots$	Schema definition	$\mu \ldots$	Mu-expression
$\ldots == \ldots$	Abbreviation	$\Delta \ldots$	State change
$\ldots ::= \ldots \mid \ldots$	Free type definition	$\Xi \ldots$	Invariant state change

Sets and sequences:

$\{\ldots\}$	Set	$\ldots \setminus \ldots$	Set difference
$\{.. \mid .. \bullet ..\}$	Set comprehension	$\bigcup \ldots$	Distributed union
$\mathbb{P} \ldots$	Set of subsets of	$\# \ldots$	Cardinality
$\varnothing$	Empty set	$\ldots \subseteq \ldots$	Subset
$\ldots \times \ldots$	Cartesian product	$\ldots \subset \ldots$	Proper subset
$\ldots \in \ldots$	Set membership	$\ldots$ partition $\ldots$	Set partition
$\ldots \notin \ldots$	Set non-membership	seq	Sequences
$\ldots \cup \ldots$	Union	$\langle \ldots \rangle$	Sequence
$\ldots \cap \ldots$	Intersection	disjoint $\ldots$	Disjoint sequence of sets

Functions and relations:

$\ldots \leftrightarrow \ldots$	Relation	$\ldots^*$	Reflexive-transitive closure
$\ldots \nrightarrow \ldots$	Partial function		
$\ldots \rightarrow \ldots$	Total function	$\ldots (\!\mid \ldots \mid\!)$	Relational image
$\ldots \rightarrowtail \ldots$	Partial injection	$\ldots \oplus \ldots$	Functional overriding
$\ldots \rightarrowtail \ldots$	Injection	$\ldots \lhd \ldots$	Domain restriction
$\mathrm{dom} \ldots$	Domain	$\ldots \rhd \ldots$	Range restriction
$\mathrm{ran} \ldots$	Range	$\ldots \lhd \ldots$	Domain subtraction
$\ldots \mapsto \ldots$	maplet	$\ldots \rhd \ldots$	Range subtraction
$\ldots^\sim$	Relational inverse		

Axiomatic descriptions:

Declarations
Predicates

Schema definitions:

SchemaName

Declaration
Predicates

B Sources

As input to this work we accessed the following sources of information:

Kernel specs descriptions of control groups in the Kernel literature[4];

Kernel code the Linux Kernel[5];

experiments tests run against the `cgroups` kernel code (see appendix C);

RedHat article which provides "invariants" of the filesystem[8];

Linux kernel mailing list for informative and timely Q and A[7].

B.1 The Linux specification[4]

Here is the first section of the Linux `cgroups` specification[4]:

```
1.1 What are cgroups ?
----------------------

Control Groups provide a mechanism for aggregating/partitioning sets
of tasks, and all their future children, into hierarchical groups with
specialized behaviour.

Definitions:

A *cgroup* associates a set of tasks with a set of parameters for one
or more subsystems.

A *subsystem* is a module that makes use of the task grouping
facilities provided by cgroups to treat groups of tasks in
particular ways. A subsystem is typically a "resource controller" that
schedules a resource or applies per-cgroup limits, but it may be
anything that wants to act on a group of processes, e.g. a
virtualization subsystem.

A *hierarchy* is a set of cgroups arranged in a tree, such that
every task in the system is in exactly one of the cgroups in the
hierarchy, and a set of subsystems; each subsystem has system-specific
state attached to each cgroup in the hierarchy.  Each hierarchy has
an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task
cgroups. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy cgroups by name in an
instance of the cgroup virtual file system, specify and query to
which cgroup a task is assigned, and list the task PIDs assigned to
a cgroup. Those creations and assignments only affect the hierarchy
associated with that instance of the cgroup file system.

On their own, the only use for cgroups is for simple job
tracking. The intention is that other subsystems hook into the
generic cgroup support to provide new attributes for cgroups, such
```

```
as accounting/limiting the resources which processes in a cgroup can
access. For example, cpusets (see Documentation/cgroups/cpusets.txt)
allow you to associate a set of CPUs and a set of memory nodes with
the tasks in each cgroup.
```

Although this reads reasonably well (and by comparison to its peers it is a pretty good document) it commits some specification sins[10] (see Meyer[9]). For example, it is ambiguous concerning the number of and relationship between hierarchies; it doesn't explain what the initial state of a control group is; it talks about tasks and processes as though they were the same without comment (we think they are the same); the rules about the relationship between tasks and control groups sound contradictory; and so on.

C Notes and experiments

We gather here some observations, experiments and discussions which have informed our research.

C.1 Mailing lists discussions

It does not seem possible to define a `cgroup` hierarchy with no subsystems attached to it. See the mailing list discussion[7].

C.2 Experiments

We performed some simple experiments on a (virtual, and hence infinitely refreshable) Linux machine.

C.2.1 Multiple subsystems

Attaching multiple subsystems to a single hierarchy:

```
# cd /tmp/warden/cgroup
# mkdir test
# mount -t cgroup -o 'cpuset,blkio' none /tmp/warden/cgroup/test
# cd test
# mkdir parent
# cd parent
# echo 0 > cpuset.mems
# echo 0 > cpuset.cpus
# echo $$ > tasks
# cat tasks
```

[10] We don't mean this as an insult: without documents like these we couldn't do our work; it is no worse—and in many ways better—than many such.

```
1840
2014
# cat /proc/1840/cgroup
13:blkio,cpuset:/parent
4:memory:/parent/child
3:devices:/
2:cpuacct:/
1:cpu:/
# cat /proc/cgroups
#subsys_name hierarchy num_cgroups enabled
cpuset 13 2 1
cpu 1 1 1
cpuacct 2 1 1
memory 4 3 1
devices 3 1 1
freezer 0 1 1
blkio 13 2 1
perf_event 0 1 1
# cat /proc/mounts
...
none /tmp/warden/cgroup tmpfs rw,relatime 0 0
none /tmp/warden/cgroup/cpu cgroup rw,relatime,cpu 0 0
none /tmp/warden/cgroup/cpuacct cgroup rw,relatime,cpuacct 0 0
none /tmp/warden/cgroup/devices cgroup rw,relatime,devices 0 0
none /tmp/warden/cgroup/memory cgroup rw,relatime,memory 0 0
none /tmp/warden/cgroup/test cgroup rw,relatime,blkio,cpuset 0 0
```

C.2.2 Multiple hierarchies

Attaching a single subsystem to multiple hierarchies:

```
$ pwd
/home/vagrant
$ mkdir mem1
$ mkdir mem2
$ sudo su
# mount -t cgroup -o memory none /home/vagrant/mem1
# mount -t cgroup -o memory none /home/vagrant/mem2
# cd mem1
# mkdir inst1
# ls inst1
cgroup.clone_children  memory.failcnt ...
# ls ../mem2
cgroup.clone_children  inst1 memory.limit_in_bytes ...
# cd inst1
# echo 1000000 > memory.limit_in_bytes
# cat memory.limit_in_bytes
1003520
# cat ../../mem2/inst1/memory.limit_in_bytes
1003520
# echo $$ > tasks
# cat tasks
1365
1409
# cat ../../mem2/inst1/tasks
1365
1411
```

C.2.3 Overlapping hierarchies

Attaching a new subsystem to one of the hierarchies attached to an existing subsystem (this follows on from the previous experiment):

```
# mount -t cgroup -o cpuset none /home/vagrant/mem1
# ls /home/vagrant/mem1
cpuset.cpus ... // No memory.* files!
# ls /home/vagrant/mem2
cgroup.clone_children  inst1 memory.limit_in_bytes ...
```

D References

[1] Various authors, *Warden github repository*, `https://github.com/cloudfoundry/warden`.

[2] Various authors, *Garden github repository*, `https://github.com/pivotal-cf-experimental/garden`.

[3] Various authors *libcgroup*, `http://libcg.sourceforge.net/html/index.html`.

[4] Paul Menage, Paul Jackson and Christoph Lameter, *CGROUPS*, `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2004-2006.

[5] Linus Torvalds, *et al*, *Linux kernel source tree*, `https://github.com/torvalds/linux`.

[6] Various authors, *Memory Resource Controller*, `https://www.kernel.org/doc/Documentation/cgroups/memory.txt`.

[7] Kamezawa Hiroyuki, *et al*, *NOOP cgroup subsystem*, `http://thread.gmane.org/gmane.linux.kernel/777763`.

[8] Martin Prpič, Rüdiger Landmann, and Douglas Silas, *Red Hat Enterprise Linux 6.5 GA: Resource Management Guide*, `https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Resource_Management_Guide/Red_Hat_Enterprise_Linux-6-Resource_Management_Guide-en-US.pdf`.

[9] Bertrand Meyer, *On Formalism In Specifications*, IEEE Software, Vol. 2(1), 1985.