freedesktop.org

www/ Software/ systemd/ ControlGroupInterface

Edit Page History Repo Info

The New Control Group Interfaces

aka "I want to make use of kernel cgroups, how do I do this in the new world order?"

Starting with version 205 systemd provides a number of interfaces that may be used to create and manage labelled groups of processes for the purpose of monitoring and controlling them and their resource usage. This is built on top of the Linux kernel Control Groups ("cgroups") facility. Previously, the kernel's cgroups API was exposed directly as shared application API, following the rules of the Pax Control Groups document. However, the kernel cgroup interface has been reworked into an API that requires that each individual cgroup is managed by a single writer only. With this change the main cgroup tree becomes private property of that userspace component and is no longer a shared resource. On systemd systems PID 1 takes this role and hence needs to provide APIs for clients to take benefit of the control groups functionality of the kernel. Note that services running on systemd systems may manage their own subtrees of the cgroups tree, as long as they explicitly turn on delegation mode for them (see below).

That means explicitly, that:

1. The root control group may only be written to by systemd (PID 1). Services that create and manipulate control groups in the top level cgroup are in direct conflict with the kernel's requirement that each control group should have a single-writer only.
2. Services must set Delegate=yes for the units they intend to manage subcgroups of. If they create and manipulate cgroups outside of units that have Delegate=yes set, they violate the access contract for control groups.

For a more high-level background story, please have a look at this Linux Foundation News Story.

Why this all again?

- Objects placed in the same level of the cgroup tree frequently need to propagate properties from one to each other. For example, when using the "cpu" controller for one object then all objects on the same level need to do the same, otherwise the entire cgroup of the first object will be scheduled against the individual processes of the others, thus giving the first object a drastic malus on scheduling if it uses many processes.

- Similar, some properties also require propagation up the tree.

- The tree needs to be refreshed/built in scheduled steps as devices show up/go away as controllers like "blkio" or "devices" refer to devices via major/minor device node indexes, which are not fixed but determined only as a device appears.

- The tree also needs refreshing/rebuilding as new services are installed/started/instantiated/stopped/uninstalled.

- Many of the cgroup attributes are too low-level as API. For example, the major/minor device interface in order to be useful requires a userspace component for translating stable device paths into major/minor at the right time.

- By unifying the cgroup logic under a single arbiter it is possible to write tools that can manage all objects the system contains, including services, virtual machines containers and whatever else applications register.

- By unifying the cgroup logic under a single arbiter a good default that encompasses all kinds of objects may be shipped, thus making manual configuration unnecessary to take benefit of basic resource control.

systemd through its "unit" concept already implements a dependency network between objects where propagation can take place and contains a powerful execution queue. Also, a major part of the objects resources need to be controlled for are already systemd objects, most prominently the services systemd manages.

Why is this not managed by a component independent of systemd?

Well, as mentioned above, a dependency network between objects, usable for propagation, combined with a powerful execution engine is basically what systemd *is*. Since cgroups management requires precisely this it is an obvious choice to simply implement this in systemd itself.

Implementing a similar propagation/dependency network with execution scheduler outside of systemd in an independent "cgroup" daemon would basically mean reimplementing systemd a second time. Also, accessing such an external service from PID 1 for managing other services would result in cyclic dependencies between PID 1 which would need this functionality to manage the cgroup service which would only be available however after that service finished starting up. Such cyclic dependencies can certainly be worked around, but make such a design complex.

I don't use systemd, what does this mean for me?

Nothing. This page is about systemd's cgroups APIs. If you don't use systemd then the kernel cgroup rework will probably affect you eventually, but a different component will be the single writer userspace daemon managing the cgroup tree, with different APIs. Note that the APIs described here expose a lot of systemd-specific concepts and hence are unlikely to be available outside of systemd systems.

I want to write cgroup code that should work on both systemd systems and others (such as Ubuntu), what should I do?

On systemd systems use the systemd APIs as described below. At this time we are not aware of any component that would take the cgroup managing role on Upstart/sysvinit systems, so we cannot help you with this. Sorry.

What's the timeframe of this? Do I need to care now?

In the short-term future writing directly to the control group tree from applications should still be OK, as long as the Pax Control Groups document is followed. In the medium-term future it will still be supported to alter/read individual attributes of cgroups directly, but no longer to create/delete cgroups without using the systemd API. In the longer-term future altering/reading attributes will also be unavailable to userspace applications, unless done via systemd's APIs (either D-Bus based IPC APIs or shared library APIs for *passive* operations).

It is recommended to use the new systemd APIs described below in any case. Note that the kernel cgroup interface is currently being reworked (available when the "sane_behaviour" kernel option is used). This will change the cgroupfs interface. By using systemd's APIs this change is abstracted away and invisible to applications.

systemd's Resource Control Concepts

Systemd provides three unit types that are useful for the purpose of resource control:

- *Services* encapsulate a number of processes that are started and stopped by systemd based on configuration. Services are named in the style of `quux.service`.
- *Scopes* encapsulate a number of processes that are started and stopped by arbitrary processes via fork(), and then registered at runtime with PID1. Scopes are named in the style of `wuff.scope`.
- *Slices* may be used to group a number of services and scopes together in a hierarchial tree. Slices do not contain processes themselves, but the services and slices contained in them do. Slices are named in the style of `foobar-waldo.slice`, where the path to the location of the slice in the tree is encoded in the name with "-" as separator for the path components (`foobar-waldo.slice` is hence a subslice of `foobar.slice`). There's one special slices defined, `-.slice`, which is the root slice of all slices (`foobar.slice` is hence subslice of `-.slice`). This is similar how in regular file paths, "/" denotes the root directory.

Service, scope and slice units directly map to objects in the cgroup tree. When these units are activated they each map to directly (modulo some character escaping) to cgroup paths built from the unit names. For example, a service `quux.service` in a slice `foobar-waldo.slice` is found in the cgroup `foobar.slice/foobar-waldo.slice/quux.service/`.

Services, scopes and slices may be created freely by the administrator or dynamically by programs. However by default the OS defines a number of built-in services that are necessary to start-up the system. Also, there are four slices defined by default: first of all the root slice `-.slice` (as mentioned above), but also `system.slice`, `machine.slice`, `user.slice`. By default all system services are placed in the first slice, all virtual machines and containers in the second, and user sessions in the third. However, this is just a default, and the administrator my freely define new slices and assign services and scopes to them. Also note that all login sessions automatically are placed in an individual scope unit, as are VM and container processes. Finally, all users logging in will also get an implicit slice of their own where all the session scopes are placed.

Here's an example how the cgroup tree could look like (as generated with `systemd-cgls(1)`, see below):

```
├─user.slice
│ └─user-1000.slice
│   ├─session-18.scope
│   │ ├─703 login -- lennart
│   │ └─773 -bash
│   ├─session-1.scope
│   │ ├─  518 gdm-session-worker [pam/gdm-autologin]
│   │ ├─  540 gnome-session
│   │ ├─  552 dbus-launch --sh-syntax --exit-with-session
│   │ ├─  553 /bin/dbus-daemon --fork --print-pid 4 --print-address 6 --session
│   │ ├─  589 /usr/libexec/gvfsd
│   │ ├─  593 /usr/libexec//gvfsd-fuse -f /run/user/1000/gvfs
│   │ ├─  598 /usr/libexec/gnome-settings-daemon
│   │ ├─  617 /usr/bin/gnome-keyring-daemon --start --components=gpg
│   │ ├─  630 /usr/bin/pulseaudio --start
│   │ ├─  726 /usr/bin/gnome-shell
│   │ ├─  728 syndaemon -i 1.0 -t -K -R
│   │ ├─  736 /usr/libexec/gsd-printer
│   │ ├─  742 /usr/libexec/dconf-service
│   │ ├─  798 /usr/libexec/mission-control-5
│   │ ├─  802 /usr/libexec/goa-daemon
│   │ ├─  823 /usr/libexec/gvfsd-metadata
│   │ ├─  866 /usr/libexec/gvfs-udisks2-volume-monitor
│   │ ├─  880 /usr/libexec/gvfs-gphoto2-volume-monitor
│   │ ├─  886 /usr/libexec/gvfs-afc-volume-monitor
│   │ ├─  891 /usr/libexec/gvfs-mtp-volume-monitor
│   │ ├─  895 /usr/libexec/gvfs-goa-volume-monitor
│   │ ├─  999 /usr/libexec/telepathy-logger
│   │ ├─ 1076 /usr/libexec/gnome-terminal-server
│   │ ├─ 1079 gnome-pty-helper
│   │ ├─ 1080 bash
│   │ ├─ 1134 ssh-agent
│   │ ├─ 1137 gedit l
│   │ ├─ 1160 gpg-agent --daemon --write-env-file
│   │ ├─ 1371 /usr/lib64/firefox/firefox
│   │ ├─ 1729 systemd-cgls
│   │ ├─ 1929 bash
│   │ ├─ 2057 emacs src/login/org.freedesktop.login1.policy.in
│   │ ├─ 2060 /usr/libexec/gconfd-2
│   │ ├─29634 /usr/libexec/gvfsd-http --spawner :1.5 /org/gtk/gvfs/exec_spaw/0
│   │ └─31416 bash
│   └─user@1000.service
│     ├─532 /usr/lib/systemd/systemd --user
│     └─541 (sd-pam)
└─system.slice
  ├─1 /usr/lib/systemd/systemd --system --deserialize 22
  ├─sshd.service
  │ └─29701 /usr/sbin/sshd -D
  ├─udisks2.service
  │ └─743 /usr/lib/udisks2/udisksd --no-debug
  ├─colord.service
  │ └─727 /usr/libexec/colord
  ├─upower.service
  │ └─633 /usr/libexec/upowerd
  ├─wpa_supplicant.service
  │ └─488 /usr/sbin/wpa_supplicant -u -f /var/log/wpa_supplicant.log -c /etc/wpa_
  ├─bluetooth.service
  │ └─463 /usr/sbin/bluetoothd -n
  ├─polkit.service
  │ └─443 /usr/lib/polkit-1/polkitd --no-debug
  ├─alsa-state.service
  │ └─408 /usr/sbin/alsactl -s -n 19 -c -E ALSA_CONFIG_PATH=/etc/alsa/alsactl.con
  ├─systemd-udevd.service
  │ └─253 /usr/lib/systemd/systemd-udevd
  ├─systemd-journald.service
  │ └─240 /usr/lib/systemd/systemd-journald
  ├─rtkit-daemon.service
  │ └─419 /usr/libexec/rtkit-daemon
  ├─rpcbind.service
  │ └─475 /sbin/rpcbind -w
  ├─cups.service
  │ └─731 /usr/sbin/cupsd -f
  ├─avahi-daemon.service
  │ ├─417 avahi-daemon: running [delta.local]
  │ └─424 avahi-daemon: chroot helper
  ├─dbus.service
  │ ├─418 /bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --sys
  │ └─462 /usr/sbin/modem-manager
  ├─accounts-daemon.service
  │ └─416 /usr/libexec/accounts-daemon
  ├─systemd-ask-password-wall.service
  │ └─434 /usr/bin/systemd-tty-ask-password-agent --wall
  ├─systemd-logind.service
  │ └─415 /usr/lib/systemd/systemd-logind
  ├─ntpd.service
  │ └─429 /usr/sbin/ntpd -u ntp:ntp -g
  ├─rngd.service
  │ └─412 /sbin/rngd -f
  ├─libvirtd.service
  │ └─467 /usr/sbin/libvirtd
  ├─irqbalance.service
  │ └─411 /usr/sbin/irqbalance --foreground
  ├─crond.service
  │ └─421 /usr/sbin/crond -n
  ├─NetworkManager.service
  │ ├─ 410 /usr/sbin/NetworkManager --no-daemon
  │ ├─1066 /sbin/dhclient -d -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/
  │ └─1070 /sbin/dhclient -d -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/
  └─gdm.service
    ├─420 /usr/sbin/gdm
    ├─449 /usr/libexec/gdm-simple-slave --display-id /org/gnome/DisplayManager/Di
    └─476 /usr/bin/Xorg :0 -background none -verbose -auth /run/gdm/auth-for-gdm-
```

As you can see, services and scopes contain process and are placed in slices, and slices do not contain processes of their own. Also note that the special "-.slice" is not shown as it is implicitly identified with the root of the entire tree.

Resource limits may be set on services, scopes and slices the same way. All active service, scope and slice units may easily be viewed with the "systemctl" command. The hierarchy of services and scopes in the slice tree may be viewed with the "systemd-cgls" command.

Service and slice units may be configured via unit files on disk, or alternatively be created dynamically at runtime via API calls to PID 1. Scope units may only be created at runtime via API calls to PID 1, but not from unit files on disk. Units that are created dynamically at runtime via API calls are called *transient* units. Transient units exist only during runtime and are released automatically as soon as they finished/got deactivated or the system is rebooted.

If a service/slice is configured via unit files on disk the resource controls may be configured with the settings documented in systemd.resource-control(5). While the unit are started they may be reconfigured for services/slices/scopes (with changes applying instantly) with the a command line such as:

```
# systemctl set-property httpd.service CPUShares=500 MemoryLimit=500M
```

This will make these changes persistently, so that after the next reboot they are automatically applied right when the services are first started. By passing the `--runtime` switch the changes can alternatively be made in a volatile way so that they are lost on the next reboot.

Note that the number of cgroup attributes currently exposed as unit properties is limited. This will be extended later on, as their kernel interfaces are cleaned up. For example cpuset or freezer are currently not exposed at all due to the broken inheritance semantics of the kernel logic. Also, migrating units to a different slice at runtime is not supported (i.e. altering the Slice= property for running units) as the kernel currently lacks atomic cgroup subtree moves.

(Note that the resource control settings are actually also available on mount, swap and socket units. This is because they may also involve processes run for them. However, normally it should not be necessary to alter resource control settings on these unit types.)

The APIs

Most relevant APIs are exposed via D-Bus, however some *passive* interfaces are available as shared library, bypassing IPC so that they are much cheaper to call.

Creating and Starting

To create and start a transient (scope, service or slice) unit in the cgroup tree use the `StartTransientUnit()` method on the `Manager` object exposed by systemd's PID 1 on the bus, see the Bus API Documentation for details. This call takes four arguments. The first argument is the full unit name you want this unit to be known under. This unit name is the handle to the unit, and is shown in the "systemctl" output and elsewhere. This name must be unique during runtime of the unit. You should generate a descriptive name for this that is useful for the administrator to make sense of it. The second parameter is the mode, and should usually be `replace` or `fail`. The third parameter contains an array of initial properties to set for the unit. It is an array of pairs of property names as string and values as variant. Note that this is an array and not a dictionary! This is that way in order to match the properties array of the `SetProperties()` call (see below). The fourth parameter is currently not used and should be passed as empty array. This call will first create the transient unit and then immediately queue a start job for it. This call returns an object path to a `Job` object for the start job of this unit.

Properties

The properties array of `StartTransientUnit()` may take many of the settings that may also be configured in unit files. Not all parameters are currently accepted though, but we plan to cover more properties with future release. Currently you may set the `Description`, `Slice` and all dependency types of units, as well as `RemainAfterExit`, `ExecStart` for service units, `TimeoutStopUSec` and `PIDs` for scope units, and `CPUAccounting`, `CPUShares`, `BlockIOAccounting`, `BlockIOWeight`, `BlockIOReadBandwidth`, `BlockIOWriteBandwidth`, `BlockIODeviceWeight`, `MemoryAccounting`, `MemoryLimit`, `DevicePolicy`, `DeviceAllow` for services/scopes/slices. These fields map directly to their counterparts in unit files and as normal D-Bus object properties. The exception here is the `PIDs` field of scope units which is used for construction of the scope only and specifies the initial PIDs to add to the scope object.

To alter resource control properties at runtime use the `SetUnitProperty()` call on the `Manager` object or `SetProperty()` on the individual Unit objects. This also takes an array of properties to set, in the same format as `StartTransientUnit()` takes. Note again that this is not a dictionary, and allows properties to be set multiple times with a single invocation. THis is useful for array properties: if a property is assigned the empty array it will be reset to the empty array itself, however if it is assigned a non-empty array then this array is appended to the previous array. This mimics behaviour of array settings in unit files. Note that most settings may only be set during creation of units with `StartTransientUnit()`, and may not be altered later on. The exception here are the resource control settings, more specifically `CPUAccounting`, `CPUShares`, `BlockIOAccounting`, `BlockIOWeight`, `BlockIOReadBandwidth`, `BlockIOWriteBandwidth`, `BlockIODeviceWeight`, `MemoryAccounting`, `MemoryLimit`, `DevicePolicy`, `DeviceAllow` for services/scopes/slices. Note that the standard D-Bus `org.freedesktop.DBus.Properties.Set()` call is currently not supported by any of the unit objects to set these properties, but might eventually (note however, that it is substantially less useful as it only allows setting a single property at a time, resulting in races).

The `systemctl set-property` command internally is little more than a wrapper around `SetUnitProperty()`. The `systemd-run` tool is a wrapper around `StartTransientUnit()`. It may be used to either run a process as a transient service in the background, where it is invoked from PID 1, or alternatively as a scope unit in the foreground, where it is run from the `systemd-run` process itself.

Enumeration

To acquire a list of currently running units, use the `ListUnits()` call on the Manager bus object. To determine the scope/service unit and slice unit a process is running in use `sd_pid_get_unit()` and `sd_pid_get_slice()`. These two calls are implemented in `libsystemd-login.so`. These call bypass the system bus (which they can because they are passive and do not require privileges) and are hence very effecient to invoke.

VM and Container Managers

Use these APIs to register any kind of process workload with systemd to be placed in a resource controlled cgroup. Note however that for containers and virtual machines it is better to use the `machined` interfaces since they provide integration with "ps" and similar tools beyond what mere cgroup registration provides. Also see Writing VM and Container Managers for details.

Reading Accounting Information

Note that there's currently no systemd API to retrieve accounting information from cgroups. For now, if you need to retrieve this information use `/proc/$PID/cgroup` to determine the cgroup path for your process in the `cpuacct` controller (or whichever controller matters to you), and then read the attributes directly from the cgroup tree.

If you want to collect the exit status and other runtime parameters of your transient scope or service unit after the processes in them ended set the `RemainAfterExited` boolean property when creating it. This will has the effect that the unit will stay around even after all processes in it died, in the `SubState="exited"` state. Simply watch for state changes until this state is reached, then read the status details from the various properties you need, and finally terminate the unit via `StopUnit()` on the `Manager` object or `Stop()` on the `Unit` object itself.

Becoming a Controller

Optionally, it is possible for a program that registers a scope unit (the "scope manager") for one or more of its child processes to hook into the shutdown logic of the scope unit. Normally, if this is not done, and the scope needs to be shut down (regardless if during normal operation when the user invokes `systemctl stop` -- or something equivalent -- on the scope unit, or during system shutdown), then systemd will simply send SIGTERM to its processes. After a timeout this will be followed by SIGKILL unless the scope processes exited by then. If a scope manager program wants to be involved in the shutdown logic of its scopes it may set the `Controller` property of the scope unit when creating it via `StartTransientUnit()`. It should be set to the bus name (either unique name or well-known name) of the scope manager program. If this is done then instead of SIGTERM to the scope processes systemd will send the RequestStop() bus signal to the specified name. If the name is gone by then it will automatically fallback to SIGTERM, in order to make this robust. As before in either case this will be followed by SIGKILL to the scope unit processes after a timeout.

Scope units implement a special `Abandon()` method call. This method call is useful for informing the system manager that the scope unit is no longer managed by any scope manager process. Among other things it is useful for manager daemons which terminate but want to leave the scopes they started running. When a scope is abandoned its state will be set to "abandoned" which is shown in the usual systemctl output, as information to the user. Also, if a controller has been set for the scope, it will be unset. Note that there is not strictly need to ever invoke the `Abandon()` method call, however it is recommended for cases like the ones explained above.

Delegation

Service and scope units know a special `Delegate` boolean property. If set, then the processes inside the scope or service may control their own control group subtree (that means: create subcgroups directly via /sys/fs/cgroup). The effect of the property is that:

1. All controllers systemd knows are enabled for that scope/service, if the scope/service runs privileged code.
2. Access to the cgroup directory of the scope/service is permitted, and files/and directories are updated to get write access for the user specified in `User=` if the scope/unit runs unprivileged. Note that in this case access to any controllers is not available.
3. systemd will refrain from moving processes across the "delegation" boundary.

Generally, the `Delegate` property is only useful for services that need to manage their own cgroup subtrees, such as container managers. After creating a unit with this property set, they should use `/proc/$PID/cgroup` to figure out the cgroup subtree path they may manage (the one from the name=systemd hierarchy!). Managers should refrain from making any changes to the cgroup tree outside of the subtrees for units they created with the `Delegate` flag turned on.

Note that scope units created by `machined`'s `CreateMachine()` call have this flag set.

Example

Please see the systemd-run sources for a relatively simple example how to create scope or service units transiently and pass properties to them.

Last edited Fri May 7 01:22:37 2021