INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

cgroups v2 Support for benchexec

Robin Gloster

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Dr. Philipp Wendler
Submission date:	2022-01-17

Abstract

The Linux kernel developers have reimplemented the *cgroup* feature due to shortcomings in its original implementation and the most common Linux distributions have switched to version two (v2). BenchExec is a tool by the University of Munich that uses cgroup and namespacing to reliably benchmark processes and effectively limit their resource usage. It currently only supports cgroup version one (v1) and in order to ensure its compatibility with modern systems, needs to be updated to also support v2. The author implemented cgroup v2 support for BenchExec. This thesis explains the implementation decisions taken during the upgrade and evaluates the changes to show that v2 support is as effective as v1 support.

Contents

1 Introduction

Control Group (*cgroup*) is a Linux kernel feature to organise and group processes hierarchically and is used to monitor and limit resources of those processes. This is utilised extensively in BenchExec, a tool developed by the University of Munich aiming to reliably benchmark processes. It is employed in competitions such as the *International Competition on Software Verification* and is available open source as a general purpose benchmarking tool.[1] cgroup lets BenchExec collect exact metrics of resources used by a group of processes that are benchmarked, as well as to limit available resources. It also ensures that the processes can reliably be terminated. Therefore BenchExec is a state-of-the-art tool for benchmarking, with no viable alternatives using the Linux operating system to achieve this functionality. [1]

Due to shortcomings and inconsistencies in the implementation of cgroup, a version 2 has been developed and most major Linux distributions have switched the default to the new version. BenchExec so far does not support this. To further ensure compatibility with modern operating systems, BenchExec needs to implement cgroup v2 support. The aim of this thesis is to add such support, detailing implementation decisions and their justifications.

1.1 Overview

This thesis consists of six further chapters. The following, Chapter 2, explains necessary knowledge of cgroup and Chapter 3 background on BenchExec and its usage of cgroups. Chapter 4 details the requirements of the implementation and the actual changes undertaken, which is then evaluated in Chapter 5. This is followed by comparison and examination of related work in Chapter 6 and finally Chapter 7, with a look at the outcome and an outlook for possible future improvements.

[1] `https://github.com/sosy-lab/benchexec`

2 *cgroup* Fundamentals

The Linux kernel includes a mechanism called *Control Group*, often abbreviated to *cgroup*, to group processes together and organise these groups in a hierarchy. Among these groups system resources can be distributed and metrics on their usage are provided. Development started in 2006 at Google [2], then called *Process Containers*, later renamed to cgroup to avoid confusion with Linux containers. The initial release was in Linux 2.6.24. Due to issues in the first implementation, Linux 4.5 included the public release of cgroup v2 [3].

In conjunction with namespaces they are used to create the basis for containers, making LXC, SYSTEMD-NSPAWN, DOCKER, KUBERNETES and other similar systems possible [4].

Interaction with cgroup is done through a virtual file system called *cgroupfs*. `/sys/fs/cgroups` has become a de facto default path to mount it on, because *systemd*—the most common init system for Linux—uses that path. A new control group is created as a subdirectory in the file system tree and processes can then be moved to it. Initially all processes are added to the root of the cgroup hierarchy and a child process of a `fork()` inherits the parent's cgroup membership.

A large advantage of using cgroup, in comparison to interacting with a number of processes individually or by other means, is that the interaction can happen atomically and race-free.

2.1 *cgroup* Features

2.1.1 Subsystems

Different subsystems, also called controllers, make up the *cgroup* system and are configured or return information on the resources they control in files in the file system hierarchy. All interaction of the user with the controllers are regular Unix file operations.

blkio/io

The **blkio** subsystem in v1 and its successor **io** in v2 can limit read and write operations, and bandwidth on block I/O devices, and give information on those.

cpuset

With the **cpuset** controller, processes can be pinned to specific CPU cores and, in case of a CPU architecture with *non-uniform memory access* (NUMA), to memory nodes.

cpu & cpuacct

The **cpuacct** controller provides information on CPU usage of the processes in the cgroup, the **cpu** controller on the other hand, regulates distribution of CPU cycles between cgroups. In cgroup v2 they are merged to a single **cpu** controller.

memory

The **memory** controller makes it possible to set limits on memory and swap. It also gives information on memory and swap usage and provides a mechanism to recognise out-of-memory (OOM) situations and react to them without necessarily having the kernel OOM killer invoked.

pid

Introduced in Linux 4.3 the **pid** controller gives information on the current number of processes and can limit the maximum number of processes in the *cgroup*. This subsytem can protect the system from *forkbombs*, where processes create an exponentially growing number of further processes, stalling the machine.

freezer

Originally a regular controller **freezer** in v1, in v2 this feature has become a cgroup utility, available to all cgroups throughout the hierarchy, without having to enable it specifically. It creates the possibility to freeze and thaw all processes of the cgroup and is therefore similar to sending `SIGSTOP` and `SIGCONT` to the processes, however, it acts atomically on the whole group.

2.1.2 kill

In Linux 5.14 a similar utility to **freezer**, **kill** has been introduced to cgroup v2 [5], which lets a single write operation to a file, kill all processes in the cgroup. It sends `SIGKILL` to all processes and makes it impossible for them to create new processes, stopping all forking.

2.1.3 Pressure Metrics

In cgroup v2, a number of controllers, i.e. **cpu**, **io** and **memory** collect Pressure Stall Information (PSI) [6, 7]. The information provides insight into bottlenecks caused by one of the above resources. These PSI metrics are supplied as two types called `some` and `full`. They are measured and returned as averages over the last 10 seconds, 1 and 5 minutes. A `total` is provided additionally, the amount of milliseconds the processes are stalled since creation of the cgroup.

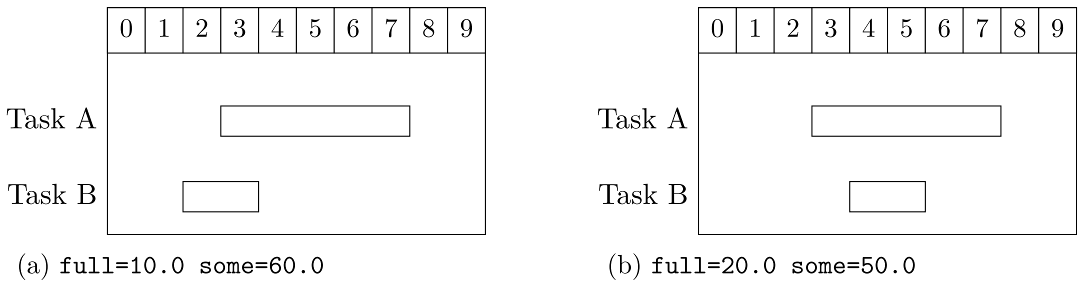

Figure 2.1: PSI 10-second example

Figure 2.1 shows two examples of stalled tasks. Both show `Task A` stalled from the timespan of 3 to 7 seconds. The first in Figure 2.1a additionally has `Task B` stalled during the span of 2 to 4 seconds. Looking at the 10-second metric `full` equates to `10.0`, because the timespan of all tasks being stalled at the same time is 1 second or 10 per cent of the time. The metric `some` is `60.0`, because at least one task is stalled for 6 seconds or 60 per cent of the time. In Figure 2.1b `full` is `20.0` and `some` is `50.0`.

2.2 cgroup v1

In cgroup v1 [8] every subsystem can be mounted at a separate mount point, or multiple controllers can be combined in one mount point, each mount point equating to one hierarchy. Having one mount point for each subsystem is equally possible, as well as having all attached to the same hierarchy.

In practice most Linux distributions mount all controllers separately, except for combining **cpu** & **cpuacct** and **net_cls** & **net_prof**.

Mounting one controller is done by:

```
$ mount --types cgroup --options memory none /sys/fs/cgroup/memory
```

Mounting multiple controllers at once:

```
$ mount --types cgroup --options cpu,cpuacct none /sys/fs/cgroup/cpu,cpuacct
```

It is, however, never possible to mount one controller on multiple hierarchies. Another possibility is mounting a cgroup hierarchy without controllers. There can be multiple as long as their names are unique. This mechanism can be used by any software to organise processes hierarchically. *systemd* tracks user sessions and services in a hierarchy that has no controllers attached.

```
$ mount --types cgroup --options none,name=example none /sys/fs/cgroup/example
```

A child cgroup is created by creating a directory in the cgroup file system. Processes are then moved to the cgroup by writing the PID to the `tasks` file. The special variable `$$` in the example is the PID of the current shell.

```
$ mkdir /sys/fs/cgroup/memory/bar
$ echo $$ > /sys/fs/cgroup/memory/bar/tasks
```

Reading the `tasks` file returns the PIDs attached to the cgroup, one per line. Each process can at all times only belong to one cgroup per hierarchy.

A child cgroup can subsequently be deleted by removing the directory, if it contains no non-zombie processes.

```
$ for p in $(</sys/fs/cgroup/memory/bar/tasks); do
      echo $p > /sys/fs/cgroup/memory/tasks
  done
$ rmdir /sys/fs/cgroup/memory/bar
```

Information on the cgroups of a process is read from `/proc/$PID/cgroup` for a given PID or `/proc/self/cgroup` for the information on the querying process itself. The information given is the mounted hierarchies, then per line the hierarchy ID, list of controllers and the relative path to the cgroup in the hierarchy. A typical user shell in a cgroup-v1-only Ubuntu 21.04 could look like the example in Figure 2.2.

```
12:blkio:/user.slice
11:rdma:/
10:perf_event:/
9:cpu,cpuacct:/user.slice
8:cpuset:/
7:memory:/user.slice/user-1000.slice/session-38.scope
6:pids:/user.slice/user-1000.slice/session-38.scope
5:devices:/user.slice
4:hugetlb:/
3:freezer:/
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-1000.slice/session-38.scope
```

Figure 2.2: Example `/proc/PID/cgroup`

The mounted cgroup hierarchies can be read from `/proc/cgroups` (see Figure 2.3), the fields in the output are the controller's name, the unique hierarchy ID, the number of cgroups in the hierarchy and whether the controller is enabled.

```
#subsys_name    hierarchy       num_cgroups     enabled
cpuset  8       3       1
cpu     9       100     1
cpuacct 9       100     1
blkio   12      100     1
memory  7       168     1
devices 5       100     1
freezer 3       4       1
net_cls 2       1       1
perf_event      10      1       1
net_prio        2       1       1
hugetlb 4       1       1
pids    6       106     1
rdma    11      1       1
```

Figure 2.3: Example `/proc/cgroups`

The mount points of the hierarchies, like any other in Linux, can be read from `/proc/mounts`.

For an example of a number of mounted cgroup v1 hierarchies and their contents, see Figure 2.4.

```
/sys/fs/cgroup/
├── blkio/
│   ├── tasks
│   ├── blkio.throttle.io_service_bytes
│   └── [...]
├── cpu -> cpu,cpuacct
├── cpu,cpuacct/
│   ├── tasks
│   ├── cpu.stat
│   ├── cpuacct.usage_percpu
│   ├── system.slice/
│   │   ├── tasks
│   │   ├── cpu.stat
│   │   ├── postgresql.service/
│   │   │   ├── tasks
│   │   │   ├── cpu.stat
│   │   │   └── [...]
│   │   └── [...]
│   └── [...]
├── cpuacct -> cpu,cpuacct
├── freezer/
│   └── [...]
├── memory/
│   └── [...]
├── systemd/
│   ├── tasks
│   ├── system.slice/
│   │   ├── tasks
│   │   ├── postgresql.service/
│   │   │   ├── tasks
│   │   │   └── [...]
│   │   └── [...]
│   └── [...]
└── [...]
```

Figure 2.4: Shortened cgroup v1 hierarchy
mounted controller hierarchies are shown in blue,
general cgroup files in orange,
controller-specific files in red,
cgroups themselves in green

2.3 Reasons for cgroup v2

A number of issues and shortcomings existed in cgroup v1 [3, 9, 10], mainly due to it having grown without any coordinated design across its controllers.

In some controllers, new cgroups inherit the parent's attributes, in others they are set to defaults. Also not all controllers follow the concept of cgroups being in a hierarchy, the **blkio** controller, while throttling, treats all cgroups as being at the same level.

Another issue is inconsistent use of values for a certain meaning, the maximum value is sometimes represented as `-1` and sometimes as `max`. In the **blkio** controller the total number of *shares* is `500`, in **cpu** it is `1024`. In cgroup v2 this inconsistency has been cleaned up, removing *shares* and having a consistent metric *weight* in the range of `1` to `10000` that always defaults to `100` across all controllers.

Comparable to this example similar issues were mitigated in v2 by having clearly documented guidelines in the kernel documentation.

2.3.1 Multiple Hierarchies

The intended flexibility in cgroup v1, allowing multiple hierarchies for different controllers, proved not to outweigh the added complexity. [11] Not having a single hierarchy, also reduces the practicality of the **freezer** controller. For example users cannot freeze a **cpu** cgroup, except when mirroring the structure across all hierarchies. *systemd* has implemented the mirroring approach.

2.3.2 Processes in Inner Nodes

In cgroup v1 tasks are allowed to be members of a cgroup that has further child cgroups. This is problematic if processes in a given subgroup are competing for resources with tasks attached to a parent group, as there is no obvious way to resolve the situation. Due to the lack of sufficient guidelines, there are inconsistencies on how this is handled, and interpreted differently across different controllers. This causes further issues if the hierarchies are mirrored, due to the different meaning of child and sibling relationships.

2.3.3 Permissions

By default the hierarchy is only modifiable by the administrative user, because the tree is mounted with *root* user as owner of all files and directories within. Access to less privileged users is granted by changing file permissions. This is referred to as *delegation*. In cgroup v1 processes can be moved from anywhere to a delegated subtree as long as it is a process started by the user, this missing boundary can enable users to stall the system, by working around limits imposed on different subtrees.

2.4 Unified Hierarchy

In cgroup v2 there is one unified hierarchy for all controllers [12]. Its implementation started in Linux 3.10, and was released as a stable implementation in Linux 4.5. All subsystems not mounted in a cgroup v1 hierarchy are enabled at the root node of the hierarchy and access to specific controllers can be passed deeper into the hierarchy. If a controller is disabled in a cgroup, it cannot be enabled in any further descendants. Figure 2.5 shows a shortened directory tree of the mounted cgroup v2 hierarchy.

```
/sys/fs/cgroup/
├── cgroup.controllers
├── cgroup.procs
├── cgroup.subtree_control
├── cpu.pressure
├── cpu.stat
├── memory.stat
├── [...]
├── system.slice/
│   ├── postgresql.service/
│   │   ├── cgroup.controllers
│   │   ├── cgroup.freeze
│   │   ├── cgroup.kill
│   │   ├── cgroup.procs
│   │   ├── cgroup.subtree_control
│   │   ├── cpu.max
│   │   ├── cpu.pressure
│   │   ├── cpu.stat
│   │   ├── cpuset.cpus.effective
│   │   ├── memory.stat
│   │   ├── memory.high
│   │   ├── memory.pressure
│   │   ├── memory.max
│   │   ├── memory.events
│   │   └── [...]
│   └── [...]
└── user.slice/
    ├── cgroup.controllers
    ├── cgroup.freeze
    ├── cgroup.kill
    ├── cgroup.procs
    ├── cgroup.subtree_control
    ├── cpu.pressure
    ├── cpu.stat
    ├── memory.stat
    ├── user-1000.slice/
    │   └── [...]
    └── [...]
```

Figure 2.5: Shortened cgroup v2 hierarchy
mounted controller hierarchies are shown in blue,
general cgroup files in orange,
controller-specific files in red,
cgroups themselves in green

There are a number of base cgroup files to control controller availability, process membership or give general information.

cgroup.controllers shows all available controllers for this sub-cgroup. In the root of the *cgroupfs* all available subsystems are returned, i.e. all not bound to cgroup v1 hierarchies.

cgroup.subtree_control enables or disables controllers for child cgroups and allows those resources to be controlled there. Enabling a subsystem creates the controller-specific attribute files in each child directory.

Creating a child cgroup is creating equivalent to creating a new directory in its parent.

```
$ mkdir subgroup1
```

Activating controllers for the cgroup is done by writing to the parent's `cgroup.subtree_control`, prefixing the subsystems with `-` to disable and `+` to enable them. Multiple changes can be written to the file in one command.

```
$ echo "+memory -cpuset" > cgroup.subtree_control
```

cgroup.procs lists the PIDs of processes belonging to this cgroup. Writing a PID to this file moves the process to the cgroup. At any time one process can only belong to one single cgroup.

2.4.1 Resource Constraints

Child cgroups are always subject to any resource constraints defined by controllers in ancestor cgroups. Constraints cannot be relaxed in child cgroups.

2.4.2 No Processes in Inner Nodes

In cgroup v2 processes can only be attached to leaf nodes, not to an internal subgroup as long as any controller is enabled. That means it is not possible to enable controllers in `cgroup.subtree_control` and have processes in `cgroups.procs` at the same time. The only exception to this restriction is the root cgroup.

2.4.3 Delegation

To pass management of a subtree to another less privileged user, write permissions have to be set on the directory at the root of the subtree to be delegated and the files `cgroups.procs`, `cgroups.subtree_control` inside this directory. That allows the delegatee to control resources in child cgroups that are created. Permissions must also be granted on any other files listed in `/sys/kernel/cgroup/delegate`.

2.4.4 Permission Boundaries

Processes can only be moved if the user has write permissions to all `cgroups.procs` files in the subtree, between the origin and destination cgroups, through which the process is moved, up to the lowest common ancestor node.

Thus, when delegating a cgroup, the first process has to be moved by the delegater to the delegated cgroup.

2.4.5 cgroup v2 Adoption

Adoption was relatively slow, because most projects using cgroup, especially container systems, e.g. DOCKER, *Kubernetes*, did not feel the need to upgrade quickly, because most Linux distributions did not support it. However, distributions could not switch to v2 either, because most software had not added support. Also the **cpu**, **cpuset** and **freezer** controller were only ported to cgroup v2 in Linux 4.15, 5.0 and 5.2 respectively, blocking adoption for aforementioned container systems.

systemd

Initial support to systemd was added in version 230 in May 2016 [13], but still had to be turned on at build-time by distributions or by setting a kernel parameter at boot.

```
systemd.unified_cgroup_hierarchy=1
```

In version 243 in September 2019 the build-time default was changed, but most distributions still left it deactivated. It could also be turned off at boot-time with the kernel parameter set to `0`.

Fedora

The first major Linux distribution to use cgroup v2 by default was Fedora in version 31 in October 2019 [14]. Citing the issues in Section 2.3, the improvements elaborated in Section 2.4, and explicitly wanting to break the dead lock between distributions and users of cgroup not upgrading, they moved forward first.

Debian and Ubuntu

Following software upgrades to also support cgroup v2, Debian decided to change the default on their unstable version in systemd version 247.2-2. The switch landed in the stable version 11 “Bullseye” in August 2021 [15]. With this, Ubuntu also changed the default in version 21.10, released in October 2021 [16].

Docker

Over one year after the switch in Fedora, Docker released version 20.10 in December 2020 [17], adding support for cgroup v2. Previously, users had to switch back to v1 to use it with the systemd kernel parameter mentioned previously [18].

2.5 cgroup and systemd

Systemd makes extensive use of cgroup, having the complete service and scope hierarchy represented as a cgroup hierarchy. In systemd there are three types of units—the systemd term for any configurable entity—that are directly represented in the hierarchy. A **slice** [19] is an inner node. The leaves are either a **scope** [20] or **service** [21]. The former is a collection of processes in a single cgroup not directly managed by systemd, such as user sessions. All processes managed by systemd are in **service**s, each also residing in their cgroup.

The cgroup hierarchy with the root cgroup named `-.slice` is split into three main parts: the system services in `system.slice`, `machine.slice` containing all virtual machines and containers, and `user.slice` being made up of a further systemd instance per user and the user's sessions and services. [22]. The main systemd instance running as PID 1 is directly beneath the root in the `init.scope`.

For example a user hierarchy for the user with UID 1000 is created as `user-1000.slice` in the main `user.slice`. This contains the user sessions in `session-N.scope` and the user systemd-daemon and services in `user@1000.service/init.scope` and `user@1000.service/app.slice` respectively.

The tree in Figure 2.6 is a shortened example of the systemd hierarchical structure. This tree is the description of the single hierarchy in cgroup v2 and is mirrored across all hierarchies in cgroup v1. There are clear guidelines that systemd is the sole manager of the hierarchy if a subtree is not explicitly delegated [23]. Also, systemd mandates that even when using cgroup v1, the no-processes-in-inner-nodes rule (Subsection 2.4.2) applies, even when not enforced by the kernel as in v2.

To delegate a subtree in systemd, the `Delegate` property can be set on a **`service`** or **`scope`** unit. This guarantees that systemd does not touch the cgroup hierarchy and its attributes below this unit and does not migrate processes across the boundaries of the subtree. If the `User` property is also set and the unified hierarchy is used, the owner of the subtree is set to that user. User permissions are not granted when using cgroup v1 controllers, due to it being unsafe as mentioned in Subsection 2.3.3.

Furthermore, three options for delegation are outlined in the guidelines by the systemd project. [23]

1. Each process needing cgroup delegation can directly be registered as either a systemd service or a scope, depending on whether systemd executes the binary or the managing software. Having `Delegate` switched on for these services or scopes, they can manipulate the cgroup hierarchy beneath their location in the tree.

2. The managing software itself is started in a service with `Delegate` turned on. The manager must subsequently move itself to a child cgroup in order to not violate the no-processes-in-inner-nodes rule.

3. Moving all processes to a single scope unit is a third option, where the cgroup sub-hierarchy can further be modified, with `Delegate` enabled on the scope.

In 1 and 3 communication has to be established with systemd through *D-Bus*, so that systemd can either initially move the processes to the scopes, or start the services. Option 2 can be a simple service and everything beneath can be managed by it without the need for communication with systemd.

Transient services and scopes can also be created from the command line with `systemd-run` [24]. This command is used to start an executable directly in a delegated subtree with `-p Delegate=yes` without having to touch systemd configuration. [25]

On minimal systems there is not always a D-Bus user-session available which is necessary to interact with the systemd user daemon. On Debian and Ubuntu for example, the package `dbus-user-session` is necessary if no graphical environment is used.

```
-.slice
├─user.slice
│ └─user-1000.slice
│   ├─user@1000.service
│   │ ├─app.slice
│   │ │ ├─gammastep.service
│   │ │ │ └─4257 /nix/store/[...]-gammastep-2.0.7/bin/gammastep [...]
│   │ │ └─[...]
│   │ └─init.scope
│   │   └─3840 /nix/store/[...]/bin/systemd --user
│   └─session-1.scope
│     ├─3852 /nix/store/[...]-sway-unwrapped-1.6.1/bin/sway
│     └─[...]
├─init.scope
│ └─1 systemd
├─system.slice
│ ├─postgresql.service
│ │ ├─3380 /nix/store/[...]-postgresql-11.13/bin/postgres
│ │ ├─3492 postgres:  checkpointer
│ │ ├─3493 postgres:  background writer
│ │ ├─3494 postgres:  walwriter
│ │ ├─3495 postgres:  autovacuum launcher
│ │ ├─3496 postgres:  stats collector
│ │ └─3498 postgres:  logical replication launcher
│ └─[...]
└─machine.slice
  └─machine-qemu\x2d1\x2dubuntu20.10.scope
    └─libvirt
      ├─1868690 /run/libvirt/nix-emulators/qemu-system-x86_64[...]
      ├─emulator
      ├─vcpu0
      └─[...]
```

Figure 2.6: Shortened systemd cgroup hierarchy
cgroups are shown in black, processes in blue

By default, systemd does not delegate all controllers to the user daemon [26, 27]. Further controllers can be passed to the per-user cgroups by editing the generic `user@.service` or the user-specific `user@1000.service` (for the user with UID 1000). Editing the systemd service is possible using `systemctl` [28].

```
$ systemctl edit user@1000.service
```

To enable all available controllers, the following line needs to be added. `Delegate` can be set to a space-separated list of controller names, to only activate specific controllers.

```
[Service]
Delegate=yes
```

Both the main systemd instance and those of affected users have to be reloaded after the change or the system needs to be rebooted. The controllers delegated to a user are listed in the cgroup of the service above.

```
$ cat /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/cgroup.controllers
cpuset cpu io memory pids
```

systemd also provides two command-line tools to introspect the cgroup hierarchy: `systemd-cgls` [29] to display the hierarchy in a tree, like the shortened version in Figure 2.6, and `systemd-cgtop` [30] that returns resource consumption per cgroup.

To interact with cgroups from systemd, a number of parameters can be set on slices, scopes or services. [31] Most controller limits can be defined with these options and can also be set, while the software is running, with `systemctl set-property`. [32]

3 Background on BenchExec

The BenchExec benchmarking framework consists of three parts. `runexec` is the tool to execute single isolated runs of the tool to be benchmarked. It sets up the encapsulation and limits with cgroup and namespaces, and collects the metrics of the run. The second, `benchexec`, is in control of starting `runexec` instances for a number of tools and inputs, in order to benchmark a whole batch of experiments. The third tool, `table-generator`, creates reports of the benchmarks by post-processing the output of `benchexec`.

Using cgroups in conjunction with namespaces is the only mechanism to reliably collect information on multiple processes, limit them to specific resources, and terminate all of them to not influence future benchmarks. Working with a group of processes is necessary to prevent measurement errors on software that uses subprocesses. It also ensures that spawned processes are subject to all limits defined for the benchmark at all times. [1]

3.1 cgroup Setup

If the cgroup in each hierarchy, in which `runexec` is started, is writable by the executing user, these cgroups are used, and further child cgroups are created within. Should this not be the case for one or more subsystems used by BenchExec, a fallback mechanism is provided.

The `.deb` package contains a systemd service, `benchexec-cgroup.service`, with `Delegate` enabled. During startup it ensures one cgroup, for each subsystems relevant to BenchExec, exists at `system.slice/benchexec-cgroup.service` relative to the root of the controller's hierarchy. systemd will by default already have created it, if it manages this hierarchy. Subsequently the service script moves itself into these cgroups and makes them writable by the `benchexec` user group which is created during installation of the `.deb` package. The service then sleeps for 10 years to keep the delegation alive and automatically restarts at termination.

This service lets all members of the `benchexec` group create cgroups beneath this fallback path if permissions in the current cgroups are not available. To run `benchexec` no further administrative intervention is necessary after installation of this service.

3.2 cgroup Usage

When benchmarks are run, a child cgroup is created for each `runexec` execution in the current cgroups or in the fallback cgroups. Essentially BenchExec mirrors the cgroup structure across all hierarchies, although the names are not identical as they are created randomly.

A number of controllers are used to set up the consistent environment of the benchmarking process. **cpuset** is utilised to constrain the run to a specified set of CPU cores and NUMA nodes. The **memory** subsystem is used to limit the maximum memory available and deactivate swap during the benchmark. It also makes it possible to replace the Linux out-of-memory killer with a notification, for which a thread listens. This then, when notified, reports `out-of-memory` as the reason for termination and kills the remaining processes.

That subsystem is also one of the controllers used for metric collection, reporting the peak memory usage of all processes combined. The **cpuacct** subsystem is utilised to collect the sum of CPU time of the benchmark and also return these statistics per CPU core. The **blkio** controller provides information on the number of bytes read and written on block devices.

To reliably kill all processes of a run, in case of out-of-memory, a time limit is exceeded, or the execution has completed and to avoid left-over processes the **freezer** subsystem is used to freeze all processes. Then the processes are killed and thawed again, to let them disappear. Subsequently the cgroups can be removed.

4 Changes to BenchExec for cgroup v2

4.1 Requirements

The expectation of the updated implementation is to support cgroup v1 and cgroup v2 in parallel. Also where possible the implementation must maintain feature parity and not influence benchmarking negatively.

Ideally with the improved possibility for delegation, tasks by an administrative user should be reduced and BenchExec can be installed and used by an unprivileged user. At the same time it must additionally adhere to new restrictions within the cgroup structure imposed by the kernel. It also must consider systemd and their recommendations for delegation, to avoid mutual interference, when managing the cgroup tree.

Furthermore, the `runexec` executable and Python module must still be usable as a simple facility to run benchmarks that are encapsulated and have resource limits set.

4.2 General Approach

The implementation constructs an abstraction around the existing cgroup code, delegating calls to code, handling the details for the detected cgroup version. Detection is done by checking `/proc/mounts` for mounted file systems of type `cgroup` or `cgroup2`. As soon as one `cgroup` mount is found, the existing code for v1 is used, as the mounted hierarchy can then not be bound to the `cgroup2` unified hierarchy. This occurs commonly on recent versions of Linux distributions that have not switched to cgroup v2 completely, all cgroup v1 controllers and the cgroup v2 hierarchy are mounted, without any controllers being active in v2. This behaviour makes it useless for our purpose, so this setup is ignored and the v1 implementation is used. [33]

The API to the existing cgroup code is mostly kept in place, a few helpers and abstractions are added where the existing code referred directly to cgroup v1 implementation details, for example filenames that changed, such as the `tasks` file to `cgroup.procs`.

4.3 cgroup Permissions

A larger necessary change, is the different handling of cgroups setup for benchmarking. It is no longer possible to use the previous fallback mechanism (Section 3.1), as all system services are created under `system.slice` and all user sessions are in the `user.slice`. The `cgroups.procs` files in between are root-owned, and the unified hierarchy imposes the permission boundaries, not permitting moving processes from one delegated hierarchy to another. This approach also does not work with user services because the same boundary exists between these and user sessions, the `cgroups.procs` file in the `user-UID.slice` being owned by the root user. The sleeping service would also have to be moved to a child, to adhere to the rule of having no processes attached to inner nodes (Subsection 2.4.2).

To alleviate this, `benchexec` now handles checking whether it is in a suitable cgroup, that is being the only process therein and having permissions to create children and delegate controllers to these. If this check fails, it tries to communicate with systemd over D-Bus, to move itself to a transient scope unit that has delegation enabled.

An alternative with systemd—apart from directly starting `benchexec` in a suitable cgroup, which is always possible—is using `systemd-run` to move the process at execution to a user scope with delegation enabled.

```
systemd-run --user --scope -p Delegate=yes benchexec ...
```

`runexec` now expects cgroup delegation and permission handling to be taken care of by the software calling it. This can either be `benchexec` or when using it directly as an executable or library, the cgroup setup has to be ensured by the user. A simple possibility is to use the mentioned `systemd-run` call.

4.4 Library to interact with systemd

To handle cgroup setup without user interaction BenchExec needs to communicate with systemd. The easiest, and by systemd recommended option, is to use the D-Bus API they provide.

4.4.1 Available Libraries

There are a number of Python libraries to communicate either with D-Bus or with systemd specifically.

dbus-python[1] is the reference implementation for D-Bus in Python with the first standalone release in July 2006. It still receives regular updates with its last release in July 2021. The authors themselves note that it might not be the best library to use, but some issues cannot be fixed without breaking compatibility. The only dependency is libdbus, but the authors also advise that it has problems with multi-threading.

gdbus[2] is part of the glib project and the D-Bus library the GNOME project uses. The last release happened in December 2021 and updates are frequent, but the API from Python is used through gi and gobject and not a clean, native API. Also Python documentation is lacking and only a few examples are available.

qtdbus[3] is maintained as part of Qt with regular releases. It has quite a large number of dependencies because it depends on parts of the Qt toolkit.

pydbus[4] is a nicer API around gdbus. Hence, it also depends on gi and gobject and needs a *glib event loop* to execute the code. Its last release was in December 2016 and there has been no activity on the repository since May 2018 and no reaction to neither issues nor pull requests.

[1] https://dbus.freedesktop.org/doc/dbus-python/
[2] https://docs.gtk.org/gio/
[3] https://doc.qt.io/qtforpython-6/PySide6/QtDBus/QDBusConnection.html
[4] https://github.com/LEW21/pydbus

dasbus[5] is a library that was originally based on PYDBUS, but largely rewritten. It has a further improved API and is actively maintained with its last release in May 2021. It also depends on `gi` and a *glib event loop*. For this library there is no package in stable Ubuntu and Debian releases.

txdbus[6] is a D-Bus implementation for the TWISTED[7] framework, with the last release in October 2020, but recent activity in repository and active maintenance. In needs the *Twisted reactor* to run, but has a modern and simple API. No system packages for it exist in any larger Linux distribution.

pysdbus[8] is a little library based on LIBSYSTEMD and their D-Bus implementation. It mainly is a simple `ctypes` wrapper. The author warns that the library still is at an early stage and many features are still missing. There has not been any release and the repository consists of only 10 commits. Also it is not included in any Linux distribution nor published on PyPI.

jeepney[9] is a library without any dependencies and a Python-only implementation, the last release having been in July 2021. The API is low-level without many helpers or introspection.

python-systemd[10] is provided by the systemd project but does not serve the purpose of creating transient services or scopes but rather for services implemented in Python to notify systemd that they finished their startup, or for journal and systemd-login interaction.

pystemd[11] is similar to PYSDBUS, also using LIBSYSTEMD—its only dependency—through `ctypes` to interact with systemd. It includes more helper functions to provide a more user-friendly API. Its last release was in October 2021 and is installable in major Linux distributions such as Ubuntu, Debian and Fedora as a system package.

4.4.2 Interaction with systemd

Based on the above, and criteria of availability in Linux distributions, active maintenance, small number of dependencies, and a user-friendly and simple API, the choice of library used in this implementation is PYSTEMD. If it is installed, it is used to create a transient `scope` on the systemd instance of the executing user, with the `Delegate` property set. All PIDs of the running BENCHEXEC instance are passed to the call that create the `scope`. SYSTEMD then handles moving those processes to the new scope and the newly created cgroup that is delegated to the executing user.

To find all relevant PIDs, the process group of the current process is queried. This is done by iterating through the `/proc/PID/stat` files. All processes that contain the process group ID of the main process are searched for. [34] The reason for this, is to also include

[5] https://dasbus.readthedocs.io/en/latest/
[6] https://github.com/cocagne/txdbus
[7] https://www.twistedmatrix.com/
[8] https://github.com/anyc/pysdbus
[9] https://gitlab.com/takluyver/jeepney/
[10] https://github.com/systemd/python-systemd
[11] https://github.com/facebookincubator/pystemd

forked processes, created during namespacing and containerisation setup for the benchmark execution.

4.5 Differences of Controllers

Apart from changes because of general cgroup management, the subsystems themselves have changed too. The **blkio** subsystem is replaced with the **io** controller and the metrics used in BenchExec are now fetched from `io.stat` instead of `blkio.throttle.io_service_bytes`. Due to file structure changes, the parsing of the files also needs minor changes.

The **cpu** controller now always provides basic information even if not explicitly turned on. For the purposes of BenchExec, this information (CPU usage) is sufficient, so the check for controller availability is not necessary. In the v1 implementation this metric is provided by the **cpuacct** controller together with per-CPU metrics. These are no longer available in cgroup v2, although as of January 2022, there is active discussion on the Linux Kernel Mailing List[12] to add them back.

Another metric that is no longer available in v2, is the peak memory usage of a cgroup which is read from either `memory.max_usage_in_bytes` or `memory.memsw.max_usage_in_bytes` depending on availability of swap, but no replacement is implemented in v2. The out-of-memory handling, on the other hand, has been simplified especially for the functionality used in BenchExec, a file `memory.events` can now be watched and in case any memory limit is surpassed an event is emitted. Information if a process was killed due to an out-of-memory event can now be read from a `oom_kill` counter in this file. Previously swap was turned off by writing `0` to `memory.swappiness`, in v2 by writing `0` to `memory.swap.max`.

To read available CPU cores and memory nodes and limit the execution to a set of these, `runexec` reads from and writes to `cpuset.cpus` and `cpuset.mems`. For v2 setting the constraints is still done with these files. Information on available cores and memory nodes is read from `cpuset.cpus.effective` and `cpuset.mems.effective`, as these list the actually granted resources including constraints imposed by parent cgroups.

To reliably kill processes, they are all frozen by writing `FROZEN` to the `freezer.state` file, and unfrozen by writing `THAWED`. Being an ordinary controller in v1, now in v2, it is a utility, always available in all cgroups. Processes are frozen or unfrozen by writing `1` or `0` respectively to `cgroup.freeze`. Another such utility was added in Linux 5.14, alleviating BenchExec's need for the **freezer**. By writing `1` to `cgroup.kill`, all processes are killed atomically. This feature is implemented in the code for cgroup v2, falling back to the freezing option if the Linux kernel on the system does not yet provide support.

4.6 PSI

When running with cgroup v2 BenchExec now provides three additional metrics `total-cpu-pressure-some`, `total-io-pressure-some` and `total-memory-pressure-some`. For CPU, memory, and I/O, these report the number of seconds any of the processes running in the benchmarked is stalled by the respective resource. The metric `some` is used because it is relevant as soon as one of the processes in the cgroup is affected and not only when all

[12] https://lkml.org/lkml/2022/1/7/833

processes are; the `total` is the interesting metric versus information on the last 10, 60 or 300 seconds, because this is information is also relevant to benchmarks that run longer than five minutes, and resource exhaustion prior to those time spans would not be detected.

5 Evaluation

To check the compatibility and feature-parity of the updated cgroup implementation, a few test benchmarks are performed. They are executed on a physical server with an `Intel(R) Xeon(R) CPU E3-1246 v3 @ 3.50GHz` CPU and 32 GB of memory running Ubuntu 21.10 with Linux 5.13. All runs are performed three times, once with BenchExec 3.10 running cgroup v1, and twice with the described implementation from the `cgroupv2` branch of the BenchExec GitHub repository[1] at revision `4d32b32`, and with `pystemd` 0.8.0, the latest version on **PyPI**. This version is used once with cgroup v1 and once with cgroup v2. To test with v1 the machine was booted with the `systemd.unified_cgroup_hierarchy` set to `0`.

5.1 Regular Benchmark

To verify that general functionality is provided, a subset of the *International Competition on Software Verification* 2022 (SV-COMP 2022) taskset[2] (git tag `svcomp22`) is run with BenchExec on CPAChecker[3] 2.1. The input set is the `ReachSafety-ControlFlow.set` run with BenchExec parameters of `timelimit=15m`, `hardtimelimit=16m`, `memlimit=31G` and `cpucores=8`. The parameters to CPAChecker are `heap=25000G` and `timelimit=900s`.

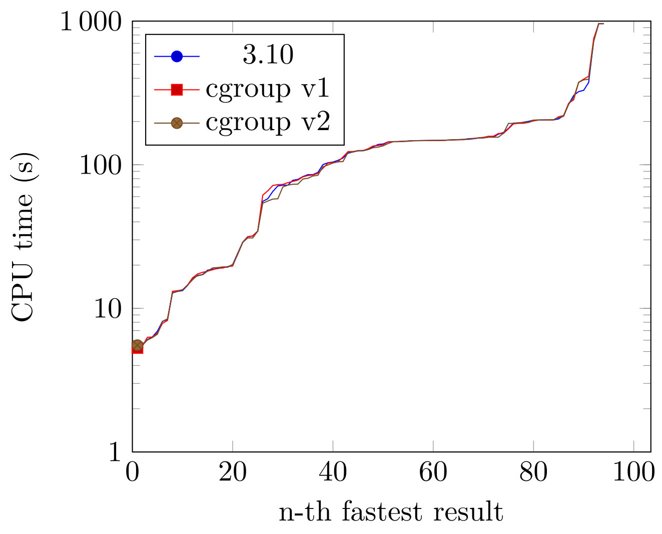

Figure 5.1: Quantile plot of CPU time

The statistics of the three benchmarks exhibit a high amount of overlap. The statuses of the runs are exactly the same, each with 88 correct results. The measured CPU time also is

[1]https://github.com/sosy-lab/benchexec
[2]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/tree/svcomp22
[3]https://cpachecker.sosy-lab.org/

very similar and differs by only less than 1% which is not a significant variation. Figure 5.1 shows a quantile plot of the CPU time of the three runs. Memory usage in the two cgroup v1 implementations also only shows insignificant deviations, in the v2 implementation it is not available due to cgroup v2 not exposing that data.

5.2 Feature Verification

To verify specific features in the updated implementation, a few short benchmarks with specific limits are run. The same machine as in the preceding experiment and the same BenchExec versions are used. The benchmark definition used for all runs is the `doc/benchmark-example-calculatepi.xml` in the BenchExec repository. This input file calculates pi to a specified number of digits in steps of 1000, starting with 1000 to 20000.

First, for verification, the benchmark is run on all versions without parameters, and returns results with insignificant variations for each. To test **CPU time limit** functionality, the limit is set to 10 seconds. Based on the previous runs, the benchmark is expected to time out for all steps starting at 5000. The expectation is observed successfully for each version.

To confirm correct functionality of **memory limit**s, the benchmark is run with a limit of 800 kB. Judging from the two runs of the cgroup v1 versions, the restriction should cause out-of-memory errors for runs beginning at the 9000 or 10000 steps. This constraint is not as deterministic as the previous constraint, because the benchmarked process might release memory if no further memory can be allocated, but otherwise not as early. Therefore with a restriction imposed, that limit is reached at a later point. For all three runs the expected out-of-memory errors occur at the 11000 step and all runs are correct as such.

6 Related Work

For benchmarking software to use cgroup is not very common. Because of this, there is not a lot of directly related work with comparable issues, but cgroup as laid out in Chapter 3 is the state-of-the-art option for benchmarking.

Mininet[1] is an emulator for prototyping Software Defined Networks. They include a benchmarking tool that collects CPU usage with the **cpuacct** controller. So far they have not upgraded to support cgroup v2 but there is ongoing discussion after requests for this have been made. [35] They use libcgroup[2] which only added support for v2 in May 2021 [36]. Only updating to the new version, though, is not sufficient for Mininet, as changes between v1 and v2 are not abstracted completely. This means that switching to libcgroup or having used it in the first place, would not have avoided the need of changes to the cgroup-related code.

LXC[3] and crun[4] have the same issues with user-space containers and cgroup permissions as BenchExec. The former therefore recommends running in a transient systemd user scope with delegation enabled using `systemd-run` [37], as laid out as a possibility in Section 4.3. The latter supports communication with systemd through D-Bus to not require user interaction, directly using systemd's D-Bus C library, similar to the use of pystemd in BenchExec (Subsection 4.4.2).

Podman[5] also recommends enabling extra cgroup controllers in the `user@.service` time to support use of these in user-space containers. (Section 2.5) [38]

Docker and Kubernetes take a different approach because they generally have daemons running as root that can handle the cgroup setup and have sufficient privileges to move processes across cgroups. Although Docker does have the possibility to run the daemon with user privileges, this requires that systemd and cgroup v2 is used on the machine, because they use systemd to set up delegation. They also document the change necessary in the `user@.service` to enable further controllers enabled in cgroups delegated to users. [39]

[1] https://github.com/mininet/mininet
[2] https://github.com/libcgroup/libcgroup/
[3] https://linuxcontainers.org/
[4] https://github.com/containers/crun
[5] https://podman.io/

7 Conclusion

The results of the benchmarks run in Chapter 5 show that—with the exception of missing support for the memory usage and per CPU usage metrics for cgroup v2—the implementation in the course of this thesis, provides feature parity to the previous state and between cgroup v1 and v2.

Implementing the missing features would have required changes to the Kernel or possibly extraction of the metrics with *extended Berkeley Packet Filters* (eBPF) [40], which might provide the necessary information, but needs further investigation. This however was not in scope for this thesis. Most importantly though, this makes it possible to run BenchExec on modern versions of Linux distributions that have already switched to cgroup v2 and allows using features that are added to the Linux kernel relating to cgroups and no longer implemented for cgroup v1, such as the `cgroup.kill` utility and PSI metrics, already implemented in this thesis.

Additionally, when running with cgroup v2 and systemd, BenchExec now harnesses the improved delegation possibilities and no longer requires setup by an administrative user to run, by either the user using `systemd-run` or with the optional library pystemd, used to communicate with systemd over their D-Bus API. This interaction makes BenchExec adhere to systemd's recommendations regarding delegation and cgroup management.

7.1 Future Work

The improved permission and delegation handling, including support for delegation boundaries at namespace boundaries, provided by cgroup v2, makes it possible to implement a feature, that benchmarked processes themselves can use cgroups. This also is made possible by cgroup v2 strictly imposing that limits, set by parent cgroups, need to be respected in all children cgroups.

BenchExec, now necessitated by processes not being able to run in inner cgroup nodes, runs in its own cgroup and metrics on itself can also be collected.

cgroup v2 also improves usability and availability of soft limits, that can be used to communicate to processes that they are reaching, e.g. memory limits, so that they can potentially react to that, without having to abort the benchmark.

Another big open issue are the missing metrics on memory usage, due to the **memory** controller in cgroup v2 no longer providing the information. It could be resolved by implementing support in the kernel for cgroup v2, checking whether the data can be retrieved with eBPF or that this issue must be solved by other means with further investigation into this matter.

List of Figures

Bibliography

[1] D. Beyer, S. Löwe, and P. Wendler, “Reliable benchmarking: Requirements and solutions,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 21, no. 1, pp. 1–29, 2019.

[2] P. B. Menage, “Adding generic process containers to the linux kernel,” in *Proceedings of the Linux symposium*, Citeseer, vol. 2, 2007, pp. 45–57.

[3] *Cgroups(7) linux manual page*, 5.13.

[4] R. Rosen, “Namespaces and cgroups, the basis of linux containers,” *Seville, Spain, Feb*, 2016.

[5] J. Corbet. “A ”kill” button for control groups,” LWN. (2021-05-03), [Online]. Available: `https://lwn.net/Articles/855049/` (visited on 2021-12-27).

[6] “Getting started with psi,” [Online]. Available: `https://facebookmicrosites.github.io/psi/docs/overview.html` (visited on 2021-12-27).

[7] “Psi pressure metrics,” [Online]. Available: `https://facebookmicrosites.github.io/cgroup2/docs/pressure-metrics.html` (visited on 2021-12-27).

[8] “Control group v1,” [Online]. Available: `https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html` (visited on 2022-01-02).

[9] “Control group v2,” [Online]. Available: `https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v2.rst` (visited on 2022-01-02).

[10] ——, “The past, present, and future of control groups,” LWN. (2013-11-20), [Online]. Available: `https://lwn.net/Articles/574317/` (visited on 2022-01-02).

[11] ——, “Fixing control groups,” LWN. (2012-02-28), [Online]. Available: `https://lwn.net/Articles/484251/` (visited on 2022-01-02).

[12] R. Rosen. “Understanding the new control groups api,” LWN. (2016-03-23), [Online]. Available: `https://lwn.net/Articles/679786/` (visited on 2022-01-02).

[13] “Systemd/news,” [Online]. Available: `https://github.com/systemd/systemd/blob/main/NEWS` (visited on 2022-01-09).

[14] D. Walsh. “Fedora 31 and control group v2.” (2019-11-11), [Online]. Available: `https://www.redhat.com/sysadmin/fedora-31-control-group-v2` (visited on 2022-01-09).

[15] M. Biebl. “Accepted systemd 247.2-2 (source) into unstable.” (2020-12-21), [Online]. Available: `https://tracker.debian.org/news/1204112/accepted-systemd-2472-2-source-into-unstable/` (visited on 2022-01-09).

[16] L. Märdian. “Systemd enabling cgroup v2 by default (default-hierarchy=unified).” (2021-08-17), [Online]. Available: `https://lists.ubuntu.com/archives/ubuntu-devel/2021-August/041598.html` (visited on 2022-01-09).

[17] A. Suda. “New features in docker 20.10 (yes, it’s alive).” (2020-12-09), [Online]. Available: https://medium.com/nttlabs/docker-20-10-59cc4bd59d37 (visited on 2022-01-09).

[18] ——, “The current adoption status of cgroup v2 in containers.” (2019-10-29), [Online]. Available: https://medium.com/nttlabs/cgroup-v2-596d035be4d7 (visited on 2022-01-09).

[19] *System.slice(5) manual page*, 249.

[20] *Systemd.scope(5) manual page*, 249.

[21] *Systemd.service(5) manual page*, 249.

[22] *Systemd.special(5) manual page*, 249.

[23] “Control group apis and delegation.” (2018-04-20), [Online]. Available: https://systemd.io/CGROUP_DELEGATION/ (visited on 2022-01-09).

[24] *Systemd-run(1) manual page*, 249.

[25] “The new control group interfaces,” [Online]. Available: https://www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/ (visited on 2022-01-09).

[26] L. Poettering. “Cgroup controllers are not activated for the user instance in the unified hierarchy.” (2018-05-24), [Online]. Available: https://github.com/systemd/systemd/issues/3500#issuecomment-391675687 (visited on 2022-01-09).

[27] “Cgroups - user delegation,” [Online]. Available: https://wiki.archlinux.org/title/Cgroups#User_delegation (visited on 2022-01-09).

[28] *Systemctl(1) manual page*, 249.

[29] *Systemd-cgls(1) manual page*, 249.

[30] *Systemd-cgtop(1) manual page*, 249.

[31] *Systemd.resource-control(5) manual page*, 249.

[32] M. Richter. “World domination with cgroups part 8: Down and dirty with cgroup v2,” Red Hat Blog. (2020-08-05), [Online]. Available: https://www.redhat.com/en/blog/world-domination-cgroups-part-8-down-and-dirty-cgroup-v2 (visited on 2022-01-09).

[33] L. Poettering. “How does hybrid cgroup setup work?” systemd-devel. (2017-11-10), [Online]. Available: https://lists.freedesktop.org/archives/systemd-devel/2017-November/039754.html (visited on 2022-01-09).

[34] *Proc(5) linux manual page*, 5.13.

[35] “[feature request] support for cgroup2,” [Online]. Available: https://github.com/mininet/mininet/issues/1051 (visited on 2022-01-09).

[36] “Add cgroup v2 support,” [Online]. Available: https://github.com/libcgroup/libcgroup/issues/12 (visited on 2022-01-09).

[37] “Add cgroup v2 support,” [Online]. Available: https://linuxcontainers.org/lxc/getting-started/#creating-unprivileged-containers-as-a-user (visited on 2022-01-09).

[38] “Podman - troubleshooting,” [Online]. Available: https://github.com/containers/podman/blob/3fac03cf04e68eb3351aff8c33bac6bea85810f6/troubleshooting.md#26-running-containers-with-cpu-limits-fails-with-a-permissions-error (visited on 2022-01-09).

[39] “Run the docker daemon as a non-root user (rootless mode),” [Online]. Available: https://docs.docker.com/engine/security/rootless/ (visited on 2022-01-09).

[40] J. Corbet. “Extending extended bpf,” LWN. (2014-07-02), [Online]. Available: https://lwn.net/Articles/603983/ (visited on 2022-01-09).