

# Service Management with systemd

Michal Sekletár  
msekleta@redhat.com

April 15, 2021

- Principal Software Engineer @ Red Hat
- systemd maintainer for Red Hat Enterprise Linux
- Open source software contributor

# Agenda

- systemd recap
- Cgroup v2 and resource management
- Service sandboxing

- PART I: systemd 101

# What is systemd?

- Implementation of the `init` process, PID 1
- Service manager
- Compatible with SysVinit (modulo [Documented incompatibilities](#))
- Open source project that provides basic user-space for Linux distributions
- Growing community of developers and users (All Systems Go!)

# Components of systemd

systemd	init	machined	VM/container registry
udev	Dynamic device management	localed	DBus API for locale and language settings
journal	Log aggregator	hostnamed	Hostname setting
logind	Session tracking	timedated	Time synchronization DBus API
resolved	Caching DNS resolver	timesyncd	Implements sNTP
networkd	Network configuration service	nspawn	Simple container runtime
homed	Management of home directories		

- systemd is dependency based execution engine
- Dependencies are relations
- Relations are defined on set of objects
- Objects that systemd manages are called "units"

# Unit types

- service
- target
- socket
- mount
- automount
- swap
- device
- path
- timer
- slice
- scope

See `man systemd.service`, `systemd.socket`, ..., for more information.

- systemd's units abstract system entities (resources)
- Units are created from various sources
- For example, mount unit may exist because administrator mounted a filesystem
- Most of the time however, units we deal with (services, sockets) exist because there is config file of the same name
- Unit files are simple text files in `.ini` format

# Unit file – example

```
# /usr/lib/systemd/system/cups.service
[Unit]
Description=CUPS Scheduler
Documentation=man:cupsd(8)
After=network.target

[Service]
ExecStart=/usr/sbin/cupsd -l
Type=notify

[Install]
Also=cups.socket cups.path
WantedBy=printer.target
```

# Unit files – Hierarchy of configuration

systemd loads unit files from following directories<sup>1</sup>,

- ❶ `/etc/systemd/system` – Owned by administrator
- ❷ `/run/systemd/system` – Runtime configuration, i.e. affects only single boot
- ❸ `/usr/lib/systemd/system` – Configuration shipped by the distribution

When there are two configuration files with the same name then systemd will load only one from the directory that is highest in the hierarchy. For example, configuration in `/etc` always overrides configuration in `/usr`.

After changing configuration it is necessary to reload systemd, `systemctl daemon-reload`

---

<sup>1</sup>`systemd-analyze unit-paths`

# Difference between unit and unit file

- This aspect of systemd is often confusing to new users
- It is important to recognize that there is a difference between units and unit files
- Mostly because SysVinit didn't track any service state and hence it didn't have this concept

# Dependency model in systemd

- Dependencies are very important concept to understand in order to be effective while working with systemd
- In the previous part of the tutorial we talked about units and unit files. Units are objects managed by systemd
- Dependencies are associations between them
- Each unit type has some default dependencies (unless configured otherwise)
- What types of dependencies there are,
  - Relational dependencies
  - Ordering dependencies

# Relational dependencies

- Wants – a unit should be started alongside with wanted unit
- Requires – a unit should be started alongside with required unit and if start of required unit fails then stop the former unit
- BindsTo – lifetime of two units is bound together (stronger than Requires)
- Requisite – requisitioned unit must be started already
- PartOf – dependency that propagates stop and restart actions
- Conflicts – “negative” dependency, i.e. conflicting units can’t run at the same time

# Ordering dependencies

Names of relational dependencies sort of suggest ordering, but don't be fooled. Ordering between units is undefined unless explicitly specified. Naturally, systemd provides two types of ordering dependencies,

- After
- Before

It is important to realize that ordering and relational dependencies are orthogonal and you can use ordering dependencies without defining any other relations between units.

- systemd also implements very minimal transaction logic.
- Every request (e.g. start or stop of a unit) is evaluated as a single transaction.
- systemd puts together transactions containing **job** objects (actions).
- systemd tries to carry out minimum amount of work/jobs.
- We examine a high-level overview of the transaction logic on the next slide.

# Transactions

- 1 Create job for the specified unit (anchor)
- 2 Add recursively jobs for all dependencies
- 3 Minimize the transaction in a loop
- 4
  - 1 Get rid of NOP jobs
  - 2 Get rid of jobs not referenced by anchor
- 5
  - 1 Check for ordering loops in the graph in a loop
  - 2 Break the loop by deleting a job
- 6 Get rid of jobs not referenced by anchor
- 7 Merge merge-able jobs
- 8 Get rid of jobs not referenced by anchor
- 9 Merge jobs with similar one already in job queue
- 10 Add the jobs to job queue

# Interesting options related to dependencies

- `DefaultDependencies` – Don't add default deps. of a given unit type
- `CollectMode` – Influence garbage collection logic (inactive-or-failed)
- `systemctl list-jobs --after/--before`

## Service management – Basics

- Start the service  
`systemctl start httpd.service2s`
- Stop the service  
`systemctl stop httpd.service`
- Restart service  
`systemctl restart httpd.service`
- Reload service  
`systemctl reload httpd.service`
- Send user defined signal to the service  
`systemctl --signal=SIGUSR1 kill httpd.service`

---

<sup>2</sup>You don't actually need to type `.service`, because `service` is default unit type 

## Service management – Managing unit files

- Enable service to start after a reboot,  
`systemctl enable httpd.service`
- Make service disabled, i.e. systemd won't attempt to start it after reboot,  
`systemctl disable httpd.service`
- Reset to default unit file state,  
`systemctl preset httpd.service`
- List all unit files,  
`systemctl list-unit-files`
- Determine current enablement state,  
`systemctl is-enabled httpd.service`
- Mask a unit file. Note that masked units can't be started, even when they are requested as dependencies,  
`systemctl mask httpd.service`

Notice that operations acting on unit files create or remove symlinks in the filesystem. To achieve the same end result you could create symlinks on your own.

## Service management – Unit file [Install] section

Let's consider this example [Install] section,

```
[Install]
```

```
WantedBy=multi-user.target
```

```
Also=sysstat-collect.timer
```

```
Also=sysstat-summary.timer
```

```
Alias=monitoring.service
```

What happens when we enable such unit file?

- systemd will enable `sysstat.service` in `multi-user.target` (runlevel 3)
- systemd will also enable `sysstat-collect.timer` and `sysstat-summary.timer` units according to their [Install] sections
- systemd will create alias `monitoring.service` and we will be able to use it in our follow-up work with the unit

## Service management – Extending unit files

- We already understand hierarchical nature of systemd's configuration
- **Configuration stored in /usr is overwritten on updates**
- There are multiple ways how to change or extend distribution supplied configuration,
  - One can copy configuration file from `/usr/lib/systemd/system` to `/etc/systemd/system` and edit it there
  - Or you can use configuration drop-ins. This is actually best practice
- In order to create drop-in, you need to do following,
  - 1 Create directory named after service but with `.d` suffix, e.g. `/etc/systemd/system/mariadb.service.d`
  - 2 Create configuration files in the directory. File should have `.conf` suffix
  - 3 Write part of the configuration that we want to add
- Drop-in configuration is shown in **status** output of the service
- Also configuration of systemd itself can be extended using drop-ins.

# Service management – Important unit files options

- **ExecStart** – Main service binary
- **ExecStop** – Stop command (must have synchronous behavior)
- **ExecReload** – Governs how to reload service (restart  $\neq$  reload)
- **KillMode** – Which processes get killed
- **Type** – Tells systemd how to treat service start-up
- **Restart** – Whether to restart always or only on certain events
- **PIDFile** – Relevant only for forking services. Nevertheless, very important
- **RemainAfterExit** – Used to implement idem-potency for oneshot services
- **StandardInput** – Allows you make socket a **stdin** of the service

# Service management – Service types

Type of the service determines when systemd assumes that service is started and ready to serve clients,

- **simple** – Basic (default) type. Service is considered running immediately after `fork()`
- **exec** – Service is considered running after successful execution of the service binary.
- **oneshot** – As name implies this type is used for short running services (systemd blocks until oneshot finishes)
- **forking** – Traditional UNIX double forking daemons
- **notify** – Service itself informs systemd that it finished startup
- **dbus** – Service considered up once bus name appears on system bus
- **idle** – Similar to simple, but service is started only after all other jobs were dispatched

- PART II: Resource management and workload isolation

Control groups (cgroups) is a Linux subsystem that has two main purposes,

- Process tracking
- Resource distribution

- **Cgroup** – associates a set of tasks with a set of parameters for one or more controllers.
- **Controller** – entity that schedules a resource or applies per-cgroup limits
- **Hierarchy** – Set of cgroups arranged in a tree, such that every process is in exactly one of the cgroups

# Resource management – cgroup v1 and cgroup v2

- **Multiple hierarchies** – cgroup v1 is a legacy kernel interface of the cgroup subsystem. Main difference between cgroup v1 and v2 is in the number of hierarchies. With cgroup v1 each controller is usually mounted separately, e.g. `/sys/fs/cgroup/memory`, `/sys/fs/cgroup/systemd`, `/sys/fs/cgroup/cpu`, `cpuacct`.
- **No processes in internal nodes** – cgroup v2 requires processes to reside only in the leaf nodes of the hierarchy for the purposes of resource distribution.
- **Delegation** – Parts of the cgroup tree can be delegated to less privileged users (via granting write access to cgroup interface files, `cgroup.procs`, `cgroup.threads` and `cgroup.subtree_control`) or via cgroup namespace and `nsdelegate` mount option.
- **Single writer** – In cgroup v2 the resource distribution *should* be governed by the single entity (preferably `systemd`) in non-delegated parts of the cgroup tree.

- **Weights**

- Resource is distributed by adding up the weights of all sub-cgroups and giving each the fraction matching its ratio against the sum.
- Usually used to distribute stateless resources (CPU time)
- Example: `cpu.weight` ([1-10000], default 100)

- **Limits**

- Cgroup can consume up to configured amount of the resource
- Overcommit is allowed (i.e. sum of sub-cgroup limits can exceed limit of the parent cgroup)
- Example: `memory.max`

- **Protections**

- Cgroup is protected (but not guaranteed) upto configured amount of the resource
- Overcommit is also allowed
- Example: `memory.low`

- **Allocations**

- Exclusive allocations of the absolute amount of a finite resource
- Overcommit is not allowed
- Example: `cpu.rt.max` (real-time budget)

systemd uses cgroups heavily, however it doesn't bother user with low-level cgroup interfaces. Instead it provides following high-level concepts,

- **Service** – Normal service units. Each service has its own cgroup.
- **Scope** – Similarly to services, scope's processes are also part of the cgroup. However, scope processes are not children of systemd
- **Slice** – Services and scopes can be further partitioned into slices.

To get an overview of current cgroup hierarchy on your system, you can run `systemd-cgls` command.

# Resource management – Control groups hierarchy

Control group /:

-.slice

```
├─user.slice
│   └─user-0.slice
│       ├──session-6.scope
│       │   ├──27 login -- root
│       │   ├──34 -bash
│       │   ├──52 systemd-cgls
│       │   └─53 systemd-cgls
│       └─user@0.service
│           └─init.scope
│               ├──28 /usr/lib/systemd/systemd --user
│               └─29 (sd-pam)
├─init.scope
│   └─1 /usr/lib/systemd/systemd
└─system.slice
    ├──dbus.service
    │   └─23 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile
    ├──systemd-logind.service
    │   └─22 /usr/lib/systemd/systemd-logind
    ├──systemd-resolved.service
    │   └─21 /usr/lib/systemd/systemd-resolved
    └─systemd-journald.service
        └─15 /usr/lib/systemd/systemd-journald
```

CPU controller in cgroup v2 has multiple configuration options for controlling how much CPU time is allocated to processes in cgroup. systemd provides configuration to adjust,

- **CPUWeight** – Set the value of "cpu.weight" cgroup property
- **CPUQuota** – Absolute value of CPU usage in percent

Note that default value of CPUWeight for every service is 100.

*All cgroup related options must appear in [Service] section of the unit file.*

# Resource management – Memory

Partitioning available memory with systemd and cgroup v2 memory controller is rather complicated. Multiple options are available,

- **MemoryMin** – Hard memory protection. If memory usage is below the limit the cg memory won't be reclaimed.
- **MemoryLow** – Soft memory protection. If memory usage is below the limit the cg memory can be reclaimed only if there is no memory to be reclaimed from unprotected cgroups.
- **MemoryHigh** – Memory throttle limit. If memory usage goes above the limit the processes in the cgroup are throttled and put under heavy reclaim pressure.
- **MemoryMax** – Hard limit for memory usage. You can use K, M, G, T suffixes (e.g. MemoryMax=1G).
- **MemorySwapMax** – Hard limit on swap usage.

After you exhaust your memory limit then service is very likely to get killed by OOM killer. To prevent that you need to adjust `OOMScoreAdjust` value as well.

Block I/O controller in cgroup v2 allows for quite fine grained tuning. systemd provides following options for configuring this subsystem,

- **IOWeight** – Set the default IO weight
- **IODeviceWeight** – Set the IO weight for a specific block device (e.g. `IODeviceWeight=/dev/sda 200`)
- **IOReadBandwidthMax, IOWriteBandwidthMax** – Absolute per device (or mount point) bandwidth. E.g. `IOWriteBandwidth=/var/log 5M`
- **IOReadIOPSMax, IOWriteIOPSMax** – Same as the above, except that bandwidth is configured in IOPS
- **IOLatency** – Define the per device I/O latency target (e.g. `IOLatency=/dev/sda 10ms`)

# Resource management – CPU and NUMA placement

- **CPUAffinity** – Set CPU Affinity mask for the unit.
  - **NUMAMask** – Set NUMA mask for the unit (e.g. NUMAMask=2, allow memory allocations only on NUMA node 2)
  - **NUMAPolicy** – Set NUMA memory allocation policy for the service (e.g. NUMAPolicy=bind)
  - **AllowedCPUs** – Restrict a unit to run only on selected CPUs.
  - **AllowedMemoryNodes** – Restrict a unit to be able to allocate memory only on selected NUMA nodes.
- Note that CPUAffinity, NUMAMask and NUMAPolicy can't be changed during the service runtime.

Using the pid cgroup controller you can limit number of processes that unit is allowed to spawn,

- **TasksMax** – Set the maximum number of processes that unit can create using `fork()` or `clone()`.

# Resource management – Dynamic reconfiguration

- It is trivial to set or adjust resource management configuration options at runtime.
- All of the previously discussed options are available via `systemd-run` or through D-Bus APIs
- `systemd-run` is a command line tool that allows you to run ad-hoc commands in `systemd` context
- Once you have the command wrapped as the unit you can use `systemctl set-property` to set resource management policies

# Resource management – Exercise: Database and low priority batch job

Propose a resource management policy expressed in terms of systemd unit file options that achieves following goals,

- Database gets more CPU time allocated over batch job
- Make sure that database is able to use up to 8GB of memory with incurring memory pressure
- Make sure batch job memory usage can't go over 1GB
- Set a restart policy on the database
- Decrease the chance of killing the database by OOM killer

# Resource management – Solution

- Increase CPUShares value for the important workload
- Set MemoryLow=8G memory protection for the main workload
- Set MemoryMax=1G limit for the batch job
- Restart=always
- OOMScoreAdjust=-900

## Resource management – Exercise: Critical workload

You have a mission critical workload running on the server and you want to make sure that it runs undisturbed whenever possible. Our goals are,

- Workload is running isolated on a subset of CPUs
- Workload can use all memory on NUMA nodes corresponding to those CPUs
- System services are allowed to consume only 1GB of system memory until memory reclaim pressure is applied

# Resource management – Solution

- `isolcpus` kernel command line argument
- Set `CPUAffinity=0` in `system.conf`
- `CPUAffinity` in the unit file to all other CPUs
- Set `NUMAMask=0` in `system.conf`
- `NUMAMask` set to remaining NUMA nodes for the workload
- `MemoryHigh=1GB` on `system.slice`

- PART III: Service sandboxing

# Sandboxing – Linux Namespaces

- Feature provided by Linux
- Used to virtualize various global system resources
  - mount
  - PID
  - user
  - uts
  - network
  - IPC
  - cgroup
- System calls used to manipulate namespaces,
  - clone
  - unshare
  - setns

# Sandboxing – Linux Namespaces

```
# ls -l /proc/self/ns
total 0
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 net -> 'net:[4026531969]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 pid -> 'pid:[4026531836]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 user -> 'user:[4026531837]'
lrwxrwxrwx. 1 root root 0 Nov  6 09:09 uts -> 'uts:[4026531838]'
```

# Sandboxing – Mount Namespace

- Virtualization of a filesystem view
- `unshare -m /bin/bash`
- Oldest namespace
- `clone(2)` argument `CLONE_NEWNS`
- Mount point propagation,
  - private
  - shared
  - slave
  - unchanged

# Sandboxing – PID Namespace

- Virtualization of process identifiers,
- `CLONE_NEWPID`
- `unshare -p --fork --mount-proc /bin/bash`
- `init` process in PID namespace
- Reaps zombie processes within namespaces
- Same signal handling exceptions applies as for real PID 1
- When `init` exits all other processes in a namespace get `SIGKILL` from kernel
- PID namespace of a process can't be changed
- It is possible to nest PID namespaces

# Sandboxing – User Namespace

- Virtualization of user and group databases and capabilities
- `unshare -U --map-root /bin/bash`
- Mapping of users between a container and a host system (created by writing to `/proc/[pid]/uid, gid_map`)
- User namespaces can be nested

# Sandboxing – Network Namespace

- `unshare -n /bin/bash`
- Virtualization of network related system resources,
  - Interfaces
  - IPv4 stack
  - IPv6 stack
  - Routing tables
  - Ports
- `veth` pair to create tunnel between namespaces

# Sandboxing – Other Kernel Namespaces

- IPC
  - Isolation of SystemV IPC resources and POSIX message queues
  - `unshare -i /bin/bash`
- UTS
  - Virtualization of hostname and NIS domain name
  - `unshare -u /bin/bash`
- Cgroup
  - Virtualization of a cgroup tree view
  - `unshare -C /bin/bash`

# Sandboxing

systemd provides a lot of options that help you further constrain and secure services running on your system. In most cases the only thing you need to do is to enable given feature in a unit file.

- **PrivateTmp** – Service has its own `/tmp` and `/var/tmp`
- **ProtectHome** – `/home`, `/root` and `/run/user` will appear empty
- **ProtectSystem** – Directories `/usr` and `/boot` are mounted read-only (if "full" also `/etc` is ro, on "strict" the entire filesystem is read-only)
- **ReadOnlyDirectories** – Service will have read-only access the listed directories
- **InaccessibleDirectories** – Listed directories will appear empty and will have 0000 access mode
- **RootDirectory** – Runs the service in `chroot()`-ed environment
- **PrivateDevices** – Service gets its own `/dev` with only basic device nodes, e.g `/dev/null`. `CAP_MKNOD` capability is disabled.

# Sandboxing

- **NoNewPrivileges** – Ensures that service can never gain new privileges
- **SystemCallFilter** – You can whitelist or blacklist allowed system call (note: `systemd-analyze syscall-filter [syscall-group]`)
- **PrivateNetwork** – Completely isolate service from network access (network namespace with only loopback)
- **JoinsNamespaceOf** – Enables multiple units to share `PrivateTmp` & `PrivateNetwork`
- **CapabilityBoundingSet** – List of capabilities to be included in the capability bounding set of the executed process
- **AmbientCapabilities** – List of capabilities to be included in ambient capability set
- **TemporaryFileSystem** – List of mount points where to mount `tmpfs`

# Sandboxing

- **PrivateUsers** – Run the service in its own user-namespace mapping root user to itself and everybody else to the "nobody"
- **ProtectKernelTunables** – Protect directories containing kernel runtime variables (e.g. /proc/sys, /sys)
- **ProtectKernelModules** – Disable the ability to load and unload the kernel modules
- **ProtectControlGroups** – Mount /sys/fs/cgroup read-only
- **RestrictAddressFamilies** – White-list address families (e.g. AF\_UNIX) that unit is allowed to use
- **RestrictNamespaces** – Limit access to namespace manipulation system calls (e.g. unshare, setns)
- **MemoryDenyWriteExecute** – Disable memory mapping that is simultaneously writable & executable
- **PrivateMounts** – Execute the service in its own mount namespace and turn off mount propagation towards the host's mount namespace

- **IPAccounting** – Ingress & egress IP traffic is counted for associated processes
- **IPAddressAllow** – List of allowed IP addresses that service can communicate with
- **IPAddressDeny** – IP deny list