Service Management with systemd

Michal Sekletár
msekleta@redhat.com

October 29, 2017

whoami

- Senior Software Engineer @ Red Hat
- systemd and udev maintainer
- Free/Open source software contributor

Administration

- 13:30 – Welcome and introduction
- 13:40 – Lab setup
- 14:00 – systemd basics
- 15:00 – Break
- 15:30 – Service management with systemd (resource management, journal)
- 16:50 – Review and final questions
- 17:00 – End of class

Agenda

- Introduction
- Main components of systemd
- Systemd units and unit files
- Dependency model
- Service administration with systemd
- Managing system resources with systemd
- Logging

Lab environment

- VM or container with Linux distribution[1] running systemd
- USB stick containing slides and lab manual
- Smoke test – become `root` and execute following commands,
  - `# ls -l /proc/1/exe`
  - `# systemctl status`
  - `# systemctl --type=service --state=failed list-units`

More detailed setup info is included in lab manual on page 2.

[1] Examples were tested on Fedora 26

What is systemd?

- Implementation of `init`, 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 (we even have the conference)

Components of systemd

systemd	`init`
udevd	Dynamic device management
journald	Log aggregator
logind	Session tracking
machined	VM/container registry
localed	DBus API for locale and language settings
hostnamed	Hostname setting
timedated	Time synchronization DBus API
timesyncd	Implements `sNTP`
nspawn	Simple container runtime
networkd	Network configuration service
systemd-sysctl	Applies dynamic kernel configuration
systemd-tmpfiles	Creation and cleanup of files and directories

Lab exercise – systemd in my distro

- Main idea behind this lab exercise is to discover how is systemd packaged and delivered on your Linux distribution of choice
- Note that lab manual was written using Fedora 26, hence you may need to use different commands to interact with package management system (`apt-get`, `dpkg`, `...`)
- To follow along please open lab manuals on page 3

Units

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

Unit types

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

Unit files

- 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,

1. `/etc/systemd/system` – Owned by administrator
2. `/run/systemd/system` – Runtime configuration, i.e. affects only single boot
3. `/usr/lib/systemd/system` – Configuration shipped by the distribution[2]

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`

[2] `/lib/systemd/system` on Ubuntu and Debian

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

Recap of the basics

- systemd is a service manager, but also a project that provides other basic user-space building blocks
- Notable components of systemd framework,
  - systemd
  - udevd
  - logind
  - journald
- systemd manages units and unit files
- Units abstract system resources
- Units may exists due to external events and can be instantiated from on disk configuration (unit files)
- Unit files are simple configuration files in `.ini` format understood by systemd

Lab exercise – systemd units

Open lab manual on page 8 and follow instructions.

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.

Transactions

- systemd also implements very minimal transaction logic.
- Every request (e.g. start or stop of a unit) is checked using the transaction logic. Once systemd puts together transactions it will check it and if possible it will create job objects that represent actions to be taken upon units. Once these actions are carried out, then user's request is complete.
- 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

Lab exercise – Dependencies and jobs

Useful command for working with jobs and dependencies,

- `systemctl list-dependencies <SERVICE>`
- `systemctl list-jobs`

Open lab manual on page 11 and follow instructions.

Service management – Basics

- Start the service
  `systemctl start httpd.service`[3]
- 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`

[3] You don't actually need to type .service, because service is default unit type

Service management – Configuration file

- Main configuration for systemd is read from `/etc/systemd/system.conf`
- Initially all values are commented out. They represents defaults
- You can configure some default timeout values which are then inherited by all other units

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 (we will examine this in the lab exercise)
- 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()
- `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

Service management – Service types

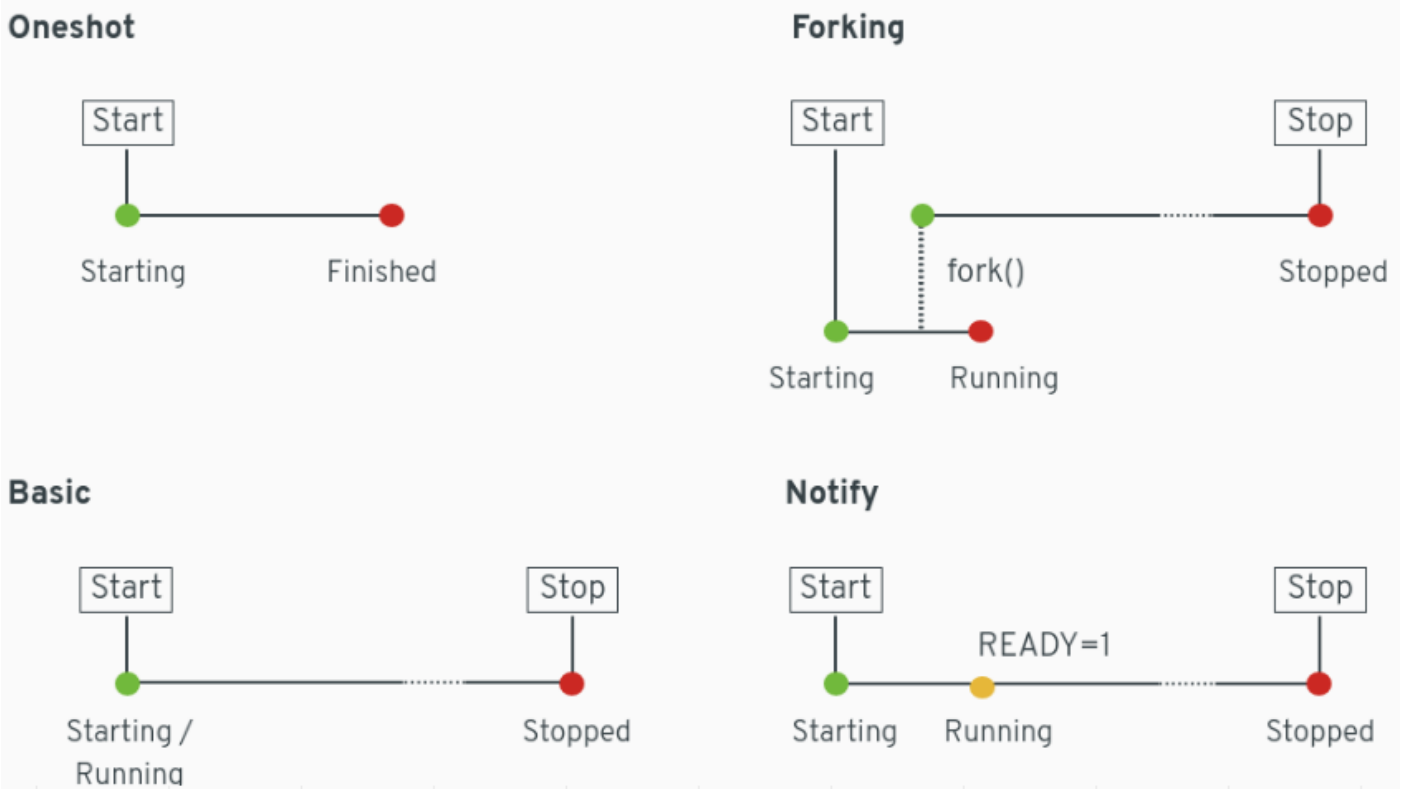

Figure: Effect of a service type on the service runtime state

Service management – Socket activation

So far we presented ways how to manually start or stop the service. However, systemd provides multiple other ways how to start or stop services, most notably socket and timer activation.

The idea behind socket activation is actually very simple. systemd starts listening on a socket (most commonly IP, but other protocols are supported as well), and on first connection, it starts the service that is activated by the socket.

There are two basic types of socket activation,

- Service is passed-in already `accept()`-ed socket file descriptor (service template must exist)
- Or service is passed in `listen()`-ing file descriptor

First type of socket activation exists to support legacy `(x)inetd` type of services.

Service management – Socket activation example

```
# /etc/systemd/system/foobar.socket
[Socket]
# accepted socket is passed to the service instance
Accept=true
ListenStream=127.0.0.1:5000

# /etc/systemd/system/foobar@.service
[Service]
ExecStart=/bin/bash -c "echo Activated by $REMOTE_ADDR"
```

Service management – Timer activation

Activation governed by calendar time or elapsed time interval works very similarly to socket activation. In this case unit activation is triggered by timer event.

```
# /usr/lib/systemd/system/fstrim.timer
[Timer]
OnStartupSec=10min
OnCalendar=weekly
AccuracySec=1h
Persistent=true

# /usr/lib/systemd/system/fstrim.service
[Service]
Type=oneshot
ExecStart=/usr/sbin/fstrim -av
```

Service management – Targets

- Target is a special unit type used for grouping other units
- Usually other units are enabled "into" targets. Meaning that when we start target all units enabled in that target will be started
- Target also serve as main synchronization points during system boot, i.e. they have ordering dependencies
- All units enabled into a target have an implicit `Before` ordering dependency on the target (unless explicitly specified otherwise)

Service management – Targets and runlevels

Concept of target units is used to implement runlevels as defined by SysVinit[4]

Runlevel	systemd target	Description
0	poweroff.target	System halt
1	rescue.target	Single user mode
3 (2,4)	multi-user.target	Multi-user (non-graphical)
5	graphical.target	Multi-user (graphical)
6	reboot.target	System reboot

Table: Runlevel to target mapping

[4] Runlevel 2, 3 and 4 are mapped to same target

Service management – Important target units

Awareness about these targets may come in handy when debugging bootup[5] issues,

- **local-fs.target** – All local file systems are mounted before this target is reached
- **swap.target** – Swaps are activated before this target
- **cryptsetup.target** – LUKS volumes are decrypted before target is reached
- **sysinit.target** – Implements minimal system initialization (pulls in previous targets)
- **timers.target** – Starts all timers
- **sockets.target** – Activates all socket units
- **paths.target** – Triggers all path units
- **remote-fs.target** – Remote filesystems (NFS, gluster, ceph, _netdev) are ordered before this target
- **basic.target** – All important subsystems has been initialized

[5] Details in `man 7 bootup` and `man 7 systemd.special`

Service management – Control groups

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

- Process tracking
- Resource management

Current state of this subsystem is somewhat confusing because we now have two different versions of cgroups. Depending on your systemd version and kernel configuration, you are maybe running cgroups-v1 or cgroups-v2 or both. For purposes of this tutorial, I will discuss cgroup-v1, because cgroup-v2 is not available in any stable distribution.

Good news is that because you are using systemd you **generally don't need to care**.

Service management – Control groups - terminology

- **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

Service management – Control groups

- Now we will examine our cgroup configuration in more detail. Let's see what controllers are supported on the system,

```
tail -n+2 /proc/cgroups | awk '{print $1}'
```

- Each controller is represented to user-space as cgroupfs mount point with specific options,

```
mount | grep cgroup
```

- As you can see we have all controller mounted in distinct paths and hence we have orthogonal hierarchies
- We also see one named hierarchy (name=systemd). This hierarchy is used for process tracking purposes.

Service management – Control groups and systemd

systemd uses cgroups very heavily, however it doesn't bother user with rather clunky 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 foreign (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.

Service 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
```

Service management – Resource management - CPU

CPU controller in cgroup-v1 has multiple configuration options for controlling how much CPU time is allocated to processes in cgroup. systemd provides API to adjust,

- **CPUShares** – Conceptually you can think of the CPUShare value as a weight. In other words, if I give one service some value (no matter what value actually is) and I give other service twice that, then second service will have twice as much CPU time if there is CPU contention on the system.
- **CPUQuota** – Absolute value of CPU usage in percent.

Note that default value of CPUShares for every service is 1024.

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

Service management – Resource management - Memory

Partitioning available memory with systemd and cgroup-v1 memory controller is rather simple. Only one option is available,

- **MemoryLimit** – Hard limit for memory usage. You can use K, M, G, T suffixes. E.g. MemoryLimit=1G

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.

Service management – Resource management - Block I/O

Block I/O controller in cgroup-v1 allows for quite fine grained tuning. systemd provides following options for configuring this subsystem,

- **BlockIOWeight** – Assigns an IO weight to a specific service (requires CFQ)
- **BlockIODeviceWeight** – Can be defined per device (or mount point). Default value is 1000.
- **BlockIOReadBandwidth, BlockIOWriteBandwidth** – Absolute per device (or mount point) bandwidth. E.g. `BlockIOWriteBandwith=/var/log 5M`

Service management – Securing your services

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`
- **PrivateNetwork** – Completely isolate service from network access (network namespace with only loopback)
- **SELinuxContex** – Run service binary with explicit SELinux context
- **ProtectHome** – `/home`, `/root` and `/run/user` will appear empty
- **ProtectSystem** – Directories `/usr` and `/boot` are mounted read-only (if "full" also `/etc` is ro)
- **ReadOnlyDirectories** – Service will have read-only access the listed directories
- **InaccessibleDirectories** – Listed directories will appear empty and will have 0000 access mode

Service management – Securing your services

- **PrivateDevices** – Service gets its own /dev with only basic device nodes, e.g `/dev/null`. CAP_MKNOD capability is disabled.
- **LimitNPROC** – Defines maximum number of processes that comprise the service
- **CapabilityBoundingSet** – You can specify which capabilities will service retain
- **NoNewPrivileges** – Ensures that service can never gain new privileges
- **SystemCallFilter** – You can whitelist or blacklist allowed system call (use sec-comp)
- **RootDirectory** – Runs the service in chroot()-ed environment

Lab exercise – Service management

Open lab manual on page 14 and follow instructions.

systemd-journald

As previously mentioned systemd is very tightly integrated with journald logging service. Journald brings a lot of innovation and advantages over traditional syslog.

- Structured logs
- Log meta-data
- Rich filtering capabilities
- Indexed
- Security
- Reliability (logs from early boot to late shutdown)
- Intelligently rotated (based on available disk space)

Journald can be configured to store logs,

- Persistently
- In memory (available until system reboots)

Main configuration file is stored in `/etc/systemd/journald.conf`

systemd-journald - Log aggregation

Journald is a central place where all system logs eventually end up. It gathers and stores data from various sources,

- `/dev/log` – standard `syslog()` socket
- `/run/systemd/journal/socket` – socket used by native journald clients (i.e. those using sd-journal)
- `/dev/kmsg` – journald also stores kernel log messages
- `NETLINK_AUDIT` – optionally journal can also store audit logs

The advantage of putting all logs in a central place is a presentation to end user. We can conveniently display log messages from all sources interleaved together, ordered by time.

Lab exercise – journald

Open lab manual on page 18 and follow instructions.

Resources

- Upstream web page
- Upstream issue tracker
- How to debug systemd issues
- Red Hat documentation

Thank you!

Please remember to complete your tutorial evaluation.