Chapter W27 Advanced Applications of systemd

Objectives:

\* Expand the coverage of systemd “superkernel” topics, sub-commands, and applications

\* Expose and emphasize the importance of systemd units and unit files

\* Dissect the anatomy of a unit file, with a specific example

\* Show how to create instance units from unit template files

\* Describe systemd targets, their variety, and their utility

\* Give a specific systemd target example – running a clock-time-based script

\* Add to the repertoire of unit management commands

\* Provide much additional practice with target units to the novice user

\* Illustrate how to switch the system between important target states

\* Expand our coverage of Linux cgroups, in the context of a systemd environment

\* Briefly mention the Linux kernel namespace concept

\* Define and expound upon the systemd journal daemon, journald

\* How to view system logs with the journalctl command

\* How to maintain the journal, and execute varieties of boot process querying

\* Give further examples of systemd-controlled timers

\* Illustrate the activation behavior of “new-style” daemons

\* Depict a graphical user interface to systemd, known as systemadm

To cover the Commands and primitives-

**journalctl, ncat, systemctl**

W27.1 Introduction

Chapter 18 in the printed book gave a novice user a brief, but useful, introduction to the Linux “superkernel” known as systemd. Unless that novice user, or even a more seasoned system professional, has not only a basic, but also a more complete knowledge of how systemd controls and oversees every process operation of a modern Linux system, they will never be able to master administrating and implementing the kind of functionality that their use case(s) might ultimately require. Particularly for the user base on the system, and the demands that user base makes. This chapter adds more flesh onto the skeletal framework presented in Chapter 18.

Everything illustrated in this chapter, in the specific form (and the syntax of commands) found here, is explicitly applicable to all of our representative Linux systems- Debian-family distributions and versions of Debian, Ubuntu, and Linux Mint, and RedHat-family CentOS. We encourage you to use one of these seminal, mainline distributions. As stated in the Preface to the printed book, one of the major areas of future development of Linux will be the expansion of the role that systemd plays in every aspect of Linux operating system use.

The numbering of the Sections, and Examples in this chapter, proceeds in a mostly-sequential order as they have been selectively extracted from, and cross referenced to, the Sections and Examples in Chapter 18 of the printed book. This selective extraction is done according to the pedagogic needs of the presentation of this advanced topic chapter. They are therefore not numbered here in this chapter in strictly cardinal order. The In-Chapter Exercise numbering is strictly sequential and in cardinal order.

Anything that you are required to type on the command line is show in **bold** type.

W27.2.3 Bootup in the Initial RAM Disk (initrd)

The initial RAM disk implementation (initrd) can also be set up using systemd, and follows a prescribed structure. initrd is a scheme for loading a root file system, from a variety of possible sources, into memory, and which is used as part of the Linux startup process.

W27.2.4 Querying the Boot Process

As shown in Section W27.6.2.6 Boot Process Querying, the systemd journald daemon and journal give you the ability to view log records of how the system boots. In particular, it allows you look at a log record of the current boot process, and past boot processes, with some specific command line tools, options, and their arguments . We refer you to Section W27.6.2.6 for a more complete treatment of the systemd journal, and its capabilities with respect to booting logs.

W27.3 systemd Units and Unit Files

As can be seen in Chapter 18 of the printed book, systemd unit files are one of the most critical and ubiquitous features of systemd that a beginner needs to understand. In this section, we discuss more of the lower level functionality and application of systemd units.

W27.3.1 Introduction

In systemd, a unit refers to an object that changes the characteristics of the steady state of the Linux system, or its normal operating condition. Units, and unit files, are the primary objects that systemd creates and manipulates, with user commands such as systemctl. These objects are configured with files called unit files. We will introduce you to the different units that systemd can handle. We will also be covering some of the many “directives” that can be put in unit files in order to configure the way these objects handle resources on your system.

W27.3.2 Roles systemd Units Play

Units and unit files take on a very standardized format, which we describe in detail beccause they are the primary instrument of user functionality and control in Linux systemd. They enable you to manage system resources using the daemons, services, and utility commands shown in Figure 18.1 of the printed book. They are found schematically on the first two top levels of that figure. We provide that figure for your reference as follows-

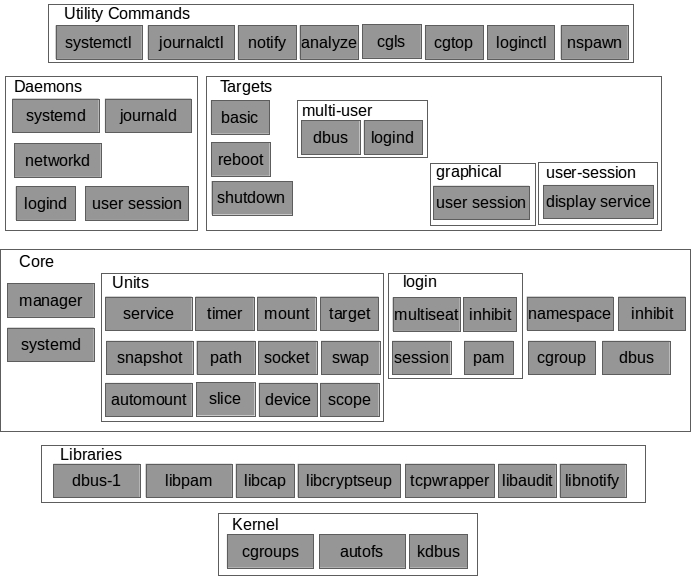


Figure 18.1 systemd Architecture

A unit works with and affects traditional system services, network resources, devices, filesystem mounts, resource pools known as Control Groups, and even very transient tasks such as single Linux commands. We gave a good example of this transient scope use in Chapter 17, Section 8.1.2 in the printed book.

Some of the key advantageous features that units supply are:

1. socket-based activation: The delay of the the start of a service until activation of its socket(s). Very crucial to make the system run faster, and startup much faster.

2. D-bus-based activation: Units can be started when an associated bus is published.

3. path-based activation: A unit can be started based on activity on or the availability of certain filesystem paths.

4. udev device-based activation: Start of Units can also be delayed until the first time hardware becomes available. Very crucial to make the system run faster, and startup much faster.

5. implicit dependency mapping: Most of the dependence between units, in their start time, can be built by systemd itself, although by editing the dependencies, this can be modified by the user.

6. instances and templates: Template unit files can be used to create multiple instances of the same general unit. Very critical and efficient in container virtualization!

7. security: Units can implement security features, via the use of directives in the unit file.

8. drop-ins and snippets extensibility: Units can easily be extended by providing snippets that will override parts of the system's unit file.

We address many of these features in the following sections.

W27.3.3 Unit File Locations in the Filesystem, and Editing or Modifying Them

Unit files are located in basically three different locations, depending upon how important the unit file is.

Table W27.1 lists these default locations and a brief description of their utility.

|  |  |
| --- | --- |
| /etc/systemd/system | Place where unit files used to override default ones are stored. |
| /run/systemd/system | Middle-priority unit file location, the systemd process itself uses this location for dynamically created unit files created at runtime. |
| /lib/systemd/system | System copy where software using the unit file is installed. Also locates the default controlling unit file for the software. |

Table W27.1 Unit File Locations

Adding, or modifying a unit by editing it to modify the way that the unit functions, should be done by creating it in the /etc/systemd/system directory.

Unit files found in this directory location take precedence over any of the other locations.

If you need to modify the system's copy of a unit file, putting a replacement in this directory is the most reliable, safest, and flexible way to do this.

To override specific directives from the system's generic unit file, you create unit file “snippets”, i.e. smaller versions, within a very standard and specific subdirectory. This will add to or modify the directives of the system's copy, allowing you to specify only the behavior of the unit you want to change.

Directives are what specify explicitly how the unit behaves.

To do this, you create a directory named after the unit file with .d appended to the name. For a unit called example.service, you create a subdirectory called example.service.d . Within this directory a file ending with .conf can be used to override or add to the directives of the system's unit file.

W27.3.3.1 Editing Unit Files

The procedure of editing unit files is only second in importance to actually creating a unit file from scratch for a service that you want to create and run on the system. We show how to create a unit file below. The premier systemd command, systemctl, provides options for editing and modifying unit files if you need to make adjustments. The systemctl edit command, by default, will open a unit file snippet for the unit in question.

For example, to further customize the nginx webserver service, which has a unit file automatically created by the Aptitude package manager when it is installed on Linux Mint, use the following command:

$ **sudo systemctl edit nginx.service**

This will be a blank file that can be used to override or add directives on top of the existing service unit definition. A directory will be created within the /etc/systemd/system directory which contains the name of the unit with .d appended. For instance, for the nginx.service, a directory called nginx.service.d will be created.

Within this directory, a snippet will be created called override.conf. When the unit is loaded, systemd will merge the override snippet with the full unit file. The snippet's directives will take precedence over those found in the original unit file.

To edit the full unit file, instead of creating a snippet, use the --full option:

$ **sudo systemctl edit --full nginx.service**

This will load the current unit file into the editor, where it can be changed. When you exit the editor, the changed file will be written to /etc/systemd/system, which will take precedence over the system's unit definition found in /lib/systemd/system.

To remove any additions you have made, either delete the unit's .d configuration directory, or the modified service file from /etc/systemd/system. To remove a snippet, use the following command:

$ **sudo rm -r /etc/systemd/system/nginx.service.d**

To remove a full modified unit file, we would type:

$ **sudo rm /etc/systemd/system/nginx.service**

After deleting the file or directory, you should reload the systemd process so that it no longer attempts to reference these files and reverts back to using the system default copies. You can do this with the following very ubiquitous and critical command:

$ **sudo systemctl daemon-reload**

In-Chapter Exercise

1. When you installed the nginx Webserver, as required in Chapter 17, Exercise 17.26 of the printed book, where did the installer locate the default copy of the service unit file? How did you find this out? If you haven’t done the installation of nginx as requested in Chapter 17, either do that now, or answer this exercise with another service that you have installed.

W27.3.4 Types of Units

systemd units, and the files that define them, are the primary instruments that systemd uses to keep in control of system state . If you look at the suffix attached to a unit file, you can determine which of the twelve types of unit it is. The following list describes the types of units available to systemd:

1. .service: A service unit, which we detail most extensively below, describes how to manage a service or application on the system. This will include important things like what the path is to the executable code that defines the application, the conditions affecting the starting or stopping of the service, under which circumstances it should be automatically started, and the dependencies and order-of-starting information for the software that is the core of the service. These are also a crucial cgroup category, as we detail in Section W27.6.1.

2. .socket: A socket unit file defines a network, IPC socket, or an FIFO buffer, that systemd uses for a socket-based activation procedure. We detail this below in Section . In socket-based activation, socket units are used to start services only when the socket the service is attached to is addressed. This allows the starting of services in parallel. It has a .service file, that will be started when connections are made to the socket that this unit defines.

3. .device: A unit that describes a device that requires systemd management by udev or the sysfs filesystem. udev is a device manager for the Linux kernel, and manages device nodes in the /dev directory. udev also handles all user space events and hardware devices that are added into the system or removed from it.

4. .mount: Defines a mountpoint on the system to be managed by systemd. These are named after the mount path, with slashes changed to dashes.

5. .automount: An .automount unit defines a mountpoint that will be automatically mounted. These must be named after the mount point they refer to and must have a matching .mount unit to define the specifics of the mount.

6. .swap: This unit defines swap space on the system. The name of these units is derived from the device name or file pathname of the space.

7. .target: A target unit provides a way of coordinating for other units when the system starts up, or more importantly when there are changes in system state. A good example, that we describe in Section 5.1.8.5, is going from the multi-user.target state to the graphical.target state, and vice versa. We cover much of what can be done with targets in this chapter. Other units define their relation to targets to become synchronized to the target's operations.

8. .path: This unit defines a path that can be used for path-based activation.

Path-based activation means you can make the operating system take actions if a certain file, group of files, or directory get modified in any way.

9. .timer: A .timer unit, which we illustrate in detail in Section W27.6.3, defines a timing control that systemd uses when controlling system state.

10. .snapshot: A .snapshot unit is created automatically by the systemctl snapshot command. Although it may not really be considered a unit in the strict sense, since there is no unit file associated with snapshots, it is still a managed system entity. It allows you to reconstruct certain aspects of the current state of the system after making changes. Snapshots do not survive across reboot or restart, and are used primarily to roll back to temporary states during the current boot environment.

11. .slice: A .slice unit is defined by Linux cgroup nodes, and allows resources to be delegated to any processes associated with the named slice. These are also a crucial cgroup category, as we detail in Section W27.6.1.

12. .scope: Scope units are created automatically by systemd, using information from its bus interfaces.

These are also a crucial cgroup category, as we detail in Section W27.6.1.

We will mainly be focusing on .service unit files, and their modification and creation for the ordinary user of the system. This is due because they are very useful for an ordinary user, and the use cases she might put the system to. We will also detail, to some extent, their use by an appointed system administrators to manage the state of the system in general. Aside from .service unit files, all of the other unit file types are coordinating and synchronization tools, that tie services to hardware, cgroups, IPC sockets, timing constraints, etc.. This coordination and synchronization is perhaps the hallmark of systemd, and its major advantage over the older Linux init system(s).

W27.3.5 Anatomy of a Unit File

The internal structure of unit files are organized with sections. Sections are denoted by a pair of square brackets "[" and "]" with the section name enclosed within. Each section extends until the beginning of the subsequent section or until the end of the file.

W27.3.5.1 An Example Service Unit File - sshd

It would be very instructive at this point to show the structure, and the exact contents of a typical service unit file. Figure W27.4 shows the contents of the sshd unit file.

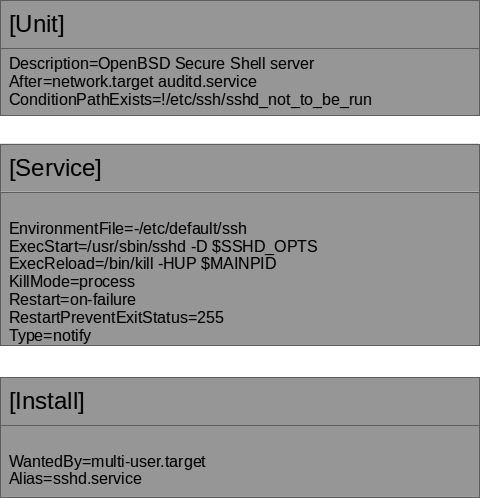


Figure W27.4 Unit File Example

W27.3.5.2 The General Format of Unit Files

As seen in Figure W27.4, the file is divided into sections. Section names are case-sensitive, and enclosed in square brackets [ ]. Within each section, the way the unit works and stores its characteristic data is done using standard, simple directives with allowable values assigned as follows-

[Section]

Directive=value

Directive=value

If you use an override file, as is done when editing a unit file in Section W27.3.3, directives are reset by assigning them to an empty string or null string. For example, the system's copy of a unit file may contain a directive set to a value like this:

Directive=default\_value

The default\_value can be eliminated in an override file by referencing the directive without a value, like this:

Directive=

W27.3.5.3 sshd Example Unit, Service, and Install Sections Directives

This section details the specific directives for the three sections of the sshd example unit file shown in Figure W27.4.

[Unit] Section Directives

The first section found in the sshd.service unit file is the [Unit] section. This is generally used for defining metadata for the unit and configuring the relationship of the unit to other units.

Although section order does not matter to systemd when reading the file, this section is traditionally placed at the top because it provides an overview of the unit. The particular directives that are in the [Unit] section are:

1. Description=OpenBSD Secure Shell server

This directive can be used to describe the name and basic functionality of the unit. It is returned by various systemd tools, so it is good to set this to something short, specific, and informative.

2. After=network.target auditd.service

The units listed in this directive will be started before starting the current unit. This does not imply a dependency relationship and one must be established through the above directives if this is required.

3. ConditionPathExists=!/etc/ssh/sshd\_not\_to\_be\_run

There are a number of directives that start with Condition which allow the administrator to test certain conditions prior to starting the unit. This can be used to provide a generic unit file that will only be run when on appropriate systems. If the condition is not met, the unit is gracefully skipped.

[Service] Section Directives

The [Service] section is used to provide a configuration that is only applicable for services.

One of the basic things that should be specified within the [Service] section is the Type= of the service. This categorizes services by their process and behavior as a daemon. This is important because it tells systemd how to correctly manage the service and find out its state.

1. EnvironmentFile=-/etc/default/ssh

Similar to Environment= but reads the environment variables from a text file. The text file should contain new-line-separated variable assignments. Empty lines, lines without an "=" separator, or lines starting with ; or # will be ignored, which may be used for commenting. A line ending with a backslash will be concatenated with the following one, allowing multi-line variable definitions. The parser strips leading and trailing whitespace from the values of assignments, unless you use double quotes (").

The argument passed should be an absolute filename or wildcard expression, optionally prefixed with "-", which indicates that if the file does not exist, it will not be read and no error or warning message is logged. This option may be specified more than once in which case all specified files are read. If the empty string is assigned to this option, the list of file to read is reset, all prior assignments have no effect.

2. ExecStart=/usr/bin/sshd –D $SSHD\_OPTS

Probably the most critical directive! It designates the full path and the possible arguments of the command to be executed to start the process. This may only be specified once (except for "oneshot" services).

3. ExecReload=/bin/kill –HUP $MAINPID

This optional directive indicates the command necessary to reload the configuration of the service if available.

4. KillMode=process

Specifies how processes of this unit shall be killed. One of control-group, process, mixed, none.

If set to control-group, all remaining processes in the control group of this unit will be killed on unit stop (for services: after the stop command is executed, as configured with ExecStop=). If set to process, only the main process itself is killed. If set to mixed, the SIGTERM signal (see below) is sent to the main process while the subsequent SIGKILL signal (see below) is sent to all remaining processes of the unit's control group. If set to none, no process is killed. In this case, only the stop command will be executed on unit stop, but no process be killed otherwise. Processes remaining alive after stop are left in their control group and the control group continues to exist after stop unless it is empty.

5. Restart=on-failure

This indicates the circumstances under which systemd will attempt to automatically restart the service. This can be set to values like "always", "on-success", "on-failure", "on-abnormal", "on-abort", or "on-watchdog". These will trigger a restart according to the way that the service was stopped.

6. RestartPreventExitStatus=255

Takes a list of exit status definitions that, when returned by the main service process, will prevent automatic service restarts, regardless of the restart setting configured with Restart=. Exit status definitions can either be numeric exit codes or termination signal names, and are separated by spaces. Defaults to the empty list, so that, by default, no exit status is excluded from the configured restart logic.

7. Type=notify

Configures the process start-up type for this service unit. Can be simple, forking, oneshot, dbus, notify or idle. notify is similar to simple; however, it is expected that the daemon sends a notification message via sd\_notify or an equivalent call when it has finished starting up. systemd will proceed with starting follow-up units after this notification message has been sent. If NotifyAccess= is not set, it will be implicitly set to main. Note that currently Type=notify will not work if used in combination with PrivateNetwork=yes.

[Install] Section Directives

On the opposite side of unit file, the last section found in the sshd.service unit file is the [Install] section. This section is optional and is used to define the behavior or a unit if it is enabled or disabled. Enabling a unit marks it to be automatically started at boot. In essence, this is accomplished by latching the unit in question onto another unit that is somewhere in the line of units to be started at boot.

Because of this, only units that can be enabled will have this section. The directives within dictate what should happen when the unit is enabled:

1. WantedBy=multi-user.target

The WantedBy= directive is the most common way to specify how a unit should be enabled. This directive allows you to specify a dependency relationship in a similar way to the Wants= directive does in the [Unit] section. The difference is that this directive is included in the ancillary unit allowing the primary unit listed to remain relatively clean. When a unit with this directive is enabled, a directory will be created within /etc/systemd/system named after the specified unit with .wants appended to the end. Within this, a symbolic link to the current unit will be created, creating the dependency. For instance, if the current unit has WantedBy=multi-user.target, a directory called multi-user.target.wants will be created within /etc/systemd/system (if not already available) and a symbolic link to the current unit will be placed within. Disabling this unit removes the link and removes the dependency relationship.

2. Alias=sshd.service

This directive allows the unit to be enabled under another name as well. Among other uses, this allows multiple providers of a function to be available, so that related units can look for any provider of the common aliased name.

W27.3.5.4 Additional Unit File Sections and Their Unit-Specific Section Directives

As can be seen in our sshd service unit file example in Section W27.3.5.1, the Service Section is found between the Unit Section and the Install Section.

In addition, in a more complex unit file, the following sections are found between the Unit and Install Sections. Note that of the 12 unit types, most contain directives that only apply to their specific type. Examples of unit types that do not contain unit-specific directives are the device, target, snapshot, and scope units.

The following brief listing provides references, contained in specific sections, for additional important values assigned to directives in those sections:

The [Socket] Section

Socket unit files are contained in systemd configurations because, as seen in many sections and problems we present below, many services implement socket-based activation to improve overall system parallelization and speed.

Note that each socket unit must have a matching service unit that will be activated when the socket receives activity.

Perhaps the hallmark of systemd control, the methods of socket control simplify and enhance an administrators ability to oversee everything that is happening on the computer. By default, the socket name will attempt to start the service of the same name upon receiving a connection. When the service is initialized, the socket will be passed to it, allowing it to begin processing any requests.

To specify the actual socket, the premier directive is ListenStream= ,which defines an address for a stream socket, and which supports socket IPC communication. Services that use TCP generally use this unit type.

The [Timer] Section

Timer units are used to schedule system events at some specific time, whether that time is measured on a clock or a calendar, or after a certain delay. This unit file type can replace or supplement the functions of the cron daemon, for example. An associated unit file, such as a service unit, must be created, to be activated when the timer is “goes off”.

The [Timer] section of a unit file most importantly contains some of the directive Unit= , which is used to specify the unit that should be activated when the timer signals the event.

W27.3.6 Creating Instance Units from Template Unit Files

For an ordinary user, template unit files let you reproduce a unit file several times, to get multiple copies of it (which you can slightly modify per copy). But, the primary commercial reason for using template unit files, and their mechanisms, is to run multiple containers on a server. For example, when you want to have many services listen on many sockets or ports at the same time, you could run multiple instances of the service as separate isolated environments, each with a slightly different name and different ephemeral port it is listening on. Of course, you must map these container’s ports to network-facing addresses. Possibly these could be public-facing network addresses assigned by a DHCP server, rather than private addresses, as is used primarily by container software such as Docker.

W27.3.6.1 Template and Instance Unit Names for Services

The name of a template unit file contains an @ symbol after the unit name, and before the unit type. A classic systemd example, found in /etc/systemd/system is the following:

serial-getty@ttyS2.service

This is the serial getty service instantiated for ttyS2.

When an instance is created from a template, a particular instance identifier is placed between the @ symbol and the period signifying the start of the unit type, as seen above.

In-Chapter Exercise

2. In [serial-getty@ttyS2.service](mailto:serial-getty@ttyS2.service) above, what is the name of the service, and what is the instance identifier? Why would you need different instances of the getty service running at the same time?

An instance file is created as a symbolic link to the template file, with the link name including the instance identifier. Multiple links with unique identifiers can point back to a single template file. When called upon to use an instance unit, systemd will look for a file with the exact instance name you specify on the command line. If that cannot be found, systemd will look for an associated template file.

W27.3.6.2 Common Template Specifications

Template unit files give you the ability to substitute appropriate directive information within the unit definition according to your run time needs. This is done by setting the directives in the template file as normal, but replacing certain values or parts of values with variable specifiers in the various instances. See the man page for systemd.unit for a complete listing of template specifiers.

W27.4 Targets

Targets, or more formally, target units, are the same as all other systemd unit types. They group units using dependencies (for state changes like booting or shutdown), and establish standard, or even user-defined names, for synchronization points used in dependencies between units.

As an example of their use, the processes of starting the system and shutting it down are maintained by systemd via the sequential, though not completely hard-wired, use of target units.

W27.4.1 Basic Target Concepts

systemd can meet booting, system service management, and ancillary function requirements for modern Linux systems by using standardized and named target units. For example, the target graphical.target provides a multiuser system with network connectivity and a Graphical Display Manager (GDM), such as MDM (Mint Display Manager on Linux Mint), or Lightdm. A very interdependent set of units with dependencies can group together controls on system state, and offers the user a way of defining customized system state controls.

The following table shows some critical systemd target units.

|  |  |
| --- | --- |
| default.target | The target that is booted by default. Not a real target, but a symbolic link to another target like graphical.target. |
| emergency.target | Starts an emergency shell on the console. Only used at the boot prompt as systemd.unit=emergency.target. |
| graphical.target | The default target in a desktop system GUI installation. Starts a system with network, multiuser support and a display manager. |
| halt.target | Shuts down the system. |
| multi-user.target | Starts a multiuser system with network. |
| reboot.target | Reboots the system. |
| rescue.target | Starts a single-user system without network. |

Table W27.2 systemd Important targets

Following are listings and descriptions of some other special system target units. For a full list refer to the man page on your system for systemd.special. Incidentally, these target states are the equivalents of “run levels” found in the older legacy Linux init system(s).

*basic.target*

A special target unit covering basic boot-up.

systemd automatically adds dependencies of the types Requires= and After= for

this target unit.

This pulls-in all local mount points plus /var, /tmp and /var/tmp, swap devices,

sockets, timers, path units and other basic initialization necessary for general purpose

daemons. It is also a synchronization point for late

boot services.

*ctrl-alt-del.target*

systemd starts this target whenever Control+Alt+Del is pressed on the console.

Aliased via a symbolic link to reboot.target.

*default.target*

The default unit systemd starts at bootup. Aliased via a symbolic link to

multi-user.target or graphical.target.

*emergency.target*

A special target unit that starts an emergency shell on the main console. Compare

with rescue.target, which is similar purpose, but also starts the most basic services

and mounts all file systems.

*exit.target*

A special service unit for shutting down the system or user service manager.

It is equivalent to poweroff.target on non-containerized systems, and also

works in containerized ones.

*final.target*

A special target unit that is used during system shutdown and may be used to pull in late

services after all normal services are already terminated and all mounts unmounted.

*graphical.target*

A special target unit for graphical login and operation, the default target on

all of our representative Linux systems that use a GUI-based desktop.

This requires multi-user.target as a dependency.

*halt.target*

A special target unit for shutting down and halting the system.

*kexec.target*

A special target unit for shutting down and rebooting the system via kexec.

*multi-user.target*

A special target unit for setting up a multi-user system with a text-only console interface.

Used in server use cse machines. This target is a ependency required by graphical.target.

Units that are needed for a multi-user system add Wants= dependencies for their unit to

this unit during installation, via WantedBy=multi-user.target in the

unit's "[Install]" section.

*poweroff.target*

A special target unit for shutting down and powering off the system.

Applications wanting to power off the system start this unit.

*reboot.target*

A special target unit for shutting down and rebooting the system.

Applications wanting to reboot the system should start this unit.

*rescue.target*

A special target unit that pulls in the base system (including system mounts) and spawns a

rescue shell.

*shutdown.target*

A special target unit that stops the services on system shutdown.

Services to be terminated on system shutdown add Conflicts= dependencies to this

unit for their service unit, which is done when DefaultDependencies=yes is set.

*slices.target*

A special target unit that sets up all slice units that are made active after booting.

*sockets.target*

A special target unit that sets up all socket units that are active after booting.

Services that can be socket-activated add Wants= dependencies to this unit for their

socket unit during installation. This is configured via a WantedBy=sockets.target in the

socket unit's "[Install]" section.

*sysinit.target*

This target pulls in the services required for system initialization.

*system-update.target*

A special target unit that is used for off-line system updates.

*timers.target*

A special target unit that sets up all timers that are active after booting.

*umount.target*

A special target unit that unmounts all mount and automount points on system shutdown.

W27.4.2 A Target Example: Clock-Time-Based Running of a Script File

Question: Why would you want to do this, as an ordinary user?

Answer: Because at some particular time of the day, you want the system to automatically backup critical directories and files on designated media, via some automating Bash or Python3 script file.

“Monotonic Scheduling” of events can be simply thought of as basing an event happening on the system on clock time. Clock time could be designated in minutes, hours, days, weeks, etc. The following example creates a timer unit which relies on a specific target unit to execute a script file on a daily basis. The service which needs to be run daily can be called in as a dependency of this target.

The script file, which is named your\_script, is made into a service, via the creation of a service unit file. your\_script can have arguments when it is run, as shown below. Also, you must set the permissions on your\_service, with **chmod 755**, so that it will execute properly when it is a service!

To get more information about how systemd specifies time, see the man page for systemd.time.

1. Use the following command to create a directory which will hold the calendar-based timer unit:

$ **sudo mkdir /etc/systemd/system/timer\_daily.target.wants**

2. The following timer unit file will need to be created, using your favorite text editor, in the path specified below:

$ **sudo emacs /etc/systemd/system/timer\_daily.timer**

**[Unit]**

**Description=Daily Timer for Events**

**[Timer]**

**OnBootSec=5min**

**OnUnitActiveSec=1d**

**Unit=timer\_daily.target**

**[Install]**

**WantedBy=basic.target**

3. Create the target unit, with your favorite text editor, in the specified directory:

$ **emacs /etc/systemd/system/timer\_daily.target**

**[Unit]**

**Description=Daily Timer Target**

**StopWhenUnneeded=yes**

4. Now that we have a timer unit and timer target file created, adding events to this target involves placing an event into the /etc/systemd/system/timer\_daily.target.wants folder.

For any particular event to take place daily, create a service unit file for the particular event in the /etc/systemd/system/timer\_daily.target.wants folder.

For example, if you wish to run your\_script.service daily (which for example, runs a Bash script named your\_script) , create the following file with your favorite text editor:

$ **sudo emacs /etc/systemd/system/timer\_daily.target.wants/your\_script.service**

**[Unit]**

**Description=Whatever your script file does**

**[Service]**

**User=bob**

**Type=Simple**

**Nice=19**

**IOSchedulingClass=2**

**IOSchedulingPriority=7**

**ExecStart=/home/bob/your\_script –arg1 –arg2**

Use more ExecStart lines in the above file if you want to start more than one event daily.

5. Start and enable the daily timer:

$ **sudo systemctl start timer\_daily.timer**

$ **sudo systemctl enable timer\_daily.timer**

In-Chapter Exercise

3. Using the 5 above steps, take a Bash script of your own, and have systemd execute it daily.

W27.3.1.7 Unit Management with Additional Commands

So far, we have been working with services and displaying information about the unit and unit files that systemd is maintaining. However, we can find out more specific information about units using some additional commands. The following topics provide that specific information, by using the commands shown.

Displaying a Unit File

To display the unit file that systemd has loaded into its system, you can use the cat command. For example, to see the unit file for the sshd daemon, type the following:

$ **sudo systemctl cat sshd**

[sudo] password for bob: **zzzzz**

# /lib/systemd/system/ssh.service

**[Unit]**

**Description=OpenBSD Secure Shell server**

**After=network.target auditd.service**

**ConditionPathExists=!/etc/ssh/sshd\_not\_to\_be\_run**

**[Service]**

**EnvironmentFile=-/etc/default/ssh**

**ExecStart=/usr/sbin/sshd -D $SSHD\_OPTS**

**ExecReload=/bin/kill -HUP $MAINPID**

**KillMode=process**

**Restart=on-failure**

**RestartPreventExitStatus=255**

**Type=notify**

**[Install]**

**WantedBy=multi-user.target**

**Alias=sshd.service**

$

The output is the unit file as available to the currently running systemd process. This can be critical if you have modified unit files recently, or if you are overriding certain options in a unit file that was installed and built by the package management system.

Displaying Dependencies

To see a unit's dependency tree, you can use the list-dependencies command:

$ **sudo systemctl list-dependencies sshd.service**

This will display a hierarchy mapping the dependencies that must be dealt with in order to start the unit in question. Dependencies, in this context, include those units that are either required by or wanted by the units above it.

sshd.service

├─system.slice

└─basic.target

├─microcode.service

├─rhel-autorelabel-mark.service

├─rhel-autorelabel.service

├─rhel-configure.service

├─rhel-dmesg.service

├─rhel-loadmodules.service

├─paths.target

├─slices.target

Output truncated…

The recursive dependencies are only displayed for .target units, which indicate system states. To recursively list all dependencies, include the --all flag.

To show reverse dependencies (units that depend on the specified unit), you can add the --reverse flag to the command. Other flags that are useful are the --before and --after flags, which can be used to show units that depend on the specified unit starting before and after themselves, respectively.

Checking Unit Properties

To see the low-level properties of a unit, you can use the show command. This will display a list of properties that are set for the specified unit using a key=value format:

$ **sudo systemctl show sshd.service**

Type=notify

Restart=on-failure

NotifyAccess=main

RestartUSec=100ms

TimeoutStartUSec=1min 30s

TimeoutStopUSec=1min 30s

RuntimeMaxUSec=infinity

WatchdogUSec=0

WatchdogTimestamp=Mon 2016-08-08 15:51:56 PDT

WatchdogTimestampMonotonic=8131498

FailureAction=none

PermissionsStartOnly=no

RootDirectoryStartOnly=no

RemainAfterExit=no

GuessMainPID=yes

MainPID=914

Output truncated…

If you want to display a single property, you can use the -p flag with the property name. For example, to see the conflicts that the sshd.service unit has, you can type:

$ **sudo systemctl show sshd.service -p Conflicts**

Conflicts=shutdown.target

Masking and Unmasking Units

At the beginning of Section 18.5.1.1 in the printed book, we sawhow to stop or disable a service. But systemd also has the ability to mark a unit as completely unstartable, automatically or manually, by linking it to /dev/null. This is called “masking” the unit, and is done with the mask command:

$ **sudo systemctl mask nginx.service**

This will prevent the nginx service from being started, automatically or manually, for as long as it is masked.

If you check the list-unit-files, you will see the service is now listed as masked:

$ **sudo systemctl list-unit-files**

If you attempt to start the service, you will see a message like this:

$ **sudo systemctl start nginx.service**

Failed to start nginx.service: Unit nginx.service is masked.

To unmask a unit, making it available for use again, simply use the unmask command:

$ **sudo systemctl unmask nginx.service**

This will return the unit to its previous state, allowing it to be started or enabled.

Editing Unit Files

While the specific format for all unit files is outside of the scope of this tutorial, systemctl provides several built-in mechanisms for editing and modifying unit files if you need to change them. The edit command, by default, will open a unit file snippet for the unit in question:

$ **sudo systemctl edit nginx.service**

This will be a blank file that can be used to override or add directives to the unit definition. A directory will be created within the /etc/systemd/system directory which contains the name of the unit with .d appended. For instance, for the nginx.service, a directory called nginx.service.d will be created.

Within this directory, a snippet will be created called override.conf. When the unit is loaded, systemd will, in memory, merge the override snippet with the full unit file. The snippet's directives will take precedence over those found in the original unit file.

If you wish to edit the full unit file instead of creating a snippet, you can pass the --full flag:

$ **sudo systemctl edit --full nginx.service**

This will load the current unit file into the default editor, where it can be modified. On our Linux Mint system, the default editor was nano. When you save and exit the editor, the saved file will be written to /etc/systemd/system, which will take precedence over the system's unit definition, found in /lib/systemd/system.

To remove any changes you made, either delete the unit's .d configuration directory or the modified service file from /etc/systemd/system. For example, to remove a snippet, we could type:

$ **sudo rm -r /etc/systemd/system/nginx.service.d**

To remove a full modified unit file, we would type:

$ **sudo rm /etc/systemd/system/nginx.service**

After deleting the file or directory, you should reload the systemd process so that it no longer tries to reference these files and reverts back to using the system copies. The following command does this:

$ **sudo systemctl daemon-reload**

In-Chapter Exercise

4. To get practice with the service management commands presented in the above section W27.5.1.7, execute them on your system, using the nginx service rather than sshd.

W27.5.1.8.2 Practicing on Target Units

Systemd “target” states are represented by target units. Target units end with the .target file extension, and they are used to group together other systemd units through dependencies.

For example, the graphical.target unit, which is used to start a GUI session on a desktop system, starts system services such as the Mint Display Manager (mdm.service) or Accounts Service (accounts-daemon.service) on our representative Linux Mint system. It also activates the multi-user.target unit, because that target is a dependency of graphical.target. The multi-user.target unit starts other essential system services such as NetworkManager (NetworkManager.service) or D-Bus (dbus.service), and has as a dependency a milestone target unit (as seen in Figure 18.2 in the printed book) named basic.target.

W27.5.1.8.3 Viewing the Default Target

To determine which target unit is the final target determining the overall state of the system, use the following command:

$ **sudo systemctl get-default**

graphical.target

This command resolves the symbolic link located at /etc/systemd/system/default.target and displays the result.

Note carefully that the default target unit can be different from the current target that defines the current state of the system! We illustrate this in Section W27.5.1.8.6.

W27.5.1.8.4 Viewing All Targets

To list all currently loaded target units, type the following command at a shell prompt:

$ **sudo systemctl list-units --type target**

For each target unit, this command displays its full name (UNIT) followed by a note whether the unit has been loaded (LOAD), its high-level (ACTIVE) and low-level (SUB) unit activation state, and a short description (DESCRIPTION).

By default, the systemctl list-units command displays only active units. If you want to list all loaded units regardless of their state, run this command with the --all or -a command line option:

$ **sudo systemctl list-units --type target --all**

When we executed this command on our representative Linux Mint system, we got the following output:

$ **systemctl list-units --type target --all**

UNIT LOAD ACTIVE SUB DESCRIPTION

basic.target loaded active active Basic System

cryptsetup.target loaded active active Encrypted Volumes

emergency.target loaded inactive dead Emergency Mode

getty.target loaded active active Login Prompts

graphical.target loaded active active Graphical Interface

local-fs-pre.target loaded active active Local File Systems (Pre)

local-fs.target loaded active active Local File Systems

multi-user.target loaded active active Multi-User System

network-online.target loaded active active Network is Online

network-pre.target loaded active active Network (Pre)

network.target loaded active active Network

nfs-client.target loaded active active NFS client services

nss-user-lookup.target loaded active active User and Group Name Lookups

paths.target loaded active active Paths

remote-fs-pre.target loaded active active Remote File Systems (Pre)

remote-fs.target loaded active active Remote File Systems

rescue.target loaded inactive dead Rescue Mode

rpcbind.target loaded inactive dead RPC Port Mapper

shutdown.target loaded inactive dead Shutdown

slices.target loaded active active Slices

sockets.target loaded active active Sockets

swap.target loaded active active Swap

sysinit.target loaded active active System Initialization

● syslog.target not-found inactive dead syslog.target

time-sync.target loaded active active System Time Synchronized

timers.target loaded active active Timers

umount.target loaded inactive dead Unmount All Filesystems

zfs.target loaded active active ZFS startup target

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

28 loaded units listed.

To show all installed unit files use 'systemctl list-unit-files'.

$

W27.5.1.8.5 Viewing the Currently Loaded Targets

To list all currently loaded target units, run the following command:

$ s**udo systemctl list-units --type target**

The following command illustrates the systemd logging facility (which has effectively replaced older, more traditional logging mechanisms) known as *journald*. The command that allows access to systemd-style logging is the **journalctl** command. We describe journalctl in detail in Section W27.6.2. To use **journalctl**, along with the **grep** command, to list the current target, type the following:

$ **sudo journalctl | grep Reached | tail -3**

Jul 17 20:31:44 bob-ProLiant-MicroServer systemd[1]: Reached target Multi-User System.

Jul 17 20:32:27 bob-ProLiant-MicroServer systemd[1]: Reached target User and Group Name Lookups.

Jul 17 20:32:27 bob-ProLiant-MicroServer systemd[1]: Reached target Graphical Interface.

The above command will show you the current target unit and the two previous to it.

W27.5.1.8.6 Changing the Current Target by Isolating Targets

It is possible to start all of the units associated with a target and stop all units that are not part of the dependency tree for that target. This is similar to changing the runlevel in older, legacy init systems.

For instance, if you are operating in a GUI environment, with graphical.target actively defining the system state, you can shut down the graphical system, and put the system into a multi-user CUI state by isolating the multi-user.target. This is done with the isolate command. Since graphical.target (lower down on the dependency tree shown in Figure W27.2) depends on multi-user.target, all of the graphical units below multi-user.target will be stopped.

But you need to carefully look at the dependencies of the target you are isolating before performing this procedure, to ensure that you are not stopping vital services. Use the following command to do this:

$ **sudo systemctl list-dependencies multi-user.target**

To change to a different target unit in the current session, type the following at a shell prompt as root:

sudo systemctl isolate name.target

Replace name with the name of the target unit you want to use (for example, multi-user). This command starts the target unit named name and all dependent units, and immediately stops all others. To turn off the graphical user interface and change to the multi-user.target unit in the current session, run the following command as root:

$ **sudo systemctl isolate multi-user.target**

Caveat-

After executing the above command on 3 of 4 of our representative Linux systems, at the time of the writing of this book, it was necessary to change the virtual terminal to another virtual terminal (tty1 through 7 were the possiblities), such as tty2 by holding down the following keystroke sequence at one time-

**<Ctrl>+<Alt>+F2**

where **F2** represents the function key F2 on your keyboard.

You are then able to login to the system now running in the multi-user.target state, in a text-only, CUI interface. This caveat was necessary because the active graphical display manager, and the normal virtual terminal for display of X Window System programs on our desktop, did not hook up tty7 (the default screen display in a graphical.target state) to the display screen! So we simply switched our virtual display to tty2 with the keystroke sequence above to facilitate the target isolation.

We have tried the command-

**sudo systemctl isolate multi-user.target**

on our representative Debian, Ubuntu, Linux Mint, and CentOS systems, and found that you need to use the caveat on Debian-family systems, but not on CentOS. CentOS switched to multi-user.target in the same tty.

To switch back to a graphical.target state, run the following command:

$ **sudo systemctl isolate graphical.target**

W27.5.1.8.7 Changing the Default Target

To configure the system to use a different target unit by default when it starts up, type the following at a shell prompt:

**sudo systemctl set-default name.target**

Replace name with the name of the target unit you want to use by default (for example, multi-user). This command replaces the /etc/systemd/system/default.target file with a symbolic link to /usr/lib/systemd/system/name.target, where name is the name of the target unit you want to use.

To configure the system to use the multi-user.target unit by default, run the following command as root:

$ **sudo systemctl set-default multi-user.target**

After you restart the system, it will allow you to exclusively and permanently login via a text-only, CUI. To permanently switch back to a graphical.target state on every restart, use the following command

$ **sudo systemctl set-default graphical.target**

W27.5.1.8.8 Changing to Rescue Mode

Generally, an ordinary user of a Linux system, would not have to use the commands found in this section, and the following section. That is because a very large part of the time, the system is operating normally, and also starts up normally.

Rescue mode provides a very valuable single-user environment which allows you to repair your system, when it is unable to complete the regular boot and Startup processes. In rescue mode, the system attempts to mount all local file systems and start some important system services, but it does not activate network interfaces or allow multiple users to be logged into the system at the same time.

To change the current target and enter rescue mode in the current session, type the following at a shell prompt as root:

$ **sudo systemctl rescue**

Welcome to rescue mode! After logging in, type-

journalctl -xb to view system logs,

systemctl reboot to reboot,

systemctl default or the key combination **<Ctrl>+D** to boot into default mode:

Give root password for maintenance (or press <Ctrl>+D to continue):

This command is similar to systemctl isolate rescue.target, but it also sends an informative message to all users that are currently logged into the system. To prevent systemd from sending this message, run this command with the --no-wall command line option:

$ **sudo systemctl --no-wall rescue**

W27.5.1.8.9 Changing to Emergency Mode

Emergency mode provides the most minimal environment possible and allows you to repair your system even in situations when the system is unable to enter rescue mode. In emergency mode, the system mounts the root file system only for reading, does not attempt to mount any other local file systems, does not activate network interfaces, and only starts a few essential services.

To change the current target and enter emergency mode, type the following at a shell prompt as root:

$ **sudo systemctl emergency**

Welcome to emergency mode!

After logging in, type journalctl -xb to view system logs,

systemctl reboot to reboot,

systemctl default or <Ctrl>+D to boot into default mode.

Give root password for maintenance (or press Control-D to continue):

This command is similar to **systemctl isolate emergency.target**, but it can also send an informative message to all users that are currently logged into the system. To prevent systemd from sending this message, run this command with the --no-wall command line option:

$ **sudo systemctl --no-wall emergency**

The above two sections, that allow you to change the system state to rescue or emergency modes, are most likely to be used when the system is not working optimally. But then what does an ordinary user do when a normal boot into either multi-user.target or graphical.target cannot be done at all, for whatever reason? Generally, that is not the case.

Barring power or hardware-related issues, such as failed hard disk or memory, it is possible to use GRUB2 to enter a “recovery” mode upon booting. On Linux Mint for example, when POST is completed successfully, and just before GRUB2 takes over the boot process, you can hold down the Shift key on the keyboard to enter a GRUB2 menu. The advanced option of that menu includes choices for booting into previous “boot environments”, and into recovery modes for those environments. These environments are named according to kernel version. So if you have upgraded the operating system kernel during the course of using your computer system, different environments with different kernels will be available to you. For an ordinary user, this choice for troubleshooting the system, by entering these system states during the boot stage, is a useful possibility that would help you recover data, logs, and other valuable information.

W27.5.1.8.10 Practice in Working With Targets

Here is a short command line practice session that allows you to work with the systemctl command and options shown in the preceding sections, and systemd target states.

1. Check the default systemd.target-

$ **sudo systemctl get-default**

graphical.target

$

2. List the target units and determine the current systemd target with the Linux multiple command line who -r command, options, and grep-

$ **sudo systemctl list-units --type=target |grep active |egrep "graphical|multi|rescue|emergency"**

graphical.target loaded active active Graphical Interface

multi-user.target loaded active active Multi-User System

The following command checks the target state we are currently in:

$ **sudo who -r**

run-level 5 2017-08-10 13:50

$

The legacy run-level 5 designation corresponds to the graphical.target state.

3. Now we can change the systemd target to multi-user.target , and use the Linux multi-command systemctl list-units, with grep and egrep to check units availability and status.

$ **sudo systemctl set-default multi-user.target**

Removed symlink /etc/systemd/system/default.target.

Created symlink from /etc/systemd/system/default.target to /lib/systemd/system/multi-user.target.

$ **sudo systemctl list-units --type=target | grep active | egrep "graphical|multi|rescue|emergency"**

graphical.target loaded active active Graphical Interface

multi-user.target loaded active active Multi-User System

$ **sudo who -r**

run-level 5 2017-04-18 03:46

$

Remember that dependencies between targets mean one systemd target can be part of another systemd target. Both graphical. target includes multi-user.target and multi-user.target depend on various other targets. Now check the systemd target dependencies using the following systemctl list-dependencies command, which checks the dependencies for systemd multi-user.target:

$ **sudo systemctl list-dependencies multi-user.target |grep target**

multi-user.target

● ├─basic.target

● │ ├─paths.target

● │ ├─slices.target

● │ ├─sockets.target

● │ ├─sysinit.target

● │ │ ├─cryptsetup.target

● │ │ ├─local-fs.target

● │ │ └─swap.target

● │ └─timers.target

● ├─getty.target

● └─remote-fs.target

● └─remote-fs-pre.target

$

To list the available systemd targets on the system, use the following command:

$ **sudo systemctl list-units --type=target**

UNIT LOAD ACTIVE SUB DESCRIPTION

basic.target loaded active active Basic System

cryptsetup.target loaded active active Encrypted Volumes

getty.target loaded active active Login Prompts

graphical.target loaded active active Graphical Interface

local-fs-pre.target loaded active active Local File Systems (Pre)

local-fs.target loaded active active Local File Systems

multi-user.target loaded active active Multi-User System

network-online.target loaded active active Network is Online

network.target loaded active active Network

nss-user-lookup.target loaded active active User and Group Name Lookups

paths.target loaded active active Paths

remote-fs-pre.target loaded active active Remote File Systems (Pre)

remote-fs.target loaded active active Remote File Systems

slices.target loaded active active Slices

sockets.target loaded active active Sockets

swap.target loaded active active Swap

sysinit.target loaded active active System Initialization

time-sync.target loaded active active System Time Synchronized

timers.target loaded active active Timers

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

19 loaded units listed. Pass --all to see loaded but inactive units, too.

To show all installed unit files use 'systemctl list-unit-files'.

4. Reboot the system using the systemctl reboot command. Since we set the default systemd target to multi-user.target, the system will restart into that target state.

$ **sudo systemctl reboot**

Output truncated...

5. Once the system has restarted, check the systemd target units, and the current active target state.

$ **sudo systemctl list-units --type=target |grep active |egrep "graphical|multi|rescue|emergency"**

basic.target loaded active active Basic System

cryptsetup.target loaded active active Encrypted Volumes

getty.target loaded active active Login Prompts

local-fs-pre.target loaded active active Local File Systems (Pre)

local-fs.target loaded active active Local File Systems

mail-transport-agent.target loaded active active Mail Transport Agent

**multi-user.target** loaded active active Multi-User System

network-online.target loaded active active Network is Online

network-pre.target loaded active active Network (Pre)

network.target loaded active active Network

paths.target loaded active active Paths

remote-fs-pre.target loaded active active Remote File Systems (Pre)

remote-fs.target loaded active active Remote File Systems

slices.target loaded active active Slices

sockets.target loaded active active Sockets

swap.target loaded active active Swap

sysinit.target loaded active active System Initialization

time-sync.target loaded active active System Time Synchronized

timers.target loaded active active Timers

zfs.target loaded active active ZFS startup target

20 loaded units listed. Pass --all to see loaded but inactive units, too.

$ **sudo who -r**

run-level 3 2017-08-23 06:32

Since we are now in the multi-user.target state, multi-user.target is highlighted in the above output. Also, the sudo who -r command shows us we are in runlevel 3, the legacy equivalent of the multi-user.target state.

W27.5.1.9 Other systemctl Options that Work with Target Units

Targets are special unit files that describe a system state or synchronization point. Like other units, the files that define targets can be identified by the ending .target.

Targets are used to group units together.

This is done in order to bring the system to desired states, which allow use case-dictated functionality. For example, a Linux server system with a text-only CUI, without the performance overhead of a graphical display manager or desktop environment, or an emergency mode that allows debugging and rescue of data.

Targets are used as a reference point to make certain functionality available, allowing you to specify the “milestone” state consisting of many inter-dependent targets, rather than the individual units you would have to startup to produce that state.

Target unit files can specify, in their configuration, that they are WantedBy= or RequiredBy= some particular unit file, creating dependency relationships. Units that are required to be available can specify this condition using the Wants=, Requires=, and After= specifications to indicate that dependency relationship.

W27.5.1.10 Using Target Shortcuts

There are target units defined for critical events, such as powering off the system, or rebooting. The systemctl command has some direct shortcut options that give you quick and easy methods to execute those critical events.

To halt the system, you can use the halt command:

$ **sudo systemctl halt**

To initiate a full shutdown, you can use the poweroff command:

$ **sudo systemctl poweroff**

A restart can be started with the reboot command:

$ **sudo systemctl reboot**

The above commands all alert logged-in users that the critical event is occurring (very important in a mult-user system), something that running or isolating a particular target will not do.

Ordinary Linux system commands will link the necessary operations on the system, so that they work properly with systemd-controlled state changes.

For example, to reboot the system, you can simply type:

$ **sudo reboot**

In-Chapter Exercise

5. What systemd command from the above sections gives the procedure for Primary Example 1 in Chapter 18, Section 1. of the printed book?

W27.6 Other Important systemd Commands

In addition to the systemctl command that affects Units and Targets, systemd has other important commands that provides control of many system functions.

The following sub-sections detail the use of some of these other important systemd commands

W27.6.1 Cgroups

Control groups (cgroups), as updated by the inclusion of systemd in the Linux kernel, allow you to create hierarchically-ordered groups of processes running on your system. They can be affected by the systemd systemctl command. Practically speaking, among the various ways of using cgroups with systemd, the most important one is to monitor and control Linux system resources. As outlined in Chapter 17, Section 8.1.2 in the printed book, this monitoring, assessment, and control can be used at the site of systemd unit files. It can be short-term (transient) or long-term (persistent).

As shown in Chapter 17, Section 8.1.2 in the printed book, the systemd-run command is used to create and start a transient, or temporary, unit as its own cgroup module, and then run a Linux command in the unit.

To create persistent, or permanent, cgroups, it is necessary to construct service unit files for them in the /etc/systemd/system directory. This is the standard location for user-installed and user-defined services in systemd.

The objective of this section is to provide a graphic description of the cgroup “tree”, give some basic definitions, further describe the structure of cgroup unit types, and illustrate their standard arrangement. We define Linux “namespaces”, and point you to a system programming example of namespace creation. We also provide sources of further documentation for you to explore.

W27.6.1.1 Default Cgroup Hierarchies for System Resource Control

It is best to begin with a couple of instructive diagrams, the first one of which you can easily reproduce on your system with the systemd-cgls command. Following are the abbreviated results of that command on two of our Linux Mint systems, the first one running the graphical.target with the lxd container virtualizer installed, and an lxc container named containerx running in it. The second one shows the results of the command run inside the lxc container:

$ **systemd-cgls**

Control group /:

-.slice

├─1021 /sbin/cgmanager -m name=systemd

├─lxc

│ └─containerx

│ ├─init.scope

│ │ └─2495 /sbin/init

│ └─system.slice

...

├─init.scope

│ └─1 /sbin/init splash

...

│ ├─lxd.service

│ │ ├─1862 /usr/bin/lxd --group lxd --logfile=/var/log/lxd/lxd.log

│ │ ├─2389 dnsmasq --strict-order --bind-interfaces --pid-file=/var/lib/lxd/n...

│ │ └─2478 [lxc monitor] /var/lib/lxd/containers containerx

...

│ ├─vsftpd.service

│ │ └─1839 /usr/sbin/vsftpd /etc/vsftpd.conf

│ ├─cups.service

│ │ ├─ 922 /usr/sbin/cupsd -l

│ │ └─1182 /usr/lib/cups/notifier/dbus dbus://

...

└─user.slice

└─user-1000.slice

├─user@1000.service

│ └─init.scope

│ ├─3222 /lib/systemd/systemd --user

│ └─3226 (sd-pam)

└─session-c1.scope

├─2205 /usr/sbin/mdm --nodaemon

├─3235 /usr/bin/gnome-keyring-daemon --daemonize --login

├─3315 cinnamon-session --session cinnamon

...

├─3892 /usr/lib/gnome-terminal/gnome-terminal-server

├─3896 bash

├─3926 /home/bob/.dropbox-dist/dropbox-lnx.x86-32.4.23/dropbox

└─4018 systemd-cgls

containerx $ **systemd-cgls**

Control group /:

-.slice

├─649 bash

├─init.scope

│ └─1 /sbin/init

├─system.slice

...

│ ├─cron.service

│ │ └─300 /usr/sbin/cron -f

...

└─user.slice

└─user-1001.slice

├─session-c1.scope

│ ├─686 sshd: bob [priv]

│ ├─720 sshd: bob@pts/0

│ ├─723 -bash

│ ├─746 systemd-cgls

│ └─747 systemd-cgls

└─user@1001.service

└─init.scope

├─688 /lib/systemd/systemd --user

└─689 (sd-pam)

In-Chapter Exercises

6. Compare the output of the above two commands with the pstree output shown in Chapter 10, Section 7.1. How are they similar in both structure and content? How are they different? Execute the pstree command and the systemd-cgls command on your system, and compare the output you get to the output we show, in terms of fine-grained similarities and differences. What options of the pstree command can give displays of PID’s, similar to the output of the systemd-cgls command?

7. Are PID and cgroup number the same for all processes and threads? Using the techniques of Chapter 17, write a system program that creates multiple persistent threads with the fork() system call, and then examine the cgroup numbers assigned to those threads.

We provide some basic definitions to more fully describe the objects found in the systemd-cgls commands output diagrams.

Slice — A grouping of units that organizes them in some way. Slices are not numbered processes, they just produce, or are the framework, within which service and scope units (defined below) are placed. Processes are contained in services or in scopes. In the cgroup tree, every name of a unit that is contained in a slice shows the “path” to a location in the tree. The dash ("-") character separates the path components. For example, user-1001.slice path components contain the parent slice, “user” and the named slice, “1001.slice”. There is one root slice, -.slice.

Service — A process, or a group of processes, which is started using a service unit configuration file. Services contain the specified processes that can be controlled with the systemctl command.

For example, in the diagrams above, systemd-logins.service is a service.

Scope — Processes that are started and stopped by transient processes that use the fork() system call, and are registered by systemd at runtime. For example, all user sessions.

In the diagrams above, initial.scope, session-c1.scope are examples.

Slices, services, and scopes, can be very usefully created by the system administrator, or by system programs. By default though, systemd and the operating system start up mandatory and essential services automatically at system startup, according to the final target state that the system will run in.

Also, there are four slices created by default, as seen in the first systemd-cgls command output shown:

* -.slice — the root slice;
* system.slice — the default path location for all system services;
* user.slice — the default path location for all user sessions;
* machine\_name.slice — if lxd has been installed, the location(s) for all lxd/lxc Linux containers, where machine\_name is the name of the lxc containers that have been registered with systemd. For example, in the first systemd-cgls diagram above, directly under the -.slice, lxc-containerx.

Notice from the above systemd-cgls command diagrams that all user sessions are automatically placed in a distinct and separate scope unit.

In-Chapter Exercise

8. Are the threads you created for Exercise W27.14 slices, services, or scope unit? Why?

To summarize, looking at the output of the systemd-cgls commands above, the root of the cgroup tree is the *root slice*, -.slice. Proceeding down the tree, there are a number of slices grouped under the major “branch” known as the system.slice, for example avahi-daemon.service, cron.service, cups.service, ssh.service, etc.. Most interestingly, under the system.slice, an lxd/lxd container which is installed on the system can also be found. The next major branch, aside from the system.slice, is the *user.slice*, with a number of *scopes* under that for user sessions. That is a capsule overview of the tree.

It is useful to list loaded slice unit types, by use the following command:

$ **systemctl list-units –type=slice**

UNIT LOAD ACTIVE SUB DESCRIPTION

-.slice loaded active active Root Slice

system-getty.slice loaded active active system-getty.slice

system-systemd\x2dbacklight.slice loaded active active system-systemd\x2dbacklight.slice

system.slice loaded active active System Slice

user-1000.slice loaded active active User Slice of bob

user.slice loaded active active User and Session Slice

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

6 loaded units listed. Pass --all to see loaded but inactive units, too.

To show all installed unit files use 'systemctl list-unit-files'.

$

W27.6.1.2 Additional cgroup Reference Resources

To find more information about resource control under systemd, the unit hierarchy, as well as the kernel resource controllers, refer to the materials listed below:

Cgroup-Related Systemd Documentation

The following man pages give you more information on systemd cgroups:

systemd.resource-control - describes the configuration options for resource control shared by system units.

systemd.unit - describes common options of all unit configuration files.

systemd.slice - provides general information about .slice units.

systemd.scope - provides general information about .scope units.

systemd.service - provides general information about .service units.

Additionally, you can install the kernel documentation on cgroups, by using the following command on a Linux Mint system:

$ **sudo apt install linux-doc**

Output truncated…

$

Once the documentation for the kernel has been downloaded, you can access the cgroups-specific content using the following command on Linux Mint:

$ **cd /usr/share/doc/linux-doc/cgroups**

$

To view what resource information is available in that documentation directory, and then view content for resources such as CPU, memory, etc., use the following command on a Linux Mint system:

$ **ls**

00-INDEX devices.txt.gz namespace.txt.gz

blkio-controller.txt.gz freezer-subsystem.txt.gz net\_cls.txt

cgroups.txt.gz hugetlb.txt net\_prio.txt

cpuacct.txt memcg\_test.txt.gz pids.txt

cpusets.txt.gz memory.txt.gz unified-hierarchy.txt.gz

$ less namespace.txt.gz

Output Truncated…

$

Compare the output of the **less namespace.txt.gz** command to the man page for namespaces, which we give a capsule overview of in the next sub-section..

W27.6.1.3 Linux Namespaces

A kernel-level facility particular to Linux, known as “namespaces”, in effect isolates cgroup processes and their system resources in their own environment to protect them from all other process environments on the system. A very close analogy of namespaces, which you might be familiar with, is the concept of a variables “scope” in computer programming languages.

Namespaces deploy the **clone** system call to achieve this process isolation. The premiere use of namespaces is in the system-level creation and maintenance of Linux containers, such as LXD. We show many complete examples of the user-level application of containers and LXC/LXD, at the book website site, in Chapter W23, entitled “Virtualization Methodologies”. For more information on namespaces, see the man pages on your system for namespaces(7), and user\_namespaces(7) ; particularly the EXAMPLE on the user\_namespaces man page, which gives an excellent and instructive system programming implementation of the clone system call used to create a child process that executes a shell command in a new namespace.

W27.6.2 Journal Logging

As introduced in Chapter 17, system logging and log files refer to the recording of general and specific actions and events on a UNIX and Linux system. Both families have very similar traditional methods available for logging. Logs are produced for a Linux system administrator in order to audit the general operation of the system, particularly with regard to performance enhancement and the maintenance of system security.

W27.6.2.1 systemd journal Log Messages

The journal is created and controlled by the journald daemon, which directs all of the messages produced by the kernel, initrd, services, etc. into a record structure. The systemd journal is a single, centralized management tool for logs, regardless of where the log messages are sent from.

One important feature of using systemd is that log messages currently can be output using the traditional message printing APIs with the syslog function call, as well as by using the journal API function calls. We only cover the systemd method of logging in this chapter.

A critical, and somewhat controversial aspect of systemd journal logging, is that the log files are stored as binary data, and can be searched by specialized database traversal. They are not plain text files.

Storing the log data in a binary format means that the data can be displayed in useful output formats specific to database management technologies, as well as by using the journalctl command that we show in the next section.

W27.6.2.2 Using the journalctl Command to Query the Journal

The journalctl command is a convenient way of querying entries in the journal database. Following is a synopsis of the command-

**journalctl** - Query the systemd journal

**Syntax:**

journalctl [OPTIONS...] [MATCHES...]

**Purpose:**

journalctl may be used to query the contents of the systemd journal as written by systemd-journald.service. If called without options or arguments, it will show the full contents of the journal, starting with the oldest entry collected.

All users are granted access to their private per-user journals. However, by default, only root and users who are members of a few special groups are granted access to the system journal and the journals of other users.

**Commonly Used Features:**

-a, --all Show all fields in full, even if they include unprintable characters or are very long.

-f, --follow Show only the most recent journal entries, and continuously print new entries as

they are appended to the journal.

-r, --reverse Reverse output so that the newest entries are displayed first.

-u, --unit=UNIT|PATTERN Show messages for the specified systemd unit UNIT

(such as a service unit), or for any of the units matched by PATTERN.

-S, --since=, -U, --until= Start showing entries on or newer than the specified date, or on or

older than the specified date, respectively. Date specifications

should be of the format "2012-10-30 18:17:16".

To get more information about the use of the journalctl command, particularly about the structure that the command uses to query the journal, use the man journalctl command on your system.

W27.6.2.3 Journal Logging Basics, and Applied to the Webserver2 Program

In this section, we show how to use the journalctl command and its options and arguments to do some basic systemd-style journal query operations. We further apply a set of those commands and options to a program developed in Chapter 18, Section 5.2, named “webserver2”. That program generates systemd-style journal log output. Our use of the journalctl command applied to that program’s journal output is relevant in this section, so as to better and more practically illustrate the use of journalctl options and arguments to a real application we develop.

W27.6.2.3.1 Basic Log Viewing

To see the logs that the journald daemon has collected, use the journalctl command.

When used without any options or arguments, every journal entry that is in the system will be displayed. The oldest entries will be first in the listing:

$ **journalctl**

-- Logs begin at Wed 2016-07-27 15:14:52 PDT, end at Sat 2016-07-30 12:32:36 PDT. --

Jul 27 15:14:52 bob-ProLiant-MicroServer systemd-journald[329]: Runtime journal (/run/log/journal/) is 7.3M, max 58.4M, 51.1M free.

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Initializing cgroup subsys cpuset

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Initializing cgroup subsys cpu

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Initializing cgroup subsys cpuacct

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Linux version 4.4.0-31-generic (buildd@lgw01-16) (gcc version 5.3.1 20160413 (Ubuntu 5.3.1-14ubuntu2.1) ) #5

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Command line: BOOT\_IMAGE=/boot/vmlinuz-4.4.0-31-generic root=UUID=cfd3750d-3a6c-49e0-89b9-55e402e5aafb ro qu

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: KERNEL supported cpus:

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Intel GenuineIntel

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: AMD AuthenticAMD

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: Centaur CentaurHauls

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: tseg: 0000000000

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: x86/fpu: Legacy x87 FPU detected.

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: x86/fpu: Using 'lazy' FPU context switches.

Jul 27 15:14:52 bob-ProLiant-MicroServer kernel: e820: BIOS-provided physical RAM map:

Output truncated…

But what if you want to see the journal with the newest entry first? Use the –r (reverse) option on the basic journalctl command.

$ **journalctl -r**

-- Logs begin at Wed 2016-07-27 15:14:52 PDT, end at Sat 2016-07-30 12:32:36 PDT. --

Jul 30 12:32:36 bob-ProLiant-MicroServer console-kit-daemon[1552]: missing action

Jul 30 12:32:36 bob-ProLiant-MicroServer console-kit-daemon[1552]: (process:25803): GLib-CRITICAL \*\*: g\_slice\_set\_config: assertion 'sys\_page\_size == 0' fail

Jul 30 12:32:36 bob-ProLiant-MicroServer systemd[1]: Started Session 87 of user bob.

Jul 30 12:32:36 bob-ProLiant-MicroServer systemd-logind[733]: New session 87 of user bob.

Jul 30 12:32:36 bob-ProLiant-MicroServer sshd[25799]: pam\_unix(sshd:session): session opened for user bob by (uid=0)

Jul 30 12:32:36 bob-ProLiant-MicroServer sshd[25799]: Accepted password for bob from 192.168.0.11 port 54189 ssh2

Output truncated…

In-Chapter Exercise

9. From the output of the above two commands, what can you tell about the most recent current boot time, and the current time?

W27.6.2.3.2 Journal Query Structures

The primary purpose of gathering the log information from many sources and centralizing them in one place, such as the journal, is to be able to quickly and easily observe and assess entries in the log that are important to you for some reason. That could be true of an ordinary desktop user, or a server system administrator.

Because of this, the most important use features of the journalctl command are its methods of structuring the querying of the journal, or culling more useful, understandable, and compact information from it. Following are some of the ways these searches through the log can be done.

For example, the journal has many “fields” that can be used for querying. Each of these fields acts as an index, or key, into specific classes of entries in the journal. Some of those fields are passed from the process being logged and some are applied by the daemon journald, with information it gathers from the system at the time of the log.

A leading underscore indicates that a field is of the latter type. The journal automatically records and indexes logging for that type of query. You can get more information about all of the available journal fields by typing:

$ **man systemd.journal-fields**

W27.6.2.3.3 Querying by Time

Often you may want to see parts of the log in a from-to manner. This is true when the system has been operating for a long time without a reboot.

You can query by using the --since and --until options, which restrict the entries displayed to those after or before the given times.

The time parameters are in a variety of formats. For absolute time values, you should use the following format:

YYYY-MM-DD HH:MM:SS

For example, we can see all of the entries since January 10th, 2017 at 9:17 PM using this command:

$ **journalctl --since "2017-01-10 21:15:00"**

If parts of the above time specification argument are left off, standard defaults are used instead. For example, if the date is omitted, the current date will be used. If the time component is missing, "00:00:00" (midnight) will be used. The seconds field can be left off to default to "00":

$ **journalctl --since "2017-01-10" --until "2017-01-11 03:00"**

The journal also understands some relative values and English-language shortcuts. For example, you can use the words "yesterday", "today", "tomorrow", or "now". You do relative times by placing these sysmbols before arguments: "-" or "+" to a numbered value or using words like "ago" .

To get the data from yesterday, use this command:

$ **journalctl --since yesterday**

If you used another system monitoring tool, and it gave information about a service interruption starting at 11:00 PM and continuing until an hour ago, you could type:

$ **journalctl --since 11:00 --until "1 hour ago"**

W27.6.2.3.4 Querying By Unit

The most useful and practical way of querying is by the service unit you are interested in. You use the -u option to query by unit.

For example, to see all of the logs from the webserver2 unit on your system, we can type:

$ **journalctl -u webserver2.service**

You could perform a compound query by adding arguments to the above unit query. To query by time as well, to check on how the service is running today, you can type:

$ **journalctl -u webserver2.service --since today**

Compound querying is very helpful when you want to compare log entries from possibly related units. For example, if you want to relate log entries from your webserver2 service to another unit, you can view the entries from both in chronological order by using the following compound query statement:

$ **journalctl -u webserver2.service -u systemd-logind.service --since yesterday**

This allows you to study the interactions between different programs and debug targets and the interaction of dependencies, instead of just individual, isolated units.

W27.6.2.3.5 Querying By Process, User, or Group ID

Some services fork many child processes to do work. If you know the exact PID of the process you are interested in, you can perform a query by PID that as well.

To do this, perform a query by specifying the \_PID field. For instance if the PID we're interested in is 1, we could type:

$ **journalctl \_PID=1**

You can also show all of the entries logged from a specific user or group. This can be done with the \_UID or \_GID filters. For instance, if webserver2 runs under the user bob, you can find the user ID by typing:

$ **id -u bob**

1000

Afterwards, you can use the ID that was returned to construct a query based on the results, as follows:

$ **journalctl \_UID=1000 --since today**

The -F option of the journalctl command is used to show all of the available values for a given journal field.

For example, to see which group IDs the systemd journal has entries for, you can type:

$ **journalctl -F \_GID**

125

1000

30

114

118

110

0

This show you all of the values that the journal has stored for the group ID field. This can help you construct your queries with the Group ID field.

W27.6.2.3.6 Querying By Component Path

We can also filter by providing a path location. If the path leads to an executable, journalctl will display all of the entries that involve the executable in question. For example, to find log entries that involve the webserver2 executable program, you can type:

$ **journalctl /home/bob/webserver2/webserver2**

where the path to the executable image is /home/bob/webserver2, and the name of the executable program is webserver2.

Usually, if a unit is available for the executable, that method is more understandable, and gives more useful information to someone assessing system administration tasks, such as security intrusions (entries from associated child processes, etc). Sometimes, however, this is not possible.

W27.6.2.3.7 Querying By Priority

One filter that system administrators often are interested in is the message priority. While it is often useful to log information at a very verbose level, when actually assessing the observed and available information, low priority logs can be distracting and confusing.

You can use journalctl to display only messages of a specified priority or above by using the -p option. This allows you to filter out lower priority messages.

For instance, to show only entries logged at the error level or above, you can type:

$ **journalctl - -b -p err**

This will show you all messages marked as error, critical, alert, or emergency. The journal implements the legacy syslog message levels. You can use either the priority name or its corresponding numeric value. In order of highest to lowest priority, these are:

|  |  |  |
| --- | --- | --- |
| Priority | Name | Description and Possible Action(s) to be Taken |
| 0 | emerg | Emergency- A "panic" condition - notify system administration. |
| 1 | alert | Alert- Notify system administrator who can fix the problem. Example: loss of backup ISP connection. |
| 2 | crit | Critical- Failure in a primary system. Fix crit problems before alert problems. Example: loss of disk subsystem. |
| 3 | err | Error- Non-urgent failures, should be sent to developers or development admins. |
| 4 | warning | Warning- Not an error, but shows that an error will occur if action is not taken, e.g. file system nearly full. |
| 5 | notice | Notice- Unusual events, but not error conditions. No immediate action required. |
| 6 | info | Information- Normal operating messages - No action required. |
| 7 | debug | Debug- Info useful for developers for debugging the app, not useful during operations. |

Table W27.3 systemd Error Priorities

The above numbers or names can be used interchangeably with the -p option. Selecting a priority will display messages marked at the specified level and those above it.

In-Chapter Exercises

10. What does the - -b directive in the previous command do, as far as a compound query?

11. After making the webserver2 program a service as shown in Section 7, what are the priorities of the log messages from webserver2 on your system? How did you find this out?

W27.6.2.4 Query Output Display

The above queries showed particular kinds of log entry outputs. There are other ways we can modify the structure of a query. We can adjust the journalctl output display so that it is more condensed, readable, and organized in a more understandable way.

To display a set amount of log entries, you can use the -n option. By default, it will display the most recent 10 entries:

$ **journalctl -n**

You can specify the number of entries you'd like to see with a number after the -n:

$ **journalctl -n 15**

Dynamically Following Log Entries in Real Time

To dynamically, in real time, watch the logs as they are being written, use the -f option

$ **journalctl -f**

We can also adjust how journalctl displays log data shrinking or expanding the output.

By default, journalctl will show the entire entry in the pager, allowing the entries to trail off to the right of the screen. This info can be accessed by pressing the right arrow key.

If you'd rather have the output truncated, inserting an ellipsis where information has been removed, you can use the --no-full option:

$ **journalctl - -no-full**

-- Logs begin at Mon 2016-08-08 15:51:54 PDT, end at Wed 2016-08-17 09:58:16 PDT. --

Aug 08 15:51:54 bob-ProLiant-MicroServer systemd-journald[325]: Time spent on flushing to /var is 48.603ms for 829 entries.

Aug 08 15:51:54 bob-ProLiant-MicroServer systemd[1]: Starting udev Coldplug all Devices...

Aug 08 15:51:54 bob-ProLiant-MicroServer kernel: shpchp: Standard Hot Plug PCI Controller Driver version: 0.4

Output truncated…

You can also go in the opposite direction with this and tell journalctl to display all of its information, regardless of whether it includes unprintable characters. We can do this with the -a flag:

$ **journalctl -a**

By default, journalctl displays the output of a query in a pager, so you can see one screenful at a time. If you are planning on processing the data with text manipulation tools, you can output to standard output, directly into a post-processing program, or to a disk file.

You can do this with the --no-pager option:

$ **journalclt --no-pager**

Query Output Formats

If you are processing journal log entries with some another program, as mentioned above, you most likely will have an easier time parsing the data if it is in a more consumable format. Luckily, the journal can be displayed in a variety of formats as needed. You can do this using the -o option with a format specifier.

For instance, you can output the journal entries in JSON by typing:

$ **journalctl -b -u webserver2 -o json**

This is useful for parsing with utilities. You could use the json-pretty format to get a better handle on the data structure before passing it off to the JSON consumer:

$ **journalctl -b -u webserver2 -o json-pretty**

The following table shows the formats can be used for display.

|  |  |
| --- | --- |
| cat | Displays only the message field itself. |
| export | A binary format suitable for transferring or backing up. |
| json | Standard JSON with one entry per line. |
| json-pretty | JSON formatted for better human-readability |
| json-sse | JSON formatted output wrapped to make add server-sent event compatible |
| short | The default syslog style output |
| short-iso | The default format augmented to show ISO 8601 wallclock timestamps. |
| short-monotonic | The default format with monotonic timestamps. |
| short-precise | The default format with microsecond precision |
| verbose | Shows every journal field available for the entry, including those usually hidden internally. |

Table W27.4 journalctl Output Formats

These options allow you to display the journal entries in whatever format best suits your current needs.

W27.6.2.5 Journal Maintenance

It may become necessary to not only look through older boot environments, and correlate the logs found in them with the most current boot logs, but to delete older, obsolete log entries. Log and journal maintenance is an important aspect of system administration, and contributes to the compactness, tractability, and utility of the journal.

At the time this book was written, “vacuuming”, or cleaning out journal log files, only works on archived files and corrupted files, and does not work on active log files. This will change in later releases of our base Linux system, Linux Mint, and we will provide updates on this issue at the author’s Github site for the book.

You can find out the amount of space that the active and archived journals are currently occupying on disk by using the --disk-usage option of the journalctl command:

$ j**ournalctl --disk-usage**

Archived and active journals take up 7.3M on disk.

The active journal logs cannot be pruned.

When this facility becomes available for active journal logs, there are basically two different ways you will be able to do that.

1. If you use the --vacuum-size option, you can shrink your journal by indicating a size. This will remove only old archived entries that are corrupted or empty, until the total journal space taken up on disk is at the requested size:

$ **sudo journalctl --vacuum-size=1M**

Deleted archived journal /run/log/journal/2a5d5f96ef9147c0b35535562b32d0ff/system@009d2c76a70642f68c5286581fd2388a-0000000000000001-00055bada90c29ee.journal (7.3M).

Vacuuming done, freed 7.3M of archived journals on disk.

$

2. Another way that you can shrink the archived journal log is by designating a cutoff time with the --vacuum-time option. Any entries beyond that time are deleted. This allows you to keep the entries that have been created after a specific time.

For instance, to keep entries from the last year, you can type:

$ **sudo journalctl --vacuum-time=1years**

*Limiting Journal Expansion*

You can limit how much persistent storage on disk the journal can take up. This can be done by editing the /etc/systemd/journald.conf file. The following items in that file can be used to limit the journal growth:

SystemMaxUse=: The maximum disk space that can be used by the journal in persistent storage.

SystemKeepFree=: The amount of space that the journal should leave free when adding journal entries to persistent storage.

SystemMaxFileSize=: How large individual journal files can grow to in persistent storage before being rotated.

RuntimeMaxUse=: The maximum disk space that can be used in volatile storage (within the /run filesystem).

RuntimeKeepFree=: The amount of space to be set aside for other uses when writing data to volatile storage (within the /run filesystem).

RuntimeMaxFileSize=: The amount of space that an individual journal file can take up in volatile storage (within the /run filesystem) before being rotated.

W27.6.2.6 Boot Process Querying

Using the journal, you can examine logs of the present boot record and its progress, and past boot records. This is useful from a system administration perspective, when something goes wrong with the boot process, or when software or hardware on the system fails to start or operate properly after the system boots and is in the steady state condition.

W27.6.2.6.1 Querying Past Boots

To display the information from the current boot, there are times when past boot sequence records would be helpful to examine and compare to the current one. The journal can save information from many previous boots, and the journalctl command can be made to display that information in an effective and concise comparative way.

Note that in order for you to retain journal information from past boots, you must complete the procedures shown in this section first.

To enable persistent boot information, you can do the following:

1. Create the directory to store the journal with the following command:

$ **sudo mkdir -p /var/log/journal**

2. Edit the journal configuration file with the editor of your choice:

$ **sudo emacs /etc/systemd/journald.conf**

Under the [journal] section, uncomment (remove the # sign) the Storage= option, and set it to "persistent" to enable persistent logging:

/etc/systemd/journald.conf

. . .

[Journal]

Storage=persistent

When retaining previous boots via persistence is enabled on your system, the journalctl command provides some options for working with boots as a unit. To see the boots that journald knows about, use the --list-boots option with journalctl:

$ **journalctl - -list-boots**

-1 1091930ee45b4afb8e85fff67f9e7d0a Sun 2016-07-31 10:06:30 PDT—Sun 2016-07-31 10:09:24 PDT

0 3649cf5465954570bd675a6af0485450 Sun 2016-07-31 10:10:45 PDT—Sun 2016-07-31 10:51:15 PDT

The above command will display a line for each boot as follows (boot -1 shown):

|  |  |
| --- | --- |
| -1 | The first column is the offset for the boot from 0, the current boot environment. |
| 1091930ee45b4afb8e85fff67f9e7d0a | An absolute reference, the boot ID is in the second column. |
| Sun 2016-07-31 10:06:30 PDT—Sun 2016-07-31 10:09:24 PDT | The time that the boot session spans, with two time specifications (from-to). |

Table W27.5 systemctl Boot Information

To display more verbose information from these boots, you can use information from either the first or second column. To see the journal from the previous boot, use the -1 relative pointer with the -b flag:

$ **journalctl -b -1**

You can also use the boot ID to call back the same data from boot -1:

$ **journalctl -b 1091930ee45b4afb8e85fff67f9e7d0a**

W27.6.2.6.2 The Current Boot

To display journal logs from the current boot, use the following command:

$ **journalctl -b**

This will show you all of the journal entries that have been collected since the most recent reboot, particularly if you have done the procedures in the previous section and then rebooted a number of times.

You can then monitor information about your current environment.

W27.6.2.6.3 Displaying Kernel Messages

Kernel messages related to booting, those usually found in dmesg output, can be retrieved from the journal as well. To display only these messages, we can add the -k or --dmesg flags to the journalctl command:

$ **journalctl -k**

By default, this will display the kernel messages from the current boot. You can specify an alternative boot environment using the normal boot selection flags above, being sure that a persistent boot environment was enabled. For instance, to get the messages from five boots ago, you could type:

$ **journalctl -k -b -5**

In-Chapter Exercises

12. Which command in the previous sections most compactly provides the procedure of Primary Example 2 in Chapter 18, Section 1. in the printed book?

13. To get practice with the journalctl commands from the previous section, query the boot record from 3 previous system boots from the current boot environment.

W27.6.3 systemd timers

systemd assumes much of what legacy Cron did, through built-in support for clock-time-based and calendar-time-based events.

Most importantly, systemd can do Cron-like scheduling of system events. We give examples below of clock-time-based running of a single script, and calendar-based scheduling of events.

In order to get a complete description of how systemd deals with time specifications, see the man page for systemd.time.

W27.6.3.1 An Example of Clock-Time-Based Running of a Script

If you have a script /usr/local/bin/myscript that you want to run every hour, do the following steps:

1. Create a service unit file, named myscript.service, with your favorite text editor. Save it in /etc/systemd/system/ ,with the following content:

**[Unit]**

**Description=Whatever MyScript Does**

**[Service]**

**Type=simple**

**ExecStart=/home/bob/myscript**

Note that it is important to set the Type variable to be “simple”, not “oneshot”. Using “oneshot” , the script will be run the first time, and then systemd will not run it again, and will turn off the timer.

2. Create the following timer unit file, in the same directory as the service unit file above.

**[Unit]**

**Description=Runs myscript every hour**

**[Timer]**

**# Time to wait after booting before we run first time**

**OnBootSec=10min**

**# Time between running each consecutive time**

**OnUnitActiveSec=1h**

**Unit=myscript.service**

**[Install]**

**WantedBy=multi-user.target**

3. To start and enable the service:

$ **sudo systemctl start myscript.timer**

and to enable it for every subsequent boot:

$ **sudo systemctl enable myscript.timer**

In-Chapter Exercises

14. #!/bin/bash

# My first script converted into a service

echo "Hello World!"

If the above script file in your home directory on the system (which you have named myscript) is made into a service with the 3 steps shown above, how can you get the message “Hello World!” to display every 10 minutes on the stdout of a terminal?

15. a) What command do you use to stop the service? b) What command do you use to ensure the service does not run on every subsequent boot?

W27.6.3.2 Example of Calendar-Based Running of the Above Script File

If you wish to start the service shown in Section W27.6.3.1 according to a calendar event, and not a clock-based interval, create a new timer unit and link the service unit file from that example to that new timer unit.

1. Create the timer unit with your favorite text editor, using the following command:

$ **sudo nano /etc/systemd/system/cal.timer**

Then put this text into that file-

**[Unit]**

**Description=Calendar-based timer**

**[Timer]**

**OnCalendar=Mon-Fri \*-\*-\* 00:00:00**

**Unit=myscript.service**

**[Install]**

**WantedBy=basic.target**

The service file and the timer unit file are put in the /etc/systemd/system/ folder.

2. To start the calendar timer-

$ **sudo systemctl start cal.timer**

and to enable it for every subsequent boot:

$ **sudo systemctl enable cal.timer**

In-Chapter Exercise

16. What does the time stamp Mon-Fri \*-\*-\* 00:00:00 in the above example specify for the running of the script file?

W27.7.1.3 A Python-Based Webserver as a “New-Style Daemon”

This example harnesses a special Python module that easily creates a webserver daemon, and puts this module under the control of systemd as a new-style daemon. You should have some familiarity with Python 3 from Chapter W19 on Python, but you don’t absolutely need to have gone through anything in that chapter, just to follow along with the steps shown below-

1. Create the following Python script file, which we named simple5.py, in your home directory with your favorite text editor:

**from wsgiref.simple\_server import make\_server**

**def app(environ, start\_response):**

**start\_response('200 OK', [('Content-Type','text/html')])**

**yield b"<h1>Linux: The Textbook</h1>"**

**server = make\_server('127.0.0.1', 8090, app)**

**server.serve\_forever()**

Be sure to follow the Python “rule of 4’s”, which means indent the third and fourth lines 4 spaces from the left margin!

2. In order for you to make this Python script an “executable” program, so that you can use the ExecStart directive in a service unit file to have systemd run the script as a service, use the following command:

$ **python –m py\_compile simple5.py**

The result of this command will put a compiled, and most importantly, executable Python program named simple5.pyc in your home directory. Make sure to also change the permissions on this file with the following command:

$ **chmod u+x simple5.pyc**

3. Create a service unit file for the webserver in /etc/systemd/system, named simple5.service, with the following content:

**[Service]**

**ExecStart=/home/bob/simple5.pyc**

4. To start the service, use the following command:

$ **sudo systemctl start simple5.service**

5. To view the web page this service provides, in a web browser on your local machine, type-in the URL 127.0.0.1:8090

The web page built into the simple5.py script file is now displayed in your browser.

6. To stop the service, type the following command:

$ **sudo systemctl stop simple5.service**

In-Chapter Exercise

17. Modify the Python script simple5.py, so that the webserver is exposed on port 8091. Also, modify the contents of the built-in html code in that script file so that it is customized to your liking. Add things like more text, images, links to other pages, etc.. Name the Python script file simple6.py, and complete the 6 steps shown above to make it into a systemd-controlled service. Then run the services simple5 and simple6 simultaneously, and with your favorite Web browser, browse to the ports 8090 and 8091. What do you see there?

The above three examples illustrated how simple Bourne shell script files, and a Python script, can be made into daemon services under the control and monitoring of systemd. The first and second examples illustrated how to make a shell script a daemon by adding a service unit file for it in /etc/systemd/system. The third example illustrated how to make a simple Python script, that runs a Python-based webserver application, into a systemd service.

W27.7.4 systemd Methods of Changing the Activation Behavior of a New-Style Daemon

There are several other “new-style” (systemd-controlled) methods that can be applied to an old-style daemon, to update it to be systemd-compliant and controlled by systemd.

These include the following general ways of achieving that:

Boot-Based Activation

“Old-style” daemons are started when the system boots, and manually on a per-service basis, via BSD-init or SysV-init scripts. This method of activation has been traditionally supported on UNIX and Linux systems (prior to systemd’s installation in the kernel). systemd uses a modernized version of activation, both when the system boots, and at runtime, using the more minimal service description files we have described in the sections above.

Socket-Based Activation

The chief advantage of socket-based activation of daemons is, most importantly, the simplification of configuration and program development.

In socket-based activation, the creation and binding of listening sockets happens in systemd. Using unit and service files for daemon configuration, systemd installs the sockets and then assigns them to the systemd-spawned process when the triggering event happens.

Bus-Based Activation

When the D-Bus IPC system is used, new-style daemons should deploy bus activation so that they are automatically activated when a client application accesses their IPC interfaces.

Device-Based Activation

New-style daemons that manage a particular type of hardware, like a disk volume or ZFS dataset, should be activated only when the hardware of the respective kind is plugged in or otherwise becomes available.

Path-Based Activation

systemd provides a way to bind service activation to file system changes. This is implemented using path-based activation configured in path unit files, as shown in the man page for systemd.path.

Timer-Based Activation

New-style daemons can implement clean-up jobs that are intended to be executed in regular intervals. In systemd, this is implemented via timer unit files as shown in examples above, as described in the man page for systemd.timer.

It is possible, and preferable, for services to be activated by more than one of the above methods.

Examples of this are wifi, Bluetooth, and CUPS printer services, which can be made active when their respective devices are plugged in, or when activity is first seen on a particular port.

We provide a simple example of socket-based activation in Section 7.4.1.

W27.7.4.1 A Simple Sockets-Based Activation Example

Part of the efficiency and speed advantage systemd gives Linux is the ability to delay the start of services and daemons until they are actually needed, instead of running all of them when the system boots and enters the steady operating state.

Sockets-based activation for a daemon, such as a web server, means that when a request is made on a specified port that the web server is hooked to, the web server then starts as a daemon. It then services all requests made on that port.

In this example, we use the special systemd-activate command, which is usually used to test sockets-based activation, to achieve our objectives. systemd-activate “listens” on that port.

For further references, look at the man page on your system for systemd-activate. Also, refer back to the Simple Web Server, Example W19.30 shown in Chapter W19, Section 4.2.1, and the online documentation for this Python module.

Following are steps you can easily take to make a web server application start when a HTTP request is made on a particular port of your choosing. You can change the port numbers shown in the example steps below to any ephemeral ports you want.

1. Change the current working directory to some location in the file system that you want the Python Simple Web Server to display the contents of. It might have an index.html file at that location.

2. Type the following command-

$ **/lib/systemd/systemd-activate -l 2000 -a python3 -m http.server 8096**

Listening on [::]:2000 as 3.

This command awaits a request on port 2000, and then when one comes in (from a web browser, for example), it executes the command shown after the **-a** option to activate the Python3 simple.http web server built-in application. So basically you are using the **systemd-activate** command to activate port 2000, and then coupling Python3 to this activation to run the http.server built-in application on port 8096.

3. To test this activation, with your favorite web browser, set the URL to

http://your\_ip\_address:2000.

You will get an error in the browser. No worries! You just “primed” the systemd-activate program, and it has spawned the simple.http application! You can check this by using the **ps -aux** command in another terminal window at this point.

4. Set your browser’s URL to http://your\_ip\_address:8096 , and press Enter. You have accessed the Python3 http.server on port 8096. To check this, use the **ps -aux** command again. It should now show two processes running: the systemd-activate process, and the Python3 simple.http process which was socket-activated. Plus perhaps a couple of Web browser processes as well.

Whatever is in the directory you set in step 1. above will be displayed. If the directory contains files, it will show you a listing of the files in the directory. And if there is a valid HTTP-formatted index.html file in that directory, you will see its contents displayed in the browser window.

One of the obvious drawbacks of this example is that the simple.http web server does not stop running after requests have stopped coming into it on port 8096. It’s not socket-deactivated. Building a timeout into this simple example would be an interesting exercise, so that the web server is not always running. And, as you can see, the Simple Sockets-Based Activation Example is not a rigorous and thorough explication of socket-based activation, but it gives you the idea behind it using a single, and very simple, systemd command.

Additionally, the above method of using **systemd-activate** is very similar to using the “Swiss Army knife” of Linux networking: ncat. The **ncat** command is part of the nmap suite of tools, which can be installed by using the following two commands on a Debian-family Linux Mint system:

$ **sudo apt-get update**

$ **sudo apt-get install nmap**

To achieve a result similar to systemd-activate, use the following command to launch the webserver2 program, which we developed and illustrated in Chapter 18, Section 5.2 of the printed book. On our Linux Mint system, the program from 18.5.2 was found along the pathname shown-

$ **ncat -l -p 8080 -e /home/bob/webserver2**

The process of using the webserver2 program with this approach proceeds exactly like steps 3. and 4. in the example provided.

In-Chapter Exercises

18. Do the Python3-based example of the webserver shown in this section on your Linux system. Make sure to adjust the current working directory you issue the command from, and the IP addresses and port numbers shown, so that they work with your system. Place a valid index.html file in the current working directory, in order to test the Python3 web server application. Then answer this question-

How can you run more than one Python3-based webserver on several different ephemeral ports , each serving a different LAN, intranet, or public-facing Internet index.html file? Sketch a method for achieving this scenario. In the context of systemd, can these multiple webservers be made into systemd services?

19. Do the ncat-based webserver2 example shown in this section on your Linux system. Make sure to adjust the directory pathname, IP addresses, and port numbers shown, so that they work with your system. Then answer the following questions-

a) If you have not done the Chapter 18, Section 5.2 webserver2 example, how can you launch a webserver with ncat?

b) Could you produce a Python3 script that would in effect launch http.server, and use ncat to run the Python3 script?

c) Using the ncat methodology, how can you run more than one Python3-based webserver on several different ports, each serving a different LAN, intranet, or public-facing Internet index.html file?

d) What other webserver programs are available on your Linux system, such as nginx, apache, etc.? How can they be used with the ncat command?

e) In the context of systemd, can the multiple webservers from your answers to b) and c) be made into systemd services?

W27.8 A GUI for systemd: systemadm

If you prefer a GUI interface to systemd, you can install an app for it via your package management system. For example, on Linux Mint, we used the Software Manager to install it. It is named systemdadm. You can install it by searching for Systemd-ui in the Linux Mint Software Manager.

After the app is installed, the most effective way to run it is to type the following command:

$ **sudo systemadm**

A GUI interface opens on the desktop, similar to Figure W27.5.

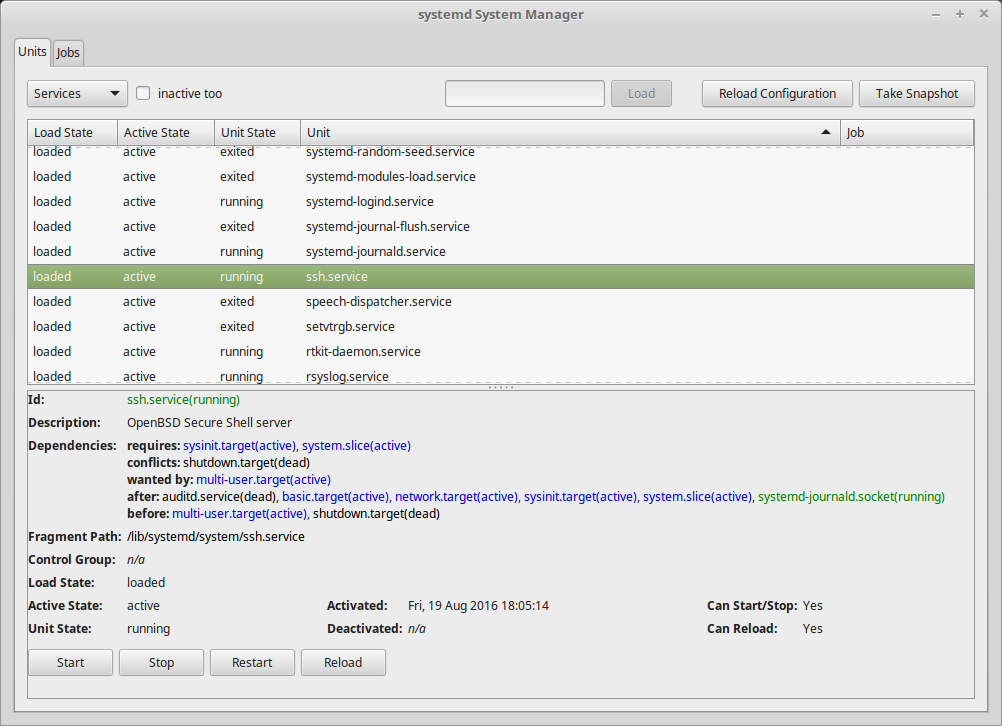


Figure W27.5 systemadm GUI

From the Units tab in the systemadm window, you can choose a pull-down menu option to only show Services, for example. To alphabetize the list of Services, click on the Unit column title heading. Then, you select a service of interest by clicking on it once. On our Linux Mint system, the lower pane needed to be “pulled” upward using the 5 dots shown at the default bottom of the screen display, using the left-most mouse button. With the lower pane pulled up as seen in Figure W27.5, you can use buttons in the lower pane of the GUI window to administrate the service. Notice there are Start, Stop, Restart, and Reload buttons available. This is seen in Figure W27.5 for the ssh service. It is done in a graphical manner that is very similar to the methods shown for administrating services with the text-based commands for systemd we present in this chapter.

In-Chapter Exercise

20. Install systemadm on your Linux system, using the appropriate package management facilitiy, and then use it to verify, and supplement your knowledge of, the systemd commands we showed in Chapter 18, and in this chapter. For example, highlight the ssh.service (or another service of your choice available on your system), as shown in Figure W27.5, on your Linux system. Then answer these questions fully-

a) What are the dependencies of this service? List them completely.

b) What is the path to the service? Start, stop, and restart the service. How do you know that these operations actually accomplished what they’re supposed to do? Give commands that would verify that the buttons actually worked.

c) What does the “Take Snapshot” button in the upper-right corner of the systemadm window do? What systemd facilities and command(s) do the same thing as the button?