opensource.com

Pragmatic systemd

by David Both

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

David Both

David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker who lives in Raleigh North Carolina. He is a strong proponent of and evangelist for the "Linux Philosophy."

David has been in the IT industry for nearly 50 years. He has taught RHCE classes for Red Hat and has worked at MCI Worldcom, Cisco, and the State of North Carolina. He has been working with Linux and Open Source Software for over 20 years.

David prefers to purchase the components and build his own computers from scratch to ensure that each new computer meets his exacting specifications. His primary workstation is an ASUS TUF X299 motherboard and an Intel i9 CPU with 16 cores (32 CPUs) and 64GB of RAM in a ThermalTake Core X9 case.

David has written articles for magazines including, Linux Magazine, Linux Journal. His article "Complete Kickstart," co-authored with a colleague at Cisco, was ranked 9th in the Linux Magazine Top Ten Best System Administration Articles list for 2008. David currently writes prolifically for OpenSource.com and Enable SysAdmin.

David currently has one book published, "The Linux Philosophy for SysAdmins." and is now is working on his next project, “Using and Administering Linux: Zero to SysAdmin,” a self-study training course in three volumes that is scheduled for release in late 2019."

David can be reached at LinuxGeek46@both.org or on Mastodon at @LinuxGeek46@linuxrocks.online

Learning to love systemd

For the record, systemd is spelled in all lower-case, even at the beginning of a sentence. systemd (see?) is a replacement for init and SystemV init scripts. It's also much more.

Like most sysadmins, when I think of the init program and SystemV, I think of Linux startup and shutdown and not really much else, like managing services once they are up and running. Like init, systemd is the mother of all processes, and it is responsible for bringing the Linux host up to a state in which productive work can be done. Some of the functions assumed by systemd, which is far more extensive than the old init program, are to manage many aspects of a running Linux host, including mounting filesystems, managing hardware, handling timers, and starting and managing the system services that are required to have a productive Linux host.

This ebook explores systemd's functions both at startup and beginning after startup finishes. It's an unofficial companion to my three-volume Linux training course, *Using and administering Linux: zero to sysadmin*, and contains great information for advanced users and beginning sysadmins alike.

Linux boot

The complete process that takes a Linux host from an off state to a running state is complex, but it is open and knowable. Before getting into the details, I'll give a quick overview from when the host hardware is turned on until the system is ready for a user to log in. Most of the time, "the boot process" is discussed as a single entity, but that is not accurate. There are, in fact, three major parts to the full boot and startup process:

- **Hardware boot:** Initializes the system hardware
- **Linux boot:** Loads the Linux kernel and then systemd
- **Linux startup:** Where systemd prepares the host for productive work

The Linux startup sequence begins after the kernel has loaded either init or systemd, depending upon whether the distribution uses the old or new startup, respectively. The init and systemd programs start and manage all the other processes and are both known as the "mother of all processes" on their respective systems.

It is important to separate the hardware boot from the Linux boot from the Linux startup and to explicitly define the demarcation points between them. Understanding these differences and what part each plays in getting a Linux system to a state where it can be productive makes it possible to manage these processes and better determine where a problem is occurring during what most people refer to as "boot."

The startup process follows the three-step boot process and brings the Linux computer up to an operational state in which it is usable for productive work. The startup process begins when the kernel transfers control of the host to systemd.

systemd controversy

systemd can evoke a wide range of reactions from sysadmins and others responsible for keeping Linux systems up and running. The fact that systemd is taking over so many tasks in many Linux systems has engendered pushback and discord among certain groups of developers and sysadmins.

SystemV and systemd are two different methods of performing the Linux startup sequence. SystemV start scripts and the init program are the old methods, and systemd using targets is the new method. Although most modern Linux distributions use the newer systemd for startup, shutdown, and process management, there are still some that do not. One reason is that some distribution maintainers and some sysadmins prefer the older SystemV method over the newer systemd.

I think both have advantages.

Why I prefer SystemV

I prefer SystemV because it is more open. Startup is accomplished using Bash scripts. After the kernel starts the init program, which is a compiled binary, init launches the **rc.sysinit** script, which performs many system initialization tasks. After **rc.sysinit** completes, init launches the **/etc/rc.d/rc** script, which in turn starts the various services defined by the SystemV start scripts in the **/etc/rc.d/rcX.d**, where "X" is the number of the runlevel being started.

Except for the init program itself, all these programs are open and easily knowable scripts. It is possible to read through these scripts and learn exactly what is taking place during the entire startup process, but I don't think many sysadmins actually do that. Each start script is numbered so that it starts its intended service in a specific sequence. Services are started serially, and only one service starts at a time.

systemd, developed by Red Hat's Lennart Poettering and Kay Sievers, is a complex system of large, compiled binary executables that are not understandable without access to the source code. It is open source, so "access to the source code" isn't hard, just less convenient. systemd appears to represent a significant refutation of multiple tenets of the Linux philosophy. As a binary, systemd is not directly open for the sysadmin to view or make easy changes. systemd tries to do everything, such as managing running services, while providing significantly more status information than SystemV. It also manages hardware, processes, and groups of processes, filesystem mounts, and much more. systemd is present in almost every aspect of the modern Linux host, making it the one-stop tool for system management. All of this is a clear violation of the tenets that programs should be small and that each program should do one thing and do it well.

Why I prefer systemd

I prefer systemd as my startup mechanism because it starts as many services as possible in parallel, depending upon the current stage in the startup process. This speeds the overall startup and gets the host system to a login screen faster than SystemV.

systemd manages almost every aspect of a running Linux system. It can manage running services while providing significantly more status information than SystemV. It also manages hardware, processes and groups of processes, filesystem mounts, and much more. systemd is present in almost every aspect of the modern Linux operating system, making it the one-stop tool for system management. (Does this sound familiar?)

The systemd tools are compiled binaries, but the tool suite is open because all the configuration files are ASCII text files. Startup configuration can be modified through various GUI and command-line tools, as well as adding or modifying various configuration files to suit the needs of the specific local computing environment.

The real issue

Did you think I could not like both startup systems? I do, and I can work with either one.

In my opinion, the real issue and the root cause of most of the controversy between SystemV and systemd is that there is no choice on the sysadmin level. The choice of whether to use SystemV or systemd has already been made by the developers, maintainers, and packagers of the various distributions—but with good reason. Scooping out and replacing an init system, by its extreme, invasive nature, has a lot of consequences that would be hard to tackle outside the distribution design process.

Despite the fact that this choice is made for me, my Linux hosts boot up and work, which is what I usually care the most about. As an end user and even as a sysadmin, my primary concern is whether I can get my work done, work such as writing my books and this article, installing updates, and writing scripts to automate everything. So long as I can do my work, I don't really care about the start sequence used on my distro.

I do care when there is a problem during startup or service management. Regardless of which startup system is used on a host, I know enough to follow the sequence of events to find the failure and fix it.

Replacing SystemV

There have been previous attempts at replacing SystemV with something a bit more modern. For about two releases, Fedora used a thing called Upstart to replace the aging SystemV, but it did not replace init and provided no changes that I noticed. Because Upstart provided no significant changes to the issues surrounding SystemV, efforts in this direction were quickly dropped in favor of systemd.

Despite the fact that most Linux developers agree that replacing the old SystemV startup is a good idea, many developers and sysadmins dislike systemd for that. Rather than rehash all the so-called issues that people have—or had—with systemd, I will refer you to two good, if somewhat old, articles that should cover most everything. Linus Torvalds, the creator of the Linux kernel, seems disinterested. In a 2014 ZDNet article, *Linus Torvalds and others on Linux's systemd*, Linus is clear about his feelings.

> "I don't actually have any particularly strong opinions on systemd itself. I've had issues with some of the core developers that I think are much too cavalier about bugs and compatibility, and I think some of the design details are insane (I dislike the binary logs, for example), but those are details, not big issues."

In case you don't know much about Linus, I can tell you that if he does not like something, he is very outspoken, explicit, and quite clear about that dislike.

Creative Commons Attribution Share-alike 4.0

In 2013, Poettering wrote a long blog post in which he debunks the myths about systemd while providing insight into some of the reasons for creating it. I highly recommend reading it.

systemd tasks

Depending upon the options used during the compile process (which are not considered in this series), systemd can have as many as 69 binary executables that perform the following tasks, among others:

- The systemd program runs as PID 1 and provides system startup of as many services in parallel as possible, which, as a side effect, speeds overall startup times. It also manages the shutdown sequence.
- The systemctl program provides a user interface for service management.
- Support for SystemV and LSB start scripts is offered for backward compatibility.
- Service management and reporting provide more service status data than SystemV.
- It includes tools for basic system configuration, such as hostname, date, locale, lists of logged-in users, running containers and virtual machines, system accounts, runtime directories and settings, daemons to manage simple network configuration, network time synchronization, log forwarding, and name resolution.
- It offers socket management.
- systemd timers provide advanced cron-like capabilities to include running a script at times relative to system boot, systemd startup, the last time the timer was started, and more.
- It provides a tool to analyze dates and times used in timer specifications.
- Mounting and unmounting of filesystems with hierarchical awareness allows safer cascading of mounted filesystems.
- It enables the positive creation and management of temporary files, including deletion.
- An interface to D-Bus provides the ability to run scripts when devices are plugged in or removed. This allows all devices, whether pluggable or not, to be treated as plug-and-play, which considerably simplifies device handling.
- Its tool to analyze the startup sequence can be used to locate the services that take the most time.
- It includes journals for storing system log messages and tools for managing the journals.

Architecture

Those tasks and more are supported by a number of daemons, control programs, and configuration files. Figure 1 shows many of the components that belong to systemd. This is a simplified diagram designed to provide a high-level overview, so it does not include all of the individual programs or files. Nor does it provide any insight into data flow, which is so complex that it would be a useless exercise in the context of this series of articles.

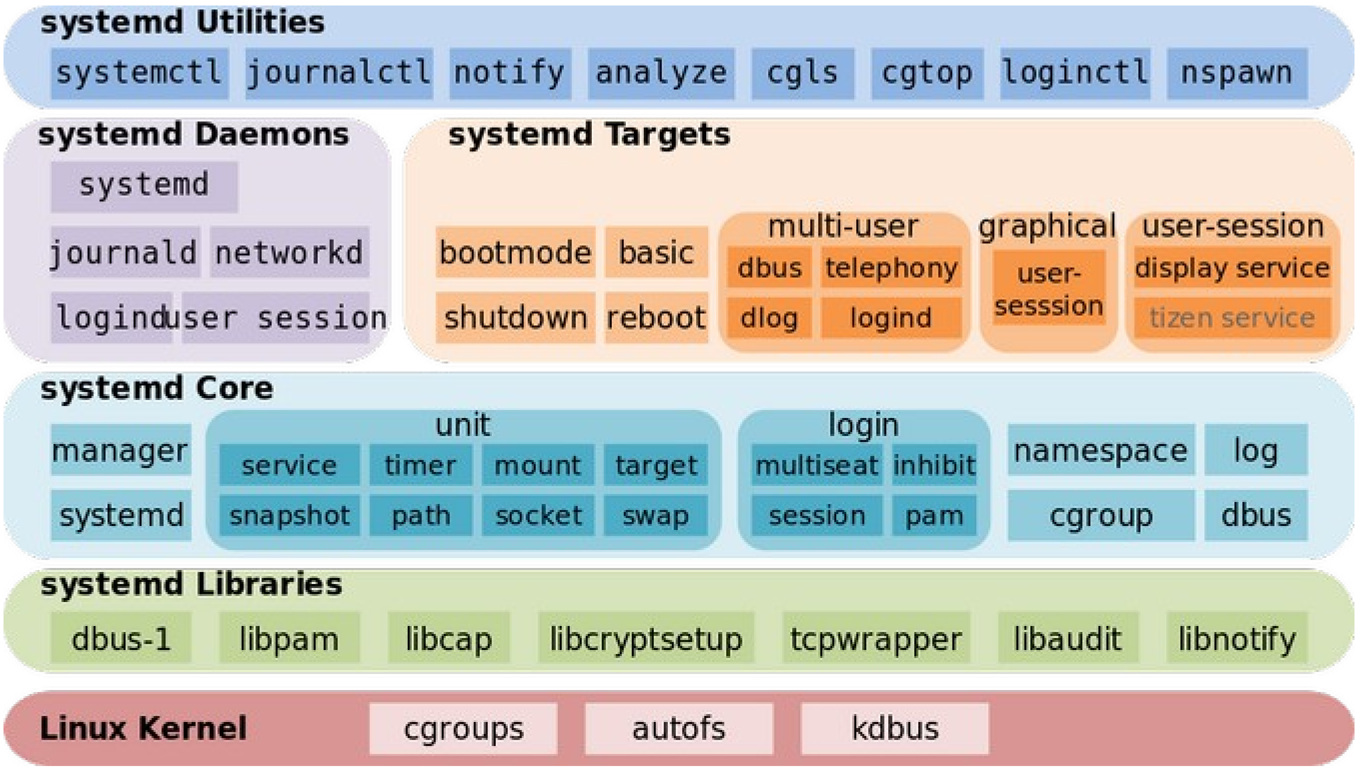


Fig 1: Architecture of systemd, by Shmuel Csaba Otto Traian (CC BY-SA 3.0)

A full exposition of systemd would take a book on its own. You do not need to understand the details of how the systemd components in Figure 1 fit together; it's enough to know about the programs and components that enable managing various Linux services and deal with log files and journals. But it's clear that systemd is not the monolithic monstrosity it is purported to be by some of its critics.

systemd as PID 1

systemd is PID 1. Some of its functions, which are far more extensive than the old SystemV3 init program, are to manage many aspects of a running Linux host, including mounting filesystems and starting and managing system services required to have a productive Linux host. Any of systemd's tasks that are not related to the startup sequence are outside the scope of this article (but some will be explored later in this series).

First, systemd mounts the filesystems defined by **/etc/fstab**, including any swap files or partitions. At this point, it can access the configuration files located in **/etc**, including its own. It uses its configuration link, **/etc/systemd/system/default.target**, to determine which state or target it should boot the host into. The **default.target** file is a symbolic link to the true target file. For a desktop workstation, this is typically going to be the **graphical.target**, which is equivalent to runlevel 5 in SystemV. For a server, the default is more likely to be the **multi-user.target**, which is like runlevel 3 in SystemV. The **emergency.target** is similar to single-user mode. Targets and services are systemd units.

The table below (Figure 2) compares the systemd targets with the old SystemV startup runlevels. systemd provides the systemd target aliases for backward compatibility. The target aliases allow scripts—and many sysadmins—to use SystemV commands like **init 3** to change runlevels. Of course, the SystemV commands are forwarded to systemd for interpretation and execution.

systemd targets	**SystemV runlevel**	**target aliases**	**Description**
default.target			This target is always aliased with a symbolic link to either **multi-user.target** or **graphical.target**. systemd always uses the **default.target** to start the system. The **default.target** should never be aliased to **halt.target**, **poweroff.target**, or **reboot.target**.
graphical.target	5	runlevel5.target	**Multi-user.target** with a GUI
	4	runlevel4.target	Unused. Runlevel 4 was identical to runlevel 3 in the SystemV world. This target could be created and customized to start local services without changing the default **multi-user.target**.
multi-user.target	3	runlevel3.target	All services running, but command-line interface (CLI) only
	2	runlevel2.target	Multi-user, without NFS, but all other non-GUI services running
rescue.target	1	runlevel1.target	A basic system, including mounting the filesystems with only the most basic services running and a rescue shell on the main console
emergency.target	S		Single-user mode—no services are running; filesystems are not mounted. This is the most basic level of operation with only an emergency shell

systemd targets	SystemV runlevel	target aliases	Description
			running on the main console for the user to interact with the system.
halt.target			Halts the system without powering it down
reboot.target	6	runlevel6.target	Reboot
poweroff.target	0	runlevel0.target	Halts the system and turns the power off

Fig. 2: Comparison of SystemV runlevels with systemd targets and some target aliases

Each target has a set of dependencies described in its configuration file. systemd starts the required dependencies, which are the services required to run the Linux host at a specific level of functionality. When all the dependencies listed in the target configuration files are loaded and running, the system is running at that target level. In Figure 2, the targets with the most functionality are at the top of the table, with functionality declining towards the bottom of the table.

systemd also looks at the legacy SystemV init directories to see if any startup files exist there. If so, systemd uses them as configuration files to start the services described by the files. The deprecated network service is a good example of one that still uses SystemV startup files in Fedora.

Figure 3 is copied directly from the bootup man page. It shows a map of the general sequence of events during systemd startup and the basic ordering requirements to ensure a successful startup.

```
                                         cryptsetup-pre.target
                                                   |
 (various low-level                                v
     API VFS mounts:             (various cryptsetup devices...)
  mqueue, configfs,                                |    |
  debugfs, ...)                                    v    |
  |                                  cryptsetup.target  |
  |  (various swap                                 |    |    remote-fs-pre.target
  |   devices...)                                  |    |     |        |
  |    |                                           |    |     |        v
  |    v                       local-fs-pre.target |    |     |  (network file systems)
  |  swap.target                       |           |    v     v                 |
  |    |                               v           |  remote-cryptsetup.target  |
  |    |  (various low-level  (various mounts and  |             |              |
  |    |   services: udevd,    fsck services...)   |             |    remote-fs.target
  |    |   tmpfiles, random            |           |             |             /
  |    |   seed, sysctl, ...)          v           |             |            /
  |    |      |                 local-fs.target    |             |           /
  |    |      |                        |           |             |          /
  \____|______|_______________   ______|___________/             |         /
                              \ /                                |        /
                               v                                 |       /
                        sysinit.target                           |      /
                               |                                 |     /
        ______________________/|\_____________________           |    /
       /              |        |      |               \          |   /
       |              |        |      |               |          |  /
       v              v        |      v               |          | /
  (various       (various      |  (various            |          |/
   timers...)      paths...)   |   sockets...)        |          |
       |              |        |      |               |          |
       v              v        |      v               |          |
 timers.target  paths.target   |  sockets.target      |          |
       |              |        |      |               v          |
       v              \_______ | _____/         rescue.service   |
                              \|/                     |          |
                               v                      v          |
                           basic.target         rescue.target    |
                               |                                 |
                       ________v____________________             |
                      /              |              \            |
                      |              |              |            |
                      v              v              v            |
                  display-    (various system   (various system  |
              manager.service     services        services)      |
                      |         required for        |            |
                      |        graphical UIs)       v            v
                      |              |            multi-user.target
 emergency.service    |              |              |
         |            \_____________ | _____________/
         v                          \|/
 emergency.target                    v
                              graphical.target
```

Fig 3: The systemd startup map

Creative Commons Attribution Share-alike 4.0

The **sysinit.target** and **basic.target** targets can be considered checkpoints in the startup process. Although one of systemd's design goals is to start system services in parallel, certain services and functional targets must be started before other services and targets can start. These checkpoints cannot be passed until all of the services and targets required by that checkpoint are fulfilled.

The **sysinit.target** is reached when all of the units it depends on are completed. All of those units, mounting filesystems, setting up swap files, starting udev, setting the random generator seed, initiating low-level services, and setting up cryptographic services (if one or more filesystems are encrypted), must be completed but, within the **sysinit.target**, those tasks can be performed in parallel.

The **sysinit.target** starts up all of the low-level services and units required for the system to be marginally functional and that are required to enable moving onto the **basic.target**.

After the **sysinit.target** is fulfilled, systemd then starts all the units required to fulfill the next target. The basic target provides some additional functionality by starting units that are required for all of the next targets. These include setting up things like paths to various executable directories, communication sockets, and timers.

Finally, the user-level targets, **multi-user.target** or **graphical.target**, can be initialized. The **multi-user.target** must be reached before the graphical target dependencies can be met. The underlined targets in Figure 3 are the usual startup targets. When one of these targets is reached, startup has completed. If the **multi-user.target** is the default, then you should see a text-mode login on the console. If **graphical.target** is the default, then you should see a graphical login; the specific GUI login screen you see depends on your default display manager.

The bootup man page also describes and provides maps of the boot into the initial RAM disk and the systemd shutdown process.

systemd also provides a tool that lists dependencies of a complete startup or for a specified unit. A unit is a controllable systemd resource entity that can range from a specific service, such as httpd or sshd, to timers, mounts, sockets, and more. Try the following command and scroll through the results.

```
systemctl list-dependencies graphical.target
```

Notice that this fully expands the top-level target units list required to bring the system up to the graphical target run mode. Use the **--all** option to expand all of the other units as well.

```
systemctl list-dependencies --all graphical.target
```

You can search for strings such as "target," "slice," and "socket" using the search tools of the **less** command.

Try these commands:

```
systemctl list-dependencies multi-user.target

systemctl list-dependencies rescue.target

systemctl list-dependencies local-fs.target

systemctl list-dependencies dbus.service
```

This tool helps me visualize the specifics of the startup dependencies for the host I am working on. Go ahead and spend some time exploring the startup tree for one or more of your Linux hosts. But be careful because the systemctl man page contains this note:

> "Note that this command only lists units currently loaded into memory by the service manager. In particular, this command is not suitable to get a comprehensive list at all reverse dependencies on a specific unit, as it won't list the dependencies declared by units currently not loaded."

Final thoughts

Even before getting very deep into systemd, it's obvious that it is both powerful and complex. It is also apparent that systemd is not a single, huge, monolithic, and unknowable binary file. Rather, it is composed of a number of smaller components and subcommands that are designed to perform specific tasks.

Resources

There's a series of deeply technical articles for Linux sysadmins by Lennart Poettering, the designer and primary developer of systemd. These articles were written between April 2010 and September 2011, but they're just as relevant now as they were then. Much of everything else good that has been written about systemd and its ecosystem is based on these papers.

- Rethinking PID 1
- systemd for Administrators, Part I

- systemd for Administrators, Part II
- systemd for Administrators, Part III
- systemd for Administrators, Part IV
- systemd for Administrators, Part V
- systemd for Administrators, Part VI
- systemd for Administrators, Part VII
- systemd for Administrators, Part VIII
- systemd for Administrators, Part IX
- systemd for Administrators, Part X
- systemd for Administrators, Part XI

Understanding systemd at startup

In the previous chapter, I looked at systemd's functions and architecture and the controversy around its role as a replacement for the old SystemV init program and startup scripts. In this chapter, I explore the files and tools that manage the Linux startup sequence. I'll explain the systemd startup sequence, how to change the default startup target (runlevel in SystemV terms), and how to manually switch to a different target without going through a reboot.

I'll also look at two important systemd tools. The first is the **systemctl** command, which is the primary means of interacting with and sending commands to systemd. The second is **journalctl**, which provides access to the systemd journals that contain huge amounts of system history data such as kernel and service messages (both informational and error messages).

Be sure to use a non-production system for testing and experimentation in this and future articles. Your test system needs to have a GUI desktop (such as Xfce, LXDE, Gnome, KDE, or another) installed.

Exploring Linux startup with systemd

Before you can observe the startup sequence, you need to do a couple of things to make the boot and startup sequences open and visible. Normally, most distributions use a startup animation or splash screen to hide the detailed messages that would otherwise be displayed during a Linux host's startup and shutdown. This is called the Plymouth boot screen on Red Hat-based distros. Those hidden messages can provide a great deal of information about startup and shutdown to a sysadmin looking for information to troubleshoot a bug or to just learn about the startup sequence. You can change this using the GRUB (Grand Unified Boot Loader) configuration.

The main GRUB configuration file is **/boot/grub2/grub.cfg**, but, because this file can be overwritten when the kernel version is updated, you do not want to change it. Instead, modify the **/etc/default/grub** file, which is used to modify the default settings of **grub.cfg**.

Start by looking at the current, unmodified version of the **/etc/default/grub** file:

```
# cd /etc/default ; cat grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/fedora_testvm1-swap rd.lvm.
lv=fedora_testvm1/root rd.lvm.lv=fedora_testvm1/swap rd.lvm.lv=fedora_
testvm1/usr rhgb quiet"
GRUB_DISABLE_RECOVERY="true"
```

Chapter 6 of the GRUB documentation contains a list of all the possible entries in the **/etc/default/grub** file, but I focus on the following:

- I change **GRUB_TIMEOUT**, the number of seconds for the GRUB menu countdown, from five to 10 to give a bit more time to respond to the GRUB menu before the countdown hits zero.
- I delete the last two parameters on **GRUB_CMDLINE_LINUX**, which lists the command-line parameters that are passed to the kernel at boot time. One of these parameters, **rhgb** stands for Red Hat Graphical Boot, and it displays the little Fedora icon animation during the kernel initialization instead of showing boot-time messages. The other, the **quiet** parameter, prevents displaying the startup messages that document the progress of the startup and any errors that occur. I delete both **rhgb** and **quiet** because sysadmins need to see these messages. If something goes wrong during boot, messages on the screen can point to the cause of the problem.

After you make these changes, your GRUB file will look like:

```
# cat grub
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/fedora_testvm1-swap rd.lvm.
lv=fedora_testvm1/root rd.lvm.lv=fedora_testvm1/swap rd.lvm.lv=fedora_
testvm1/usr"
GRUB_DISABLE_RECOVERY="false"
```

Creative Commons Attribution Share-alike 4.0

The **grub2-mkconfig** program generates the **grub.cfg** configuration file using the contents of the **/etc/default/grub** file to modify some of the default GRUB settings. The **grub2-mkconfig** program sends its output to **STDOUT**. It has a **-o** option that allows you to specify a file to send the datastream to, but it is just as easy to use redirection. Run the following command to update the **/boot/grub2/grub.cfg** configuration file:

```
# grub2-mkconfig > /boot/grub2/grub.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.18.9-200.fc28.x86_64
Found initrd image: /boot/initramfs-4.18.9-200.fc28.x86_64.img
Found linux image: /boot/vmlinuz-4.17.14-202.fc28.x86_64
Found initrd image: /boot/initramfs-4.17.14-202.fc28.x86_64.img
Found linux image: /boot/vmlinuz-4.16.3-301.fc28.x86_64
Found initrd image: /boot/initramfs-4.16.3-301.fc28.x86_64.img
Found linux image: /boot/vmlinuz-0-rescue-7f12524278bd40e9b10a085bc82dc504
Found initrd image: /boot/initramfs-0-rescue-7f12524278bd40e9b10a085bc82dc504.img
done
```

Reboot your test system to view the startup messages that would otherwise be hidden behind the Plymouth boot animation. But what if you need to view the startup messages and have not disabled the Plymouth boot animation? Or you have, but the messages stream by too fast to read? (Which they do.)

There are a couple of options, and both involve log files and systemd journals—which are your friends. You can use the **less** command to view the contents of the **/var/log/messages** file. This file contains boot and startup messages as well as messages generated by the operating system during normal operation. You can also use the **journalctl** command without any options to view the systemd journal, which contains essentially the same information:

```
# journalctl
-- Logs begin at Sat 2020-01-11 21:48:08 EST, end at Fri 2020-04-03 08:54:30 EDT.
--
Jan 11 21:48:08 f31vm.both.org kernel: Linux version 5.3.7-301.fc31.x86_64
(mockbuild@bkernel03.phx2.fedoraproject.org) (gcc version 9.2.1 20190827 (Red Hat
9.2.1-1) (GCC)) #1 SMP Mon Oct >
Jan 11 21:48:08 f31vm.both.org kernel: Command line:
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.3.7-301.fc31.x86_64 root=/dev/mapper/VG01-root
ro resume=/dev/mapper/VG01-swap rd.lvm.lv=VG01/root rd>
Jan 11 21:48:08 f31vm.both.org kernel: x86/fpu: Supporting XSAVE feature 0x001:
'x87 floating point registers'
Jan 11 21:48:08 f31vm.both.org kernel: x86/fpu: Supporting XSAVE feature 0x002:
'SSE registers'
[...]
```

I truncated this datastream because it can be hundreds of thousands or even millions of lines long. (The journal listing on my primary workstation is 1,188,482 lines long.) Be sure to try this on your test system. If it has been running for some time—even if it has been rebooted many times—huge amounts of data will be displayed. Explore this journal data. It contains a lot of information that can be very useful when doing problem determination. Knowing what this data looks like for a normal boot and startup can help you locate problems when they occur.

I will discuss systemd journals, the **journalctl** command, and how to sort through all of that data to find what you want in more detail later in this book.

After GRUB loads the kernel into memory, it must first extract itself from the compressed version of the file before it can perform any useful work. After the kernel has extracted itself and started running, it loads systemd and turns control over to it.

This is the end of the boot process. At this point, the Linux kernel and systemd are running but unable to perform any productive tasks for the end user because nothing else is running, there's no shell to provide a command line, no background processes to manage the network or other communication links, and nothing that enables the computer to perform any productive function.

systemd can now load the functional units required to bring the system up to a selected target run state.

Targets

A systemd target represents a Linux system's current or desired run state. Much like SystemV start scripts, targets define the services that must be present for the system to run and be active in that state. Figure 1 shows the possible run-state targets of a Linux system using systemd. As seen in the first article of this series and in the systemd bootup man page (man bootup), there are other intermediate targets that are required to enable various necessary services. These can include **swap.target**, **timers.target**, **local-fs.target**, and more. Some targets (like **basic.target**) are used as checkpoints to ensure that all the required services are up and running before moving on to the next-higher level target.

Unless otherwise changed at boot time in the GRUB menu, systemd always starts the **default.target**. The **default.target** file is a symbolic link to the true target file. For a desktop workstation, this is typically going to be the **graphical.target**, which is equivalent to runlevel 5 in SystemV. For a server, the default is more likely to be the **multi-user.target**, which is like runlevel 3 in SystemV. The **emergency.target** file is similar to single-user mode. Targets and services are systemd units.

The following table, which I included in the previous article in this series, compares the systemd targets with the old SystemV startup runlevels. The systemd target aliases are provided by systemd for backward compatibility. The target aliases allow scripts—and sysadmins—to use SystemV commands like **init 3** to change runlevels. Of course, the SystemV commands are forwarded to systemd for interpretation and execution.

systemd targets	**SystemV runlevel**	**target aliases**	**Description**
default.target			This target is always aliased with a symbolic link to either **multi-user.target** or **graphical.target**. systemd always uses the **default.target** to start the system. The **default.target** should never be aliased to **halt.target**, **poweroff.target**, or **reboot.target**.
graphical.target	5	runlevel5.target	**Multi-user.target** with a GUI
	4	runlevel4.target	Unused. Runlevel 4 was identical to runlevel 3 in the SystemV world. This target could be created and customized to start local services without changing the default **multi-user.target**.
multi-user.target	3	runlevel3.target	All services running, but command-line interface (CLI) only
	2	runlevel2.target	Multi-user, without NFS, but all other non-GUI services running
rescue.target	1	runlevel1.target	A basic system, including mounting the filesystems with only the most basic services running and a rescue shell on the main console
emergency.target	S		Single-user mode—no services are running; filesystems are not mounted. This is the most basic level of operation with only an emergency shell running on the main console for the user to interact with the system.
halt.target			Halts the system without powering it down
reboot.target	6	runlevel6.target	Reboot
poweroff.target	0	runlevel0.target	Halts the system and turns the power off

Fig. 1: Comparison of SystemV runlevels with systemd targets and target aliases.

Each target has a set of dependencies described in its configuration file. systemd starts the required dependencies, which are the services required to run the Linux host at a specific level of functionality. When all of the dependencies listed in the target configuration files are loaded and running, the system is running at that target level. If you want, you can review the systemd startup sequence and runtime targets in the first article in this series, *Learning to love systemd*.

Exploring the current target

Many Linux distributions default to installing a GUI desktop interface so that the installed systems can be used as workstations. I always install from a Fedora Live boot USB drive with an Xfce or LXDE desktop. Even when installing a server or other infrastructure type of host (such as the ones I use for routers and firewalls), I use one of these installations that installs a GUI desktop.

I could install a server without a desktop (and that would be typical for data centers), but that does not meet my needs. It is not that I need the GUI desktop itself, but the LXDE installation includes many of the other tools I use that are not in a default server installation. This means less work for me after the initial installation.

But just because I have a GUI desktop does not mean it makes sense to use it. I have a 16-port KVM that I can use to access the KVM interfaces of most of my Linux systems, but the vast majority of my interaction with them is via a remote SSH connection from my primary workstation. This way is more secure and uses fewer system resources to run **multi-user.target** compared to **graphical.target.**

To begin, check the default target to verify that it is the **graphical.target**:

```
# systemctl get-default
graphical.target
```

Now verify the currently running target. It should be the same as the default target. You can still use the old method, which displays the old SystemV runlevels. Note that the previous runlevel is on the left; it is **N** (which means None), indicating that the runlevel has not changed since the host was booted. The number 5 indicates the current target, as defined in the old SystemV terminology:

```
# runlevel
N 5
```

Note that the runlevel man page indicates that runlevels are obsolete and provides a conversion table.

Creative Commons Attribution Share-alike 4.0

You can also use the systemd method. There is no one-line answer here, but it does provide the answer in systemd terms:

```
# systemctl list-units --type target
UNIT                   LOAD   ACTIVE SUB    DESCRIPTION
basic.target           loaded active active Basic System
cryptsetup.target      loaded active active Local 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
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
rpc_pipefs.target      loaded active active rpc_pipefs.target
slices.target          loaded active active Slices
sockets.target         loaded active active Sockets
sshd-keygen.target     loaded active active sshd-keygen.target
swap.target            loaded active active Swap
sysinit.target         loaded active active System Initialization
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.

21 loaded units listed. Pass --all to see loaded but inactive units, too.
To show all installed unit files use 'systemctl list-unit-files'.
```

This shows all of the currently loaded and active targets. You can also see the **graphical.target** and the **multi-user.target**. The **multi-user.target** is required before the **graphical.target** can be loaded. In this example, the **graphical.target** is active.

Switching to a different target

Making the switch to the **multi-user.target** is easy:

```
# systemctl isolate multi-user.target
```

The display now changes from the GUI desktop or login screen to a virtual console. Log in and list the currently active systemd units to verify that **graphical.target** is no longer running:

```
# systemctl list-units --type target
```

Be sure to use the **runlevel** command to verify that it shows both previous and current "runlevels":

```
# runlevel
5 3
```

Changing the default target

Now, change the default target to the **multi-user.target** so that it will always boot into the **multi-user.target** for a console command-line interface rather than a GUI desktop interface. As the root user on your test host, change to the directory where the systemd configuration is maintained and do a quick listing:

```
# cd /etc/systemd/system/ ; ll
drwxr-xr-x. 2 root root 4096 Apr 25  2018  basic.target.wants
<snip>
lrwxrwxrwx. 1 root root   36 Aug 13 16:23  default.target ->
/lib/systemd/system/graphical.target
lrwxrwxrwx. 1 root root   39 Apr 25  2018  display-manager.service ->
/usr/lib/systemd/system/lightdm.service
drwxr-xr-x. 2 root root 4096 Apr 25  2018  getty.target.wants
drwxr-xr-x. 2 root root 4096 Aug 18 10:16  graphical.target.wants
drwxr-xr-x. 2 root root 4096 Apr 25  2018  local-fs.target.wants
drwxr-xr-x. 2 root root 4096 Oct 30 16:54  multi-user.target.wants
[...]
```

I shortened this listing to highlight a few important things that will help explain how systemd manages the boot process. You should be able to see the entire list of directories and links on your virtual machine.

The **default.target** entry is a symbolic link (symlink, soft link) to the directory **/lib/systemd/system/graphical.target**. List that directory to see what else is there:

```
# ll /lib/systemd/system/ | less
```

You see files, directories, and more links in this listing, but look specifically for **multi-user.target** and **graphical.target**. Now display the contents of **default.target**, which is a link to **/lib/systemd/system/graphical.target**:

```
# cat default.target
#  SPDX-License-Identifier: LGPL-2.1+
#
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.

[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

This link to the **graphical.target** file describes all of the prerequisites and requirements that the graphical user interface requires. I explore these options later in this book.

To enable the host to boot to multi-user mode, delete the existing link and create a new one that points to the correct target:

```
# cd /etc/systemd/system
# rm -f default.target
# ln -s /lib/systemd/system/multi-user.target default.target
```

List the **default.target** link to verify that it links to the correct file:

```
# ll default.target
lrwxrwxrwx 1 root root 37 Nov 28 16:08 default.target ->
/lib/systemd/system/multi-user.target
```

If your link does not look exactly like this, delete it and try again. List the content of the **default.target** link:

```
# cat default.target
#  SPDX-License-Identifier: LGPL-2.1+
#
#  This file is part of systemd.
#
```

```
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.

[Unit]
Description=Multi-User System
Documentation=man:systemd.special(7)
Requires=basic.target
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
AllowIsolate=yes
```

The **default.target**—which is really a link to the **multi-user.target** at this point—now has different requirements in the **[Unit]** section. It does not require the graphical display manager.

Reboot. Your virtual machine should boot to the console login for virtual console 1, which is identified on the display as tty1. Now that you know how to change the default target, change it back to the **graphical.target** using a command designed for the purpose.

First, check the current default target:

```
# systemctl get-default
multi-user.target
# systemctl set-default graphical.target
Removed /etc/systemd/system/default.target.
Created symlink /etc/systemd/system/default.target →
/usr/lib/systemd/system/graphical.target
```

Enter the following command to go directly to the **graphical.target** and the display manager login page without having to reboot:

```
# systemctl isolate default.target
```

I do not know why the term "isolate" was chosen for this sub-command by systemd's developers. My research indicates that it may refer to running the specified target but "isolating" and terminating all other targets that are not required to support the target. However, the effect is to switch targets from one run target to another—in this case, from the multi-user target to the graphical target. The command above is equivalent to the old init 5 command in SystemV start scripts and the init program.

Log into the GUI desktop, and verify that it's working.

Using the systemctl command to manage systemd units

In the previous chapters, I explored the Linux systemd startup sequence. In the first, I looked at systemd's functions and architecture and the controversy around its role as a replacement for the old SystemV init program and startup scripts. And in the second, I examined two important systemd tools, systemctl and journalctl, and explained how to switch from one target to another and to change the default target.

In this chapter, I look at systemd units in more detail, and how to use the **systemctl** command to explore and manage units. I also explain how to stop and disable units and how to create a new systemd mount unit to mount a new filesystem and enable it to initiate during startup.

Preparation

All of the experiments in this chapter should be done as the root user (unless otherwise specified). Some of the commands that simply list various systemd units can be performed by non-root users, but the commands that make changes cannot. Make sure to do all of these experiments only on non-production hosts or virtual machines (VMs).

One of these experiments requires the sysstat package, so install it before you move on. For Fedora and other Red Hat-based distributions you can install sysstat with:

```
dnf -y install sysstat
```

The sysstat RPM installs several statistical tools that can be used for problem determination. One is System Activity Report (SAR), which records many system performance data points at regular intervals (every 10 minutes by default). Rather than run as a daemon in the background, the sysstat package installs two systemd timers. One timer runs every 10 minutes to collect data, and the other runs once a day to aggregate the daily data. In this article, I will look briefly at these timers but wait to explain how to create a timer in a future article.

Creative Commons Attribution Share-alike 4.0

systemd suite

The fact is, systemd is more than just one program. It is a large suite of programs all designed to work together to manage nearly every aspect of a running Linux system. A full exposition of systemd would take a book on its own. Most of us do not need to understand all of the details about how all of systemd's components fit together, so I will focus on the programs and components that enable you to manage various Linux services and deal with log files and journals.

Practical structure

The structure of systemd—outside of its executable files—is contained in its many configuration files. Although these files have different names and identifier extensions, they are all called "unit" files. Units are the basis of everything systemd.

Unit files are ASCII plain-text files that are accessible to and can be created or modified by a sysadmin. There are a number of unit file types, and each has its own man page. Figure 1 lists some of these unit file types by their filename extensions and a short description of each.

systemd unit	Description
.automount	The **.automount** units are used to implement on-demand (i.e., plug and play) and mounting of filesystem units in parallel during startup.
.device	The **.device** unit files define hardware and virtual devices that are exposed to the sysadmin in the **/dev/directory**. Not all devices have unit files; typically, block devices such as hard drives, network devices, and some others have unit files.
.mount	The **.mount** unit defines a mount point on the Linux filesystem directory structure.
.scope	The **.scope** unit defines and manages a set of system processes. This unit is not configured using unit files, rather it is created programmatically. Per the **systemd.scope** man page, “The main purpose of scope units is grouping worker processes of a system service for organization and for managing resources.”
.service	The **.service** unit files define processes that are managed by systemd. These include services such as crond cups (Common Unix Printing System), iptables, multiple logical volume management (LVM) services, NetworkManager, and more.
.slice	The **.slice** unit defines a “slice,” which is a conceptual division of system resources that are related to a group of processes. You can think of all system resources as a pie and this subset of resources as a “slice” out of that pie.
.socket	The **.socket** units define interprocess communication sockets, such as network sockets.
.swap	The **.swap** units define swap devices or files.

.target	The **.target** units define groups of unit files that define startup synchronization points, runlevels, and services. Target units define the services and other units that must be active in order to start successfully.
.timer	The **.timer** unit defines timers that can initiate program execution at specified times.

Fig. 1: Some systemd unit file types

systemctl

The **systemctl** command is used to start and stop services, configure them to launch (or not) at system startup, and monitor the current status of running services.

In a terminal session with root priviledges, list all of the loaded and active systemd units. systemctl automatically pipes its stdout data stream through the **less** pager, so you don't have to:

```
# systemctl
UNIT                                        LOAD   ACTIVE SUB       DESCRIPTION
proc-sys-fs-binfmt_misc.automount          loaded active running   Arbitrary
Executable File>
sys-devices-pci0000:00-0000:00:01.1-ata7-host6-target6:0:0-6:0:0:0-block-
sr0.device loaded a>
sys-devices-pci0000:00-0000:00:03.0-net-enp0s3.device loaded active plugged
82540EM Gigabi>
sys-devices-pci0000:00-0000:00:05.0-sound-card0.device loaded active plugged
82801AA AC'97>
sys-devices-pci0000:00-0000:00:08.0-net-enp0s8.device loaded active plugged
82540EM Gigabi>
sys-devices-pci0000:00-0000:00:0d.0-ata1-host0-target0:0:0-0:0:0:0-block-sda-
sda1.device loa>
sys-devices-pci0000:00-0000:00:0d.0-ata1-host0-target0:0:0-0:0:0:0-block-sda-
sda2.device loa>
[...]

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.

206 loaded units listed. Pass --all to see loaded but inactive units, too.
To show all installed unit files use 'systemctl list-unit-files'.
```

As you scroll through the data in your terminal session, look for some specific things. The first section lists devices such as hard drives, sound cards, network interface cards, and TTY devices. Another section shows the filesystem mount points. Other sections include various services and a list of all loaded and active targets.

Creative Commons Attribution Share-alike 4.0

The sysstat timers at the bottom of the output are used to collect and generate daily system activity summaries for SAR. SAR is a very useful problem-solving tool. (You can learn more about it in Chapter 13 of my book *Using and Administering Linux: Volume 1, Zero to SysAdmin: Getting Started*.)

Near the very bottom, three lines describe the meanings of the statuses (loaded, active, and sub). Press **q** to exit the pager.

Use the following command (as suggested in the last line of the output above) to see all the units that are installed, whether or not they are loaded. I won't reproduce the output here, because you can scroll through it on your own. The systemctl program has an excellent tab-completion facility that makes it easy to enter complex commands without needing to memorize all the options:

```
# systemctl list-unit-files
```

You can see that some units are disabled. Table 1 in the man page for systemctl lists and provides short descriptions of the entries you might see in this listing. Use the **-t** (type) option to view just the timer units:

```
# systemctl list-unit-files -t timer
UNIT FILE                     STATE
chrony-dnssrv@.timer          disabled
dnf-makecache.timer           enabled
fstrim.timer                  disabled
logrotate.timer               disabled
logwatch.timer                disabled
mdadm-last-resort@.timer      static
mlocate-updatedb.timer        enabled
sysstat-collect.timer         enabled
sysstat-summary.timer         enabled
systemd-tmpfiles-clean.timer static
unbound-anchor.timer          enabled
```

You can do the same thing with this alternative, which provides considerably more detail:

```
# systemctl list-timers
Thu 2020-04-16 09:06:20 EDT  3min 59s left n/a                          n/a
systemd-tmpfiles-clean.timer systemd-tmpfiles-clean.service
Thu 2020-04-16 10:02:01 EDT  59min left    Thu 2020-04-16 09:01:32 EDT  49s ago
dnf-makecache.timer          dnf-makecache.service
Thu 2020-04-16 13:00:00 EDT  3h 57min left n/a                          n/a
sysstat-collect.timer        sysstat-collect.service
```

```
Fri 2020-04-17 00:00:00 EDT  14h left      Thu 2020-04-16 12:51:37 EDT  3h 49min
left mlocate-updatedb.timer       mlocate-updatedb.service
Fri 2020-04-17 00:00:00 EDT  14h left      Thu 2020-04-16 12:51:37 EDT  3h 49min
left unbound-anchor.timer         unbound-anchor.service
Fri 2020-04-17 00:07:00 EDT  15h left      n/a                          n/a
sysstat-summary.timer        sysstat-summary.service

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

Although there is no option to do systemctl list-mounts, you can list the mount point unit files:

```
# systemctl list-unit-files -t mount
UNIT FILE                     STATE
-.mount                       generated
boot.mount                    generated
dev-hugepages.mount           static
dev-mqueue.mount              static
home.mount                    generated
proc-fs-nfsd.mount            static
proc-sys-fs-binfmt_misc.mount disabled
run-vmblock\x2dfuse.mount     disabled
sys-fs-fuse-connections.mount static
sys-kernel-config.mount       static
sys-kernel-debug.mount        static
tmp.mount                     generated
usr.mount                     generated
var-lib-nfs-rpc_pipefs.mount  static
var.mount                     generated

15 unit files listed.
```

The STATE column in this data stream is interesting and requires a bit of explanation. The "generated" states indicate that the mount unit was generated on the fly during startup using the information in **/etc/fstab**. The program that generates these mount units is **/lib/systemd/system-generators/systemd-fstab-generator,** along with other tools that generate a number of other unit types. The "static" mount units are for filesystems like **/proc** and **/sys**, and the files for these are located in the **/usr/lib/systemd/system** directory.

Now look at the service units. This command will show all services installed on the host, whether or not they are active:

```
# systemctl --all -t service
```

The bottom of this listing of service units displays 166 as the total number of loaded units on my host. Your number will probably differ.

Unit files do not have a filename extension (such as **.unit**) to help identify them, so you can generalize that most configuration files that belong to systemd are unit files of one type or another. The few remaining files are mostly **.conf** files located in **/etc/systemd**.

Unit files are stored in the **/usr/lib/systemd** directory and its subdirectories, while the **/etc/systemd/** directory and its subdirectories contain symbolic links to the unit files necessary to the local configuration of this host.

To explore this, change directory to **/etc/systemd** and list its contents. Then change directory to **/etc/systemd/system** and list its contents.

Take a look at the **default.target** file, which determines which runlevel target the system will boot to. In the second article in this series, I explained how to change the default target from the GUI (**graphical.target**) to the command-line only (**multi-user.target**) target. The **default.target** file on my test VM is simply a symlink to **/usr/lib/systemd/system/graphical.target**.

Examine the contents of the **/etc/systemd/system/default.target** file:

```
# cat default.target
#  SPDX-License-Identifier: LGPL-2.1+
#
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.

[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

Note that this requires the **multi-user.target**; the **graphical.target** cannot start if the **multi-user.target** is not already up and running. It also says it "wants" the **display-manager.service** unit. A "want" does not need to be fulfilled in order for the unit to start successfully. If the "want" cannot be fulfilled, it will be ignored by systemd, and the rest of the target will start regardless.

The subdirectories in **/etc/systemd/system** are lists of wants for various targets. Explore the files and their contents in the **/etc/systemd/system/graphical.target.wants** directory.

The **systemd.unit** man page contains a lot of good information about unit files, their structure, the sections they can be divided into, and the options that can be used. It also lists many of the unit types, all of which have their own man pages. If you want to interpret a unit file, this would be a good place to start.

Service units

A Fedora installation usually installs and enables services that particular hosts do not need for normal operation. Conversely, sometimes it doesn't include services that need to be installed, enabled, and started. Services that are not needed for the Linux host to function as desired, but which are installed and possibly running, represent a security risk and should—at minimum—be stopped and disabled and—at best—should be uninstalled.

The **systemctl** command is used to manage systemd units, including services, targets, mounts, and more. Take a closer look at the list of services to identify services that will never be used:

```
# systemctl --all -t service
UNIT                           LOAD      ACTIVE SUB        DESCRIPTION
<snip>
chronyd.service                loaded    active running    NTP client/server
crond.service                  loaded    active running    Command Scheduler
cups.service                   loaded    active running    CUPS Scheduler
dbus-daemon.service            loaded    active running    D-Bus System Message
Bus
[...]
● ip6tables.service           not-found inactive dead     ip6tables.service
● ipset.service               not-found inactive dead     ipset.service
● iptables.service            not-found inactive dead     iptables.service
[...]
firewalld.service              loaded    active   running  firewalld - dynamic
firewall daemon
[...]
● ntpd.service                not-found inactive dead     ntpd.service
● ntpdate.service             not-found inactive dead     ntpdate.service
pcscd.service                  loaded    active   running  PC/SC Smart Card
Daemon
```

I have pruned out most of the output from the command to save space. The services that show "loaded active running" are obvious. The "not-found" services are ones that systemd is aware of but are not installed on the Linux host. If you want to run those services, you must install the packages that contain them.

Note the **pcscd.service** unit. This is the PC/SC smart-card daemon. Its function is to communicate with smart-card readers. Many Linux hosts—including VMs—have no need for this reader nor the service that is loaded and taking up memory and CPU resources. You can stop this service and disable it, so it will not restart on the next boot. First, check its status:

```
# systemctl status pcscd.service
● pcscd.service - PC/SC Smart Card Daemon
   Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor
preset: disabled)
   Active: active (running) since Fri 2019-05-10 11:28:42 EDT; 3 days ago
     Docs: man:pcscd(8)
 Main PID: 24706 (pcscd)
    Tasks: 6 (limit: 4694)
   Memory: 1.6M
   CGroup: /system.slice/pcscd.service
           └─24706 /usr/sbin/pcscd --foreground --auto-exit

May 10 11:28:42 testvm1 systemd[1]: Started PC/SC Smart Card Daemon.
```

This data illustrates the additional information systemd provides versus SystemV, which only reports whether or not the service is running. Note that specifying the **.service** unit type is optional. Now stop and disable the service, then re-check its status:

```
# systemctl stop pcscd ; systemctl disable pcscd
Warning: Stopping pcscd.service, but it can still be activated by:
  pcscd.socket
Removed /etc/systemd/system/sockets.target.wants/pcscd.socket.
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
   Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect;...)
   Active: failed (Result: exit-code) since Mon 2019-05-13 15:23:15 EDT; 48s ago
 Main PID: 24706 (code=exited, status=1/FAILURE)

May 10 11:28:42 testvm1 systemd[1]: Started PC/SC Smart Card Daemon.
May 13 15:23:15 testvm1 systemd[1]: Stopping PC/SC Smart Card Daemon...
May 13 15:23:15 testvm1 systemd[1]: pcscd.service: Main process exited,
code=exited, status=1/FAIL>
May 13 15:23:15 testvm1 systemd[1]: pcscd.service: Failed with 'exit-code'.
May 13 15:23:15 testvm1 systemd[1]: Stopped PC/SC Smart Card Daemon.
```

The short log entry display for most services prevents having to search through various log files to locate this type of information. Check the status of the system runlevel targets—specifying the "target" unit type is required:

```
# systemctl status multi-user.target
● multi-user.target - Multi-User System
   Loaded: loaded (/usr/lib/systemd/system/multi-user.target; static; vendor
preset: disabled)
   Active: active since Thu 2019-05-09 13:27:22 EDT; 4 days ago
     Docs: man:systemd.special(7)

May 09 13:27:22 testvm1 systemd[1]: Reached target Multi-User System.
# systemctl status graphical.target
● graphical.target - Graphical Interface
   Loaded: loaded (/usr/lib/systemd/system/graphical.target; indirect; vendor
preset: disabled)
   Active: active since Thu 2019-05-09 13:27:22 EDT; 4 days ago
     Docs: man:systemd.special(7)

May 09 13:27:22 testvm1 systemd[1]: Reached target Graphical Interface.
# systemctl status default.target
● graphical.target - Graphical Interface
   Loaded: loaded (/usr/lib/systemd/system/graphical.target; indirect; vendor
preset: disabled)
   Active: active since Thu 2019-05-09 13:27:22 EDT; 4 days ago
     Docs: man:systemd.special(7)

May 09 13:27:22 testvm1 systemd[1]: Reached target Graphical Interface.
```

The default target is the graphical target. The status of any unit can be checked in this way.

Mounts the old way

A mount unit defines all of the parameters required to mount a filesystem on a designated mount point. systemd can manage mount units with more flexibility than those using the **/etc/fstab** filesystem configuration file. Despite this, systemd still uses the **/etc/fstab** file for filesystem configuration and mounting purposes. systemd uses the **systemd-fstab-generator** tool to create transient mount units from the data in the **fstab** file.

I will create a new filesystem and a systemd mount unit to mount it. If you have some available disk space on your test system, you can do it along with me.

Note that the volume group and logical volume names may be different on your test system. Be sure to use the names that are pertinent to your system.

You need to create a partition or logical volume, then make an EXT4 filesystem on it. Add a label to the filesystem, **TestFS**, and create a directory for a mount point **/TestFS**.

To try this on your own, first, verify that you have free space on the volume group. Here is what that looks like on my VM where I have some space available on the volume group to create a new logical volume:

```
# lsblk
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda             8:0    0  120G  0 disk
├─sda1          8:1    0    4G  0 part /boot
└─sda2          8:2    0  116G  0 part
  ├─VG01-root 253:0    0    5G  0 lvm  /
  ├─VG01-swap 253:1    0    8G  0 lvm  [SWAP]
  ├─VG01-usr  253:2    0   30G  0 lvm  /usr
  ├─VG01-home 253:3    0   20G  0 lvm  /home
  ├─VG01-var  253:4    0   20G  0 lvm  /var
  └─VG01-tmp  253:5    0   10G  0 lvm  /tmp
sr0            11:0    1 1024M  0 rom
# vgs
  VG   #PV #LV #SN Attr   VSize    VFree
  VG01   1   6   0 wz--n- <116.00g <23.00g
```

Then create a new volume on **VG01** named **TestFS**. It does not need to be large; 1GB is fine. Then create a filesystem, add the filesystem label, and create the mount point:

```
# lvcreate -L 1G -n TestFS VG01
  Logical volume "TestFS" created.
# mkfs -t ext4 /dev/mapper/VG01-TestFS
mke2fs 1.45.3 (14-Jul-2019)
Creating filesystem with 262144 4k blocks and 65536 inodes
Filesystem UUID: 8718fba9-419f-4915-ab2d-8edf811b5d23
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

# e2label /dev/mapper/VG01-TestFS TestFS
# mkdir /TestFS
```

Mount the new filesystem (it fails, but try it anyway):

```
# mount /TestFS/
mount: /TestFS/: can't find in /etc/fstab.
```

Mounting the volume doesn't work, because you don't have an entry in **/etc/fstab**. You can mount the new filesystem even without the entry in **/etc/fstab** using both the device name (as it appears in **/dev**) and the mount point. Mounting in this manner is simpler than it used to be—it used to require the filesystem type as an argument. The mount command is now smart enough to detect the filesystem type and mount it accordingly.

Try it again:

```
# mount /dev/mapper/VG01-TestFS /TestFS/
# lsblk
NAME            MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda               8:0    0  120G  0 disk
├─sda1            8:1    0    4G  0 part /boot
└─sda2            8:2    0  116G  0 part
  ├─VG01-root   253:0    0    5G  0 lvm  /
  ├─VG01-swap   253:1    0    8G  0 lvm  [SWAP]
  ├─VG01-usr    253:2    0   30G  0 lvm  /usr
  ├─VG01-home   253:3    0   20G  0 lvm  /home
  ├─VG01-var    253:4    0   20G  0 lvm  /var
  ├─VG01-tmp    253:5    0   10G  0 lvm  /tmp
  └─VG01-TestFS 253:6    0    1G  0 lvm  /TestFS
sr0              11:0    1 1024M  0 rom
```

Now the new filesystem is mounted in the proper location. List the mount unit files:

```
# systemctl list-unit-files -t mount
```

This command does not show a file for the **/TestFS** filesystem because no file exists for it. The command **systemctl status TestFS.mount** does not display any information about the new filesystem either. You can try it using wildcards with the **systemctl status** command:

```
# systemctl status *mount
● usr.mount - /usr
   Loaded: loaded (/etc/fstab; generated)
   Active: active (mounted)
    Where: /usr
     What: /dev/mapper/VG01-usr
     Docs: man:fstab(5)
           man:systemd-fstab-generator(8)

[...]
● TestFS.mount - /TestFS
   Loaded: loaded (/proc/self/mountinfo)
   Active: active (mounted) since Fri 2020-04-17 16:02:26 EDT; 1min 18s ago
    Where: /TestFS
     What: /dev/mapper/VG01-TestFS
```

```
● run-user-0.mount - /run/user/0
   Loaded: loaded (/proc/self/mountinfo)
   Active: active (mounted) since Thu 2020-04-16 08:52:29 EDT; 1 day 5h ago
    Where: /run/user/0
     What: tmpfs

● var.mount - /var
   Loaded: loaded (/etc/fstab; generated)
   Active: active (mounted) since Thu 2020-04-16 12:51:34 EDT; 1 day 1h ago
    Where: /var
     What: /dev/mapper/VG01-var
     Docs: man:fstab(5)
           man:systemd-fstab-generator(8)
    Tasks: 0 (limit: 19166)
   Memory: 212.0K
      CPU: 5ms
   CGroup: /system.slice/var.mount
```

This command provides some very interesting information about your system's mounts, and your new filesystem shows up. The **/var** and **/usr** filesystems are identified as being generated from **/etc/fstab**, while your new filesystem simply shows that it is loaded and provides the location of the info file in the **/proc/self/mountinfo** file.

Next, automate this mount. First, do it the old-fashioned way by adding an entry in **/etc/fstab**. Later, I'll show you how to do it the new way, which will teach you about creating units and integrating them into the startup sequence.

Unmount **/TestFS** and add the following line to the **/etc/fstab** file:

```
/dev/mapper/VG01-TestFS  /TestFS       ext4    defaults        1 2
```

Now, mount the filesystem with the simpler **mount** command and list the mount units again:

```
# mount /TestFS
# systemctl status *mount
[...]
● TestFS.mount - /TestFS
   Loaded: loaded (/proc/self/mountinfo)
   Active: active (mounted) since Fri 2020-04-17 16:26:44 EDT; 1min 14s ago
    Where: /TestFS
     What: /dev/mapper/VG01-TestFS
[...]
```

This did not change the information for this mount because the filesystem was manually mounted. Reboot and run the command again, and this time specify **TestFS.mount** rather than using the wildcard. The results for this mount are now consistent with it being mounted at startup:

```
# systemctl status TestFS.mount
● TestFS.mount - /TestFS
   Loaded: loaded (/etc/fstab; generated)
   Active: active (mounted) since Fri 2020-04-17 16:30:21 EDT; 1min 38s ago
    Where: /TestFS
     What: /dev/mapper/VG01-TestFS
     Docs: man:fstab(5)
           man:systemd-fstab-generator(8)
    Tasks: 0 (limit: 19166)
   Memory: 72.0K
      CPU: 6ms
   CGroup: /system.slice/TestFS.mount

Apr 17 16:30:21 testvm1 systemd[1]: Mounting /TestFS...
Apr 17 16:30:21 testvm1 systemd[1]: Mounted /TestFS.
```

Creating a mount unit

Mount units may be configured either with the traditional **/etc/fstab** file or with systemd units. Fedora uses the **fstab** file as it is created during the installation. However, systemd uses the **systemd-fstab-generator** program to translate the **fstab** file into systemd units for each entry in the **fstab** file. Now that you know you can use systemd **.mount** unit files for filesystem mounting, try it out by creating a mount unit for this filesystem.

First, unmount **/TestFS**. Edit the **/etc/fstab** file and delete or comment out the **TestFS** line. Now, create a new file with the name **TestFS.mount** in the **/etc/systemd/system** directory. Edit it to contain the configuration data below. The unit file name and the name of the mount point *must* be identical, or the mount will fail:

```
# This mount unit is for the TestFS filesystem
# By David Both
# Licensed under GPL V2
# This file should be located in the /etc/systemd/system directory

[Unit]
Description=TestFS Mount

[Mount]
```

```
What=/dev/mapper/VG01-TestFS
Where=/TestFS
Type=ext4
Options=defaults

[Install]
WantedBy=multi-user.target
```

The **Description** line in the **[Unit]** section is for us humans, and it provides the name that's shown when you list mount units with **systemctl -t mount**. The data in the **[Mount]** section of this file contains essentially the same data that would be found in the **fstab** file.

Now enable the mount unit:

```
# systemctl enable TestFS.mount
Created symlink /etc/systemd/system/multi-user.target.wants/TestFS.mount →
/etc/systemd/system/TestFS.mount.
```

This creates the symlink in the **/etc/systemd/system** directory, which will cause this mount unit to be mounted on all subsequent boots. The filesystem has not yet been mounted, so you must "start" it:

```
# systemctl start TestFS.mount
```

Verify that the filesystem has been mounted:

```
# systemctl status TestFS.mount
● TestFS.mount - TestFS Mount
   Loaded: loaded (/etc/systemd/system/TestFS.mount; enabled; vendor preset:
disabled)
   Active: active (mounted) since Sat 2020-04-18 09:59:53 EDT; 14s ago
    Where: /TestFS
     What: /dev/mapper/VG01-TestFS
    Tasks: 0 (limit: 19166)
   Memory: 76.0K
      CPU: 3ms
   CGroup: /system.slice/TestFS.mount

Apr 18 09:59:53 testvm1 systemd[1]: Mounting TestFS Mount...
Apr 18 09:59:53 testvm1 systemd[1]: Mounted TestFS Mount.
```

This experiment has been specifically about creating a unit file for a mount, but it can be applied to other types of unit files as well. The details will be different, but the concepts are the same. Yes, I know it is still easier to add a line to the **/etc/fstab** file than it is to create a mount unit. But this is a good example of how to create a unit file because systemd does not have generators for every type of unit.

In summary

This article looked at systemd units in more detail and how to use the systemctl command to explore and manage units. It also showed how to stop and disable units and create a new systemd mount unit to mount a new filesystem and enable it to initiate during startup.

Start using systemd as a troubleshooting tool

No one would really consider systemd to be a troubleshooting tool, but when I encountered a problem on my webserver, my growing knowledge of systemd and some of its features helped me locate and circumvent the problem.

The problem was that my server, yorktown, which provides name services, DHCP, NTP, HTTPD, and SendMail email services for my home office network, failed to start the Apache HTTPD daemon during normal startup. I had to start it manually after I realized that it was not running. The problem had been going on for some time, and I recently got around to trying to fix it.

Some of you will say that systemd itself is the cause of this problem, and, based on what I know now, I agree with you. However, I had similar types of problems with SystemV. (In the first article in this series, I looked at the controversy around systemd as a replacement for the old SystemV init program and startup scripts.

No software is perfect, and neither systemd nor SystemV is an exception, but systemd provides far more information for problem-solving than SystemV ever offered.

Determining the problem

The first step to finding the source of this problem is to determine the httpd service's status:

```
# systemctl status httpd
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset:
disabled)
   Active: failed (Result: exit-code) since Thu 2020-04-16 11:54:37; 15min ago
     Docs: man:httpd.service(8)
  Process: 1101 ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND (code=exited,
status=1/FAILURE)
```

```
 Main PID: 1101 (code=exited, status=1/FAILURE)
   Status: "Reading configuration..."
      CPU: 60ms

Apr 16 11:54:35 yorktown.both.org systemd[1]: Starting The Apache HTTP Server...
Apr 16 11:54:37 yorktown.both.org httpd[1101]: (99)Cannot assign requested
address: AH00072: make_sock: could not bind to address 192.168.0.52:80
Apr 16 11:54:37 yorktown.both.org httpd[1101]: no listening sockets available,
shutting down
Apr 16 11:54:37 yorktown.both.org httpd[1101]: AH00015: Unable to open logs
Apr 16 11:54:37 yorktown.both.org systemd[1]: httpd.service: Main process exited,
code=exited, status=1/FAILURE
Apr 16 11:54:37 yorktown.both.org systemd[1]: httpd.service: Failed with result
'exit-code'.
Apr 16 11:54:37 yorktown.both.org systemd[1]: Failed to start The Apache HTTP
Server.
```

This status information is one of the systemd features that I find much more useful than anything SystemV offers. The amount of helpful information here leads me easily to a logical conclusion that takes me in the right direction. All I ever got from the old **chkconfig** command is whether or not the service is running and the process ID (PID) if it is. That isn't very helpful.

The key entry in this status report shows that HTTPD cannot bind to the IP address, which means it cannot accept incoming requests. This indicates that the network is not starting fast enough to be ready for the HTTPD service to bind to the IP address because the IP address has not yet been set. This is not supposed to happen, so I explored my network service systemd startup configuration files; all appeared to be correct with the right "after" and "requires" statements. Here is the **/lib/systemd/system/httpd.service** file from my server:

```
# Modifying this file in-place is not recommended, because changes
# will be overwritten during package upgrades.  To customize the
# behaviour, run "systemctl edit httpd" to create an override unit.

# For example, to pass additional options (such as -D definitions) to
# the httpd binary at startup, create an override unit (as is done by
# systemctl edit) and enter the following:

#       [Service]
#       Environment=OPTIONS=-DMY_DEFINE

[Unit]
Description=The Apache HTTP Server
Wants=httpd-init.service
After=network.target remote-fs.target nss-lookup.target httpd-init.service
Documentation=man:httpd.service(8)
```

```
[Service]
Type=notify
Environment=LANG=C

ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
# Send SIGWINCH for graceful stop
KillSignal=SIGWINCH
KillMode=mixed
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

The **httpd.service** unit file explicitly specifies that it should load after the **network.target** and the **httpd-init.service** (among others). I tried to find all of these services using the **systemctl list-units** command and searching for them in the resulting data stream. All were present and should have ensured that the httpd service did not load before the network IP address was set.

First solution

A bit of searching on the internet confirmed that others had encountered similar problems with httpd and other services. This appears to happen because one of the required services indicates to systemd that it has finished its startup—but it actually spins off a child process that has not finished. After a bit more searching, I came up with a circumvention.

I could not figure out why the IP address was taking so long to be assigned to the network interface card. So, I thought that if I could delay the start of the HTTPD service by a reasonable amount of time, the IP address would be assigned by that time.

Fortunately, the **/lib/systemd/system/httpd.service** file above provides some direction. Although it says not to alter it, it does indicate how to proceed: Use the command **systemctl edit httpd**, which automatically creates a new file (**/etc/systemd/system/httpd.service.d/override.conf**) and opens the GNU Nano editor. (If you are not familiar with Nano, be sure to look at the hints at the bottom of the Nano interface.)

Add the following text to the new file and save it:

```
# cd /etc/systemd/system/httpd.service.d/
```

```
# ll
total 4
-rw-r--r-- 1 root root 243 Apr 16 11:43 override.conf
[root@yorktown httpd.service.d]# cat override.conf
# Trying to delay the startup of httpd so that the network is
# fully up and running so that httpd can bind to the correct
# IP address
#
# By David Both, 2020-04-16

[Service]
ExecStartPre=/bin/sleep 30
```

The **[Service]** section of this override file contains a single line that delays the start of the HTTPD service by 30 seconds. The following status command shows the service status during the wait time:

```
# systemctl status httpd
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset:
disabled)
  Drop-In: /etc/systemd/system/httpd.service.d
           └─override.conf
           /usr/lib/systemd/system/httpd.service.d
           └─php-fpm.conf
   Active: activating (start-pre) since Thu 2020-04-16 12:14:29 EDT; 28s ago
     Docs: man:httpd.service(8)
Cntrl PID: 1102 (sleep)
    Tasks: 1 (limit: 38363)
   Memory: 260.0K
      CPU: 2ms
   CGroup: /system.slice/httpd.service
           └─1102 /bin/sleep 30

Apr 16 12:14:29 yorktown.both.org systemd[1]: Starting The Apache HTTP Server...
Apr 16 12:15:01 yorktown.both.org systemd[1]: Started The Apache HTTP Server.
```

And this command shows the status of the HTTPD service after the 30-second delay expires. The service is up and running correctly:

```
# systemctl status httpd
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset:
disabled)
  Drop-In: /etc/systemd/system/httpd.service.d
           └─override.conf
           /usr/lib/systemd/system/httpd.service.d
```

```
           └─php-fpm.conf
   Active: active (running) since Thu 2020-04-16 12:15:01 EDT; 1min 18s ago
     Docs: man:httpd.service(8)
  Process: 1102 ExecStartPre=/bin/sleep 30 (code=exited, status=0/SUCCESS)
 Main PID: 1567 (httpd)
   Status: "Total requests: 0; Idle/Busy workers 100/0;Requests/sec: 0; Bytes
served/sec:   0 B/sec"
    Tasks: 213 (limit: 38363)
   Memory: 21.8M
      CPU: 82ms
   CGroup: /system.slice/httpd.service
           ├─1567 /usr/sbin/httpd -DFOREGROUND
           ├─1569 /usr/sbin/httpd -DFOREGROUND
           ├─1570 /usr/sbin/httpd -DFOREGROUND
           ├─1571 /usr/sbin/httpd -DFOREGROUND
           └─1572 /usr/sbin/httpd -DFOREGROUND

Apr 16 12:14:29 yorktown.both.org systemd[1]: Starting The Apache HTTP Server...
Apr 16 12:15:01 yorktown.both.org systemd[1]: Started The Apache HTTP Server.
```

I could have experimented to see if a shorter delay would work as well, but my system is not that critical, so I decided not to. It works reliably as it is, so I am happy.

Because I gathered all this information, I reported it to Red Hat Bugzilla as Bug 1825554. I believe that it is much more productive to report bugs than it is to complain about them.

The better solution

A couple of days after reporting this as a bug, I received a response indicating that systemd is just the manager, and if httpd needs to be ordered after some requirements are met, it needs to be expressed in the unit file. The response pointed me to the **httpd.service** man page. I wish I had found this earlier because it is a better solution than the one I came up with. This solution is explicitly targeted to the prerequisite target unit rather than a somewhat random delay.

From the **httpd.service** man page:

> **Starting the service at boot time**
>
> The httpd.service and httpd.socket units are *disabled* by default. To start the httpd service at boot time, run: **systemctl enable httpd.service**. In the default configuration, the httpd daemon will accept connections on port 80 (and, if

> mod_ssl is installed, TLS connections on port 443) for any configured IPv4 or IPv6 address.
>
> If httpd is configured to depend on any specific IP address (for example, with a "Listen" directive) which may only become available during start-up, or if httpd depends on other services (such as a database daemon), the service *must* be configured to ensure correct start-up ordering.
>
> For example, to ensure httpd is only running after all configured network interfaces are configured, create a drop-in file (as described above) with the following section:
>
> ```
> [Unit]
> After=network-online.target
> Wants=network-online.target
> ```

I still think this is a bug because it is quite common—at least in my experience—to use a **Listen** directive in the **httpd.conf** configuration file. I have always used **Listen** directives, even on hosts with only a single IP address, and it is clearly necessary on hosts with multiple network interface cards (NICs) and internet protocol (IP) addresses. Adding the lines above to the **/usr/lib/systemd/system/httpd.service** default file would not cause problems for configurations that do not use a **Listen** directive, and prevents this problem for those that do.

In the meantime, I will use the suggested solution.

Next steps

This article describes a problem I had with starting the Apache HTTPD service on my server. It leads you through the problem determination steps I took and shows how I used systemd to assist. I also covered the circumvention I implemented using systemd and the better solution that followed from my bug report.

It's likely that this is the result of a problem with systemd, specifically the configuration for httpd startup. Nevertheless, systemd provided me with the tools to locate the likely source of the problem and to formulate and implement a circumvention. Neither solution really resolves the problem to my satisfaction.

One of the things I discovered during this is process is that I need to learn more about defining the sequences in which things start!

Manage startup using systemd

While setting up a Linux system recently, I wanted to know how to ensure that dependencies for services and other units were up and running before those dependent services and units start. Specifically, I needed more knowledge of how systemd manages the startup sequence, especially in determining the order services are started in what is essentially a parallel system.

You may know that SystemV (systemd's predecessor) orders the startup sequence by naming the startup scripts with an SXX prefix, where XX is a number from 00 to 99. SystemV then uses the sort order by name and runs each start script in sequence for the desired runlevel.

But systemd uses unit files, which can be created or modified by a sysadmin, to define subroutines for not only initialization but also for regular operation. In the [third article](#) in this series, I explained how to create a mount unit file. In this fifth article, I demonstrate how to create a different type of unit file—a service unit file that runs a program at startup. You can also change certain configuration settings in the unit file and use the systemd journal to view the location of your changes in the startup sequence.

Preparation

Make sure you have removed `rhgb` and `quiet` from the `GRUB_CMDLINE_LINUX=` line in the `/etc/default/grub` file. This enables you to observe the Linux startup message stream, which you'll need for some of the experiments in this article.

The program

In this tutorial, you will create a simple program that enables you to observe a message during startup on the console and later in the systemd journal.

Create the shell program `/usr/local/bin/hello.sh` and add the following content. You want to ensure that the result is visible during startup and that you can easily find it when looking through the systemd journal. You will use a version of the "Hello world" program with some bars around it, so it stands out. Make sure the file is executable and has user and group ownership by root with [700 permissions] for security:

```
#!/usr/bin/bash
# Simple program to use for testing startup configurations
# with systemd.
# By David Both
# Licensed under GPL V2
#
echo "######### Hello World! ########"
```

Run this program from the command line to verify that it works correctly:

```
# ./hello.sh
######### Hello World! ########
```

This program could be created in any scripting or compiled language. The `hello.sh` program could also be located in other places based on the Linux filesystem hierarchical structure (FHS). I place it in the `/usr/local/bin` directory so that it can be easily run from the command line without having to prepend a path when I type the command. I find that many of the shell programs I create need to be run from the command line and by other tools such as systemd.

The service unit file

Create the service unit file `/etc/systemd/system/hello.service` with the following content. This file does not need to be executable, but for security, it does need user and group ownership by root and 644 or 640 permissions:

```
# Simple service unit file to use for testing
# startup configurations with systemd.
# By David Both
# Licensed under GPL V2
#

[Unit]
Description=My hello shell script

[Service]
Type=oneshot
ExecStart=/usr/local/bin/hello.sh

[Install]
WantedBy=multi-user.target
```

Verify that the service unit file performs as expected by viewing the service status. Any syntactical errors will show up here:

```
# systemctl status hello.service
● hello.service - My hello shell script
     Loaded: loaded (/etc/systemd/system/hello.service; disabled; vendor preset:
disabled)
     Active: inactive (dead)
```

You can run this "oneshot" service type multiple times without problems. The oneshot type is intended for services where the program launched by the service unit file is the main process and must complete before systemd starts any dependent process.

There are seven service types, and you can find an explanation of each (along with the other parts of a service unit file) in the systemd.service(5) man page. (You can also find more information in the resources at the end of this article.)

As curious as I am, I wanted to see what an error might look like. So, I deleted the "o" from the `Type=oneshot` line, so it looked like `Type=neshot`, and ran the command again:

```
# systemctl status hello.service
● hello.service - My hello shell script
     Loaded: loaded (/etc/systemd/system/hello.service; disabled; vendor preset:
disabled)
     Active: inactive (dead)

May 06 08:50:09 testvm1.both.org systemd[1]:
/etc/systemd/system/hello.service:12: Failed to parse service type, ignoring:
neshot
```

These results told me where the error was and made it very easy to resolve the problem.

Just be aware that even after you restore the `hello.service` file to its original form, the error will persist. Although a reboot will clear the error, you should not have to do that, so I went looking for a method to clear out persistent errors like this. I have encountered service errors that require the command `systemctl daemon-reload` to reset an error condition, but that did not work in this case. The error messages that can be fixed with this command always seem to have a statement to that effect, so you know to run it.

It is, however, recommended that you run `systemctl daemon-reload` after changing a unit file or creating a new one. This notifies systemd that the changes have been made, and it can prevent certain types of issues with managing altered services or units. Go ahead and run this command.

After correcting the misspelling in the service unit file, a simple `systemctl restart hello.service` cleared the error. Experiment a bit by introducing some other errors into the `hello.service` file to see what kinds of results you get.

Start the service

Now you are ready to start the new service and check the status to see the result. Although you probably did a restart in the previous section, you can start or restart a oneshot service as many times as you want since it runs once and then exits.

Go ahead and start the service (as shown below), and then check the status. Depending upon how much you experimented with errors, your results may differ from mine:

```
# systemctl start hello.service
# systemctl status hello.service
● hello.service - My hello shell script
     Loaded: loaded (/etc/systemd/system/hello.service; disabled; vendor preset)
     Active: inactive (dead)

May 10 10:37:49 testvm1.both.org hello.sh[842]: ######### Hello World! ########
May 10 10:37:49 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:37:49 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:54:45 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ######### Hello World! ########
May 10 10:54:45 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:54:45 testvm1.both.org systemd[1]: Finished My hello shell script.
```

Notice in the status command's output that the systemd messages indicate that the `hello.sh` script started and the service completed. You can also see the output from the script. This display is generated from the journal entries of the most recent invocations of the service. Try starting the service several times, and then run the status command again to see what I mean.

You should also look at the journal contents directly; there are multiple ways to do this. One way is to specify the record type identifier, in this case, the name of the shell script. This shows the journal entries for previous reboots as well as the current session. As you can see, I have been researching and testing for this article for some time now:

```
# journalctl -t hello.sh
[...]
-- Reboot --
May 08 15:55:47 testvm1.both.org hello.sh[840]: ######### Hello World! ########
-- Reboot --
```

```
May 08 16:01:51 testvm1.both.org hello.sh[840]: ######### Hello World! ########
-- Reboot --
May 10 10:37:49 testvm1.both.org hello.sh[842]: ######### Hello World! ########
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ######### Hello World! ########
```

To locate the systemd records for the `hello.service` unit, you can search on systemd. You can use **G+Enter** to page to the end of the journal entries and then scroll back to locate the ones you're interested in. Use the `-b` option to show only the most recent startup:

```
# journalctl -b -t systemd
[...]
May 10 10:37:49 testvm1.both.org systemd[1]: Starting SYSV: Late init script for
live image....
May 10 10:37:49 testvm1.both.org systemd[1]: Started SYSV: Late init script for
live image..
May 10 10:37:49 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:37:49 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:37:50 testvm1.both.org systemd[1]: Starting D-Bus System Message Bus...
May 10 10:37:50 testvm1.both.org systemd[1]: Started D-Bus System Message Bus.
```

I copied a few other journal entries to give you an idea of what you might find. This command spews all of the journal lines pertaining to systemd—109,183 lines when I wrote this. That is a lot of data to sort through. You can use the pager's search facility, which is usually `less`, or you can use the built-in `grep` feature. The `-g` (or `--grep=`) option uses Perl-compatible regular expressions:

```
# journalctl -b -t systemd -g "hello"
# journalctl -b -t systemd -g "hello"
-- Logs begin at Tue 2020-05-05 18:11:49 EDT, end at Sun 2020-05-10 11:01:01 EDT.
May 10 10:37:49 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:37:49 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:37:49 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:54:45 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:54:45 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:54:45 testvm1.both.org systemd[1]: Finished My hello shell script.
```

You could use the standard GNU `grep` command, but that would not show the log metadata in the first line.

If you do not want to see just the journal entries pertaining to your `hello` service, you can narrow things down a bit by specifying a time range. For example, I will start with the beginning time of `10:54:00` on my test VM, which was the start of the minute the entries above are from. Note that the `--since=` option must be enclosed in quotes and that this option can also be expressed as `-S "<time specification>"`.

The date and time will be different on your host, so be sure to use the timestamps that match the times in your journals:

```
# journalctl --since="2020-05-10 10:54:00"
May 10 10:54:35 testvm1.both.org audit: BPF prog-id=54 op=LOAD
May 10 10:54:35 testvm1.both.org audit: BPF prog-id=55 op=LOAD
May 10 10:54:45 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ###############################
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ######### Hello World! ########
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ###############################
May 10 10:54:45 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:54:45 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:54:45 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd"'
May 10 10:54:45 testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/"'
May 10 10:56:00 testvm1.both.org NetworkManager[840]: <error> [1589122560.0633]
dhcp4 (enp0s3): error -113 dispatching events
May 10 10:56:00 testvm1.both.org NetworkManager[840]: <info>  [1589122560.0634]
dhcp4 (enp0s3): state changed bound -> fail
[...]
```

The `since` specification skips all of the entries before that time, but there are still a lot of entries after that time that you do not need. You can also use the `until` option to trim off the entries that come a bit after the time you are interested in. I want the entire minute when the event occurred and nothing more:

```
# journalctl --since="2020-05-10 10:54:35" --until="2020-05-10 10:55:00"
-- Logs begin at Tue 2020-05-05 18:11:49 EDT, end at Sun 2020-05-10 11:04:59 EDT.
May 10 10:54:35 testvm1.both.org systemd[1]: Reloading.
May 10 10:54:35 testvm1.both.org audit: BPF prog-id=27 op=UNLOAD
May 10 10:54:35 testvm1.both.org audit: BPF prog-id=26 op=UNLOAD
[...]
ay 10 10:54:35 testvm1.both.org audit: BPF prog-id=55 op=LOAD
May 10 10:54:45 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ###############################
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ######### Hello World! ########
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ###############################
May 10 10:54:45 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:54:45 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:54:45 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd>
```

```
May 10 10:54:45 testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/>
lines 1-46/46 (END)
```

If there were a lot of activity in this time period, you could further narrow the resulting data stream using a combination of these options:

```
# journalctl --since="2020-05-10 10:54:35" --until="2020-05-10 10:55:00" -t
"hello.sh"
-- Logs begin at Tue 2020-05-05 18:11:49 EDT, end at Sun 2020-05-10 11:10:41 EDT
May 10 10:54:45 testvm1.both.org hello.sh[1380]: ######### Hello World! ########
```

Your results should be similar to mine. You can see from this series of experiments that the service executed properly.

Reboot—finally

So far, you have not rebooted the host where you installed your service. So do that now because, after all, this how-to is about running a program at startup. First, you need to enable the service to launch during the startup sequence:

```
# systemctl enable hello.service
Created symlink /etc/systemd/system/multi-user.target.wants/hello.service →
/etc/systemd/system/hello.service.
```

Notice that the link was created in the `/etc/systemd/system/multi-user.target.wants` directory. This is because the service unit file specifies that the service is "wanted" by the `multi-user.target`.

Reboot, and be sure to watch the data stream during the startup sequence to see the "Hello world" message. Wait … you did not see it? Well, neither did I. Although it went by very fast, I did see systemd's message that it was starting the `hello.service`.

Look at the journal since the latest system boot. You can use the `less` pager's search tool to find "Hello" or "hello." I pruned many lines of data, but I left some of the surrounding journal entries, so you can get a feel for what the entries pertaining to your service look like locally:

```
# journalctl -b
[...]
May 10 10:37:49 testvm1.both.org systemd[1]: Listening on SSSD Kerberos Cache
Manager responder socket.
May 10 10:37:49 testvm1.both.org systemd[1]: Reached target Sockets.
May 10 10:37:49 testvm1.both.org systemd[1]: Reached target Basic System.
```

```
May 10 10:37:49 testvm1.both.org systemd[1]: Starting Modem Manager...
May 10 10:37:49 testvm1.both.org systemd[1]: Starting Network Manager...
May 10 10:37:49 testvm1.both.org systemd[1]: Starting Avahi mDNS/DNS-SD Stack...
May 10 10:37:49 testvm1.both.org systemd[1]: Condition check resulted in Secure
Boot DBX (blacklist) updater being skipped.
May 10 10:37:49 testvm1.both.org systemd[1]: Starting My hello shell script...
May 10 10:37:49 testvm1.both.org systemd[1]: Starting IPv4 firewall with
iptables...
May 10 10:37:49 testvm1.both.org systemd[1]: Started irqbalance daemon.
May 10 10:37:49 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=irqbalance comm="systemd"
exe="/usr/lib/sy>"'
May 10 10:37:49 testvm1.both.org systemd[1]: Starting LSB: Init script for live
image....
May 10 10:37:49 testvm1.both.org systemd[1]: Starting Hardware Monitoring
Sensors...
[...]
May 10 10:37:49 testvm1.both.org systemd[1]: Starting NTP client/server...
May 10 10:37:49 testvm1.both.org systemd[1]: Starting SYSV: Late init script for
live image....
May 10 10:37:49 testvm1.both.org systemd[1]: Started SYSV: Late init script for
live image..
May 10 10:37:49 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=livesys-late comm="systemd"
exe="/usr/lib/>"'
May 10 10:37:49 testvm1.both.org hello.sh[842]: ######### Hello World! ########
May 10 10:37:49 testvm1.both.org systemd[1]: hello.service: Succeeded.
May 10 10:37:49 testvm1.both.org systemd[1]: Finished My hello shell script.
May 10 10:37:49 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd>"'
May 10 10:37:49 testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/>
May 10 10:37:50 testvm1.both.org audit: BPF prog-id=28 op=LOAD
[...]
```

You can see that systemd started the `hello.service` unit, which ran the `hello.sh` shell script with the output recorded in the journal. If you were able to catch it during boot, you would also have seen the systemd message indicating that it was starting the script and another message indicating that the service succeeded. By looking at the first systemd message in the data stream above, you can see that systemd started your service very soon after reaching the basic system target.

But I would like to see the message displayed at startup as well. There is a way to make that happen: Add the following line to the `[Service]` section of the `hello.service` file:

Creative Commons Attribution Share-alike 4.0

```
StandardOutput=journal+console
```

The `hello.service` file now looks like this:

```
# Simple service unit file to use for testing
# startup configurations with systemd.
# By David Both
# Licensed under GPL V2
#

[Unit]
Description=My hello shell script

[Service]
Type=oneshot
ExecStart=/usr/local/bin/hello.sh
StandardOutput=journal+console

[Install]
WantedBy=multi-user.target
```

After adding this line, reboot the system, and watch the data stream as it scrolls up the display during the boot process. You should see the message in its little box. After the startup sequence completes, you can view the journal for the most recent boot and locate the entries for your new service.

Changing the sequence

Now that your service is working, you can look at where it starts in the startup sequence and experiment with changing it. It is important to remember that systemd's intent is to start as many services and other unit types in parallel within each of the major targets: `basic.target`, `multi-user.target`, and `graphical.target`. You should have just seen the journal entries for the most recent boot, which should look similar to my journal in the output above.

Notice that systemd started your test service soon after it reached the target basic system. This is what you specified in the service unit file in the `WantedBy` line, so it is correct. Before you change anything, list the contents of the `/etc/systemd/system/multi-user.target.wants` directory, and you will see a symbolic (soft) link to the service unit file. The `[Install]` section of the service unit file specifies which target will start the service, and running the `systemctl enable hello.service` command creates the link in the appropriate "target wants" directory:

```
hello.service -> /etc/systemd/system/hello.service
```

Certain services need to start during the `basic.target`, and others do not need to start unless the system is starting the `graphical.target`. The service in this experiment will not start in the `basic.target`—assume you do not need it to start until the `graphical.target`. So change the `WantedBy` line:

```
WantedBy=graphical.target
```

Be sure to disable the `hello.service` and re-enable it to delete the old link and add the new one in the `graphical.targets.wants` directory. I have noticed that if I forget to disable the service before changing the target that wants it, I can run the `systemctl disable` command, and the links will be removed from both "target wants" directories. Then, I just need to re-enable the service and reboot.

One concern with starting services in the `graphical.target` is that if the host boots to `multi-user.target`, this service will not start automatically. That may be what you want if the service requires a GUI desktop interface, but it also may not be what you want.

Look at the journal entries for the `graphical.target` and the `multi-user.target` using the `-o short-monotonic` option that displays seconds after kernel startup with microsecond precision:

```
# journalctl -b -o short-monotonic
```

Some results for `multi-user.target`:

```
[   17.264730] testvm1.both.org systemd[1]: Starting My hello shell script...
[   17.265561] testvm1.both.org systemd[1]: Starting IPv4 firewall...
[...]
[   19.478468] testvm1.both.org systemd[1]: Starting LSB: Init script for live
image....
[   19.507359] ...both.org iptables.init[844]: Applying firewall rules: [  OK  ]
[   19.507835] testvm1.both.org hello.sh[843]: ######### Hello World! ########
[...]
[   21.482481] testvm1.both.org systemd[1]: hello.service: Succeeded.
[   21.482550] testvm1.both.org smartd[856]: Opened configuration file
/etc/smartmontools/smartd.conf
[   21.482605] testvm1.both.org systemd[1]: Finished My hello shell script.
```

And some results for `graphical.target`:

```
[   19.436815] testvm1.both.org systemd[1]: Starting My hello shell script...
[   19.437070] testvm1.both.org systemd[1]: Starting IPv4 firewall...
[...]
[   19.612614] testvm1.both.org hello.sh[841]: ###############################
[   19.612614] testvm1.both.org hello.sh[841]: ######### Hello World! ########
[   19.612614] testvm1.both.org hello.sh[841]: ###############################
[   19.629455] testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/systemd" hostname=? addr=? terminal=? res=success'
[   19.629569] testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/systemd" hostname=? addr=? terminal=? res=success'
[   19.629682] testvm1.both.org systemd[1]: hello.service: Succeeded.
[   19.629782] testvm1.both.org systemd[1]: Finished My hello shell script.
```

Despite having the `graphical.target` "want" in the unit file, the `hello.service` unit runs about 19.5 or 19.6 seconds into startup. But `hello.service` starts at about 17.24 seconds in the `multi-user.target` and 19.43 seconds in the graphical target.

What does this mean? Look at the `/etc/systemd/system/default.target` link. The contents of that file show that systemd first starts the default target, `graphical.target`, which then pulls in the `multi-user.target`:

```
# cat default.target
#  SPDX-License-Identifier: LGPL-2.1+
#
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.

[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

Whether it starts the service with the `graphical.target` or the `multi-user.target`, the `hello.service` unit runs at about 19.5 or 19.6 seconds into startup. Based on this and the journal results (especially the ones using the monotonic output), you know that both of these targets are starting in parallel. Look at one more thing from the journal output:

```
[   28.397330] testvm1.both.org systemd[1]: Reached target Multi-User System.
[   28.397431] testvm1.both.org systemd[1]: Reached target Graphical Interface.
```

Both targets finish at almost the same time. This is consistent because `graphical.target` pulls in the `multi-user.target` and cannot finish until the `multi.user target` is reached (finished). But **hello.service** finishes much earlier than this.

What all this means is that these two targets start up pretty much in parallel. If you explore the journal entries, you will see various targets and services from each of those primary targets starting mostly in parallel. It is clear that the `multi-user.target` does not need to complete before the `graphical.target` starts. Therefore, simply using these primary targets to sequence the startup does not work very well, although it can be useful for ensuring that units are started only when they are needed for the `graphical.target`.

Before continuing, revert the `hello.service` unit file to `WantedBy=multi-user.target` (if it's not already.)

Ensure a service starts after the network is running

A common startup sequence issue is ensuring that a unit starts after the network is up and running. The Freedesktop.org article *Running services after the network is up* says there is no real consensus on when a network is considered "up." However, the article provides three options, and the one that meets the needs of a fully operational network is `network-online.target`. Just be aware that `network.target` is used during shutdown rather than startup, so it will not do you any good when you are trying to sequence the startup.

Before making any other changes, be sure to examine the journal and verify that the `hello.service` unit starts well before the network. You can look for the `network-online.target` in the journal to check.

Your service does not really require the network service, but you can use it as an avatar for one that does.

Because setting `WantedBy=graphical.target` does not ensure that the service will be started after the network is up and running, you need another way to ensure that it is. Fortunately, there is an easy way to do this. Add the following two lines to the `[Unit]` section of the `hello.service` unit file:

```
After=network-online.target
Wants=network-online.target
```

Both of these entries are required to make this work. Reboot the host and look for the location of entries for your service in the journals:

```
[   26.083121] testvm1.both.org NetworkManager[842]: <info>  [1589227764.0293]
device (enp0s3): Activation: successful, device activated.
[   26.083349] testvm1.both.org NetworkManager[842]: <info>  [1589227764.0301]
manager: NetworkManager state is now CONNECTED_GLOBAL
[   26.085818] testvm1.both.org NetworkManager[842]: <info>  [1589227764.0331]
manager: startup complete
[   26.089911] testvm1.both.org systemd[1]: Finished Network Manager Wait Online.
[   26.090254] testvm1.both.org systemd[1]: Reached target Network is Online.
[   26.090399] testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=NetworkManager-wait-online
comm="systemd" exe="/usr/lib/systemd/systemd" hostname=? addr=? termina>"'
[   26.091991] testvm1.both.org systemd[1]: Starting My hello shell script...
[   26.095864] testvm1.both.org sssd[be[implicit_files]][1007]: Starting up
[   26.290539] testvm1.both.org systemd[1]: Condition check resulted in Login and
scanning of iSCSI devices being skipped.
[   26.291075] testvm1.both.org systemd[1]: Reached target Remote File Systems
(Pre).
[   26.291154] testvm1.both.org systemd[1]: Reached target Remote File Systems.
[   26.292671] testvm1.both.org systemd[1]: Starting Notify NFS peers of a
restart...
[   26.294897] testvm1.both.org systemd[1]: iscsi.service: Unit cannot be
reloaded because it is inactive.
[   26.304682] testvm1.both.org hello.sh[1010]: ######### Hello World! ########
[   26.306569] testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/systemd" hostname=? addr=? terminal=? res=success'
[   26.306669] testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=hello comm="systemd"
exe="/usr/lib/systemd/systemd" hostname=? addr=? terminal=? res=success'
[   26.306772] testvm1.both.org systemd[1]: hello.service: Succeeded.
[   26.306862] testvm1.both.org systemd[1]: Finished My hello shell script.
[   26.584966] testvm1.both.org sm-notify[1011]: Version 2.4.3 starting
```

This confirms that the `hello.service` unit started after the `network-online.target`. This is exactly what you want. You may also have seen the "Hello World" message as it passed by during startup. Notice also that the timestamp is about six seconds later in the startup than it was before.

The best way to define the start sequence

This article explored Linux startup with systemd and unit files and journals in greater detail and discovered what happens when errors are introduced into the service file. As a sysadmin, I find that this type of experimentation helps me understand the behaviors of a program or service when it breaks, and breaking things intentionally is a good way to learn in a safe environment.

As the experiments in this article proved, just adding a service unit to either the `multi-user.target` or the `graphical.target` does not define its place in the start sequence. It merely determines whether a unit starts as part of a graphical environment or not. The reality is that the startup targets `multi-user.target` and `graphical.target`—and all of their Wants and Requires—start up pretty much in parallel. The best way to ensure that a unit starts in a specific order is to determine the unit it is dependent on and configure the new unit to "Want" and "After" the unit upon which it is dependent.

Control your computer time and date with systemd

Most people are concerned with time. We get up in time to perform our morning rituals and commute to work (a short trip for many of us these days), take a break for lunch, meet a project deadline, celebrate birthdays and holidays, catch a plane, and so much more.

Some of us are even *obsessed* with time. My watch is solar-powered and obtains the exact time from the National Institute of Standards and Technology (NIST) in Fort Collins, Colorado, via the WWVB time signal radio station located there. The time signals are synced to the atomic clock, also located in Fort Collins. My Fitbit syncs up to my phone, which is synced to a Network Time Protocol (NTP) server, which is ultimately synced to the atomic clock.

Why time is important to computers

There are many reasons our devices and computers need the exact time. For example, in banking, stock markets, and other financial businesses, transactions must be maintained in the proper order, and exact time sequences are critical for that.

Our phones, tablets, cars, GPS systems, and computers all require precise time and date settings. I want the clock on my computer desktop to be correct, so I can count on my local calendar application to pop up reminders at the correct time. The correct time also ensures SystemV cron jobs and systemd timers trigger at the correct time.

The correct time is also important for logging, so it is a bit easier to locate specific log entries based on the time. For one example, I once worked in DevOps (it was not called that at the time) for the State of North Carolina email system. We used to process more than 20 million emails per day. Following the trail of email through a series of servers or determining the exact sequence of events by using log files on geographically dispersed hosts can be much easier when the computers in question keep exact times.

Multiple times

Linux hosts have two times to consider: system time and RTC time. RTC stands for real-time clock, which is a fancy and not particularly accurate name for the system hardware clock.

The hardware clock runs continuously, even when the computer is turned off, by using a battery on the system motherboard. The RTC's primary function is to keep the time when a connection to a time server is not available. In the dark ages of personal computers, there was no internet to connect to a time server, so the only time a computer had available was the internal clock. Operating systems had to rely on the RTC at boot time, and the user had to manually set the system time using the hardware BIOS configuration interface to ensure it was correct.

The hardware clock does not understand the concept of time zones; only the time is stored in the RTC, not the time zone nor an offset from UTC (Universal Coordinated Time, which is also known as GMT, or Greenwich Mean Time).

The system time is the time known by the operating system. It is the time you see on the GUI clock on your desktop, in the output from the `date` command, in timestamps for logs, and in file access, modify, and change times.

The `rtc` man page contains a more complete discussion of the RTC and system clocks and RTC's functionality.

What about NTP?

Computers worldwide use the NTP (Network Time Protocol) to synchronize their time with internet standard reference clocks through a hierarchy of NTP servers. The primary time servers are at stratum 1, and they are connected directly to various national time services at stratum 0 via satellite, radio, or even modems over phone lines. The time services at stratum 0 may be an atomic clock, a radio receiver that is tuned to the signals broadcast by an atomic clock, or a GPS receiver using the highly accurate clock signals broadcast by GPS satellites.

To prevent time requests from time servers or clients lower in the hierarchy (i.e., with a higher stratum number) from overwhelming the primary reference servers, several thousand public NTP stratum 2 servers are open and available for all to use. Many organizations and users (including me) with large numbers of hosts that need an NTP server choose to set up their own time servers, so only one local host accesses the stratum 2 or 3 time servers. Then they configure the remaining hosts in the network to use the local time server. In the case of my home network, that is a stratum 3 server.

NTP implementation options

The original NTP implementation is **ntpd**, and it has been joined by two newer ones, **chronyd** and **systemd-timesyncd**. All three keep the local host's time synchronized with an NTP time server. The systemd-timesyncd service is not as robust as chronyd, but it is sufficient for most purposes. It can perform large time jumps if the RTC is far out of sync, and it can adjust the system time gradually to stay in sync with the NTP server if the local system time drifts a bit. The systemd-timesync service cannot be used as a time server.

Chrony is an NTP implementation containing two programs: the chronyd daemon and a command-line interface called chronyc. As I explained in a previous article, Chrony has some features that make it the best choice for many environments, chiefly:

- Chrony can synchronize to the time server much faster than the old ntpd service. This is good for laptops or desktops that do not run constantly.
- It can compensate for fluctuating clock frequencies, such as when a host hibernates or enters sleep mode, or when the clock speed varies due to frequency stepping that slows clock speeds when loads are low.
- It handles intermittent network connections and bandwidth saturation.
- It adjusts for network delays and latency.
- After the initial time sync, Chrony never stops the clock. This ensures stable and consistent time intervals for many system services and applications.
- Chrony can work even without a network connection. In this case, the local host or server can be updated manually.
- Chrony can act as an NTP server.

Just to be clear, NTP is a protocol that is implemented on a Linux host using either Chrony or the systemd-timesyncd.service.

The NTP, Chrony, and systemd-timesyncd RPM packages are available in standard Fedora repositories. The systemd-udev RPM is a rule-based device node and kernel event manager that is installed by default with Fedora but not enabled.

You can install all three and switch between them, but that is a pain and not worth the trouble. Modern releases of Fedora, CentOS, and RHEL have moved from NTP to Chrony as their default timekeeping implementation, and they also install systemd-timesyncd. I find that

Creative Commons Attribution Share-alike 4.0

Chrony works well, provides a better interface than the NTP service, presents much more information, and increases control, which are all advantages for the sysadmin.

Disable other NTP services

It's possible an NTP service is already running on your host. If so, you need to disable it before switching to something else. I have been using chronyd, so I used the following commands to stop and disable it. Run the appropriate commands for whatever NTP daemon you are using on your host:

```
# systemctl disable chronyd ; systemctl stop chronyd
Removed /etc/systemd/system/multi-user.target.wants/chronyd.service.
```

Verify that it is both stopped and disabled:

```
# systemctl status chronyd
● chronyd.service - NTP client/server
     Loaded: loaded (/usr/lib/systemd/system/chronyd.service; disabled; vendor
preset: enabled)
     Active: inactive (dead)
       Docs: man:chronyd(8)
             man:chrony.conf(5)
```

Check the status before starting

The systemd timesync's status indicates whether systemd has initiated an NTP service. Because you have not yet started systemd NTP, the `timesync-status` command returns no data:

```
# timedatectl timesync-status
Failed to query server: Could not activate remote peer.
```

But a straight `status` request provides some important information. For example, the `timedatectl` command without an argument or options implies the `status` subcommand as default:

```
# timedatectl status
           Local time: Fri 2020-05-15 08:43:10 EDT
           Universal time: Fri 2020-05-15 12:43:10 UTC
                 RTC time: Fri 2020-05-15 08:43:08
                Time zone: America/New_York (EDT, -0400)
System clock synchronized: no
              NTP service: inactive
          RTC in local TZ: yes
```

```
Warning: The system is configured to read the RTC time in the local time zone.
         This mode cannot be fully supported. It will create various problems
         with time zone changes and daylight saving time adjustments. The RTC
         time is never updated, it relies on external facilities to maintain it.
         If at all possible, use RTC in UTC by calling
         'timedatectl set-local-rtc 0'.
```

This returns the local time for your host, the UTC time, and the RTC time. It shows that the system time is set to the `America/New_York` time zone (`TZ`), the RTC is set to the time in the local time zone, and the NTP service is not active. The RTC time has started to drift a bit from the system time. This is normal with systems whose clocks have not been synchronized. The amount of drift on a host depends upon the amount of time since the system was last synced and the speed of the drift per unit of time.

There is also a warning message about using local time for the RTC—this relates to time-zone changes and daylight saving time adjustments. If the computer is off when changes need to be made, the RTC time will not change. This isn't an issue in servers or other hosts that are powered on 24/7. Also, any service that provides NTP time synchronization ensures time is set early in the startup process, so it's correct before it is fully up and running.

Set the time zone

Usually, you set a computer's time zone during the installation procedure and never need to change it. However, there are times it is necessary to change the time zone, and there are a couple of tools to help. Linux uses time-zone files to define the local time zone in use by the host. These binary files are located in the `/usr/share/zoneinfo` directory. The default for my time zone is defined by the link `/etc/localtime -> ../usr/share/zoneinfo/America/New_York`. But you don't need to know that to change the time zone.

But you do need to know the official time-zone name for your location. Say you want to change the time zone to Los Angeles:

```
# timedatectl list-timezones | column
[...]
America/La_Paz                  Europe/Budapest
America/Lima                    Europe/Chisinau
America/Los_Angeles             Europe/Copenhagen
America/Maceio                  Europe/Dublin
America/Managua                 Europe/Gibraltar
[...]
```

Now you can set the time zone. I used the `date` command to verify the change, but you could also use `timedatectl`:

```
# date
Tue 19 May 2020 04:47:49 PM EDT
# timedatectl set-timezone America/Los_Angeles
# date
Tue 19 May 2020 01:48:23 PM PDT
```

You can now change your host's time zone back to your local one.

systemd-timesyncd

The systemd timesync daemon provides an NTP implementation that is easy to manage within a systemd context. It is installed by default in Fedora and Ubuntu and started by default in Ubuntu but not in Fedora. I am unsure about other distros; you can check yours with:

```
# systemctl status systemd-timesyncd
```

Configure systemd-timesyncd

The configuration file for systemd-timesyncd is `/etc/systemd/timesyncd.conf`. It is a simple file with fewer options included than the older NTP service and chronyd. Here are the complete contents of the default version of this file on my Fedora VM:

```
#  This file is part of systemd.
#
# Entries in this file show the compile time defaults.
# You can change settings by editing this file.
# Defaults can be restored by simply deleting this file.
#
# See timesyncd.conf(5) for details.

[Time]
#NTP=
#FallbackNTP=0.fedora.pool.ntp.org 1.fedora.pool.ntp.org 2.fedora.pool.ntp.org
3.fedora.pool.ntp.org
#RootDistanceMaxSec=5
#PollIntervalMinSec=32
#PollIntervalMaxSec=2048
```

The only section it contains besides comments is `[Time]`, and all the lines are commented out. These are the default values and do not need to be changed or uncommented (unless you have some reason to do so). If you do not have a specific NTP time server defined in the `NTP=` line, Fedora's default is to fall back on the Fedora pool of time servers. I like to add the time server on my network to this line:

```
NTP=myntpserver
```

Start timesync

Starting and enabling systemd-timesyncd is just like any other service:

```
# systemctl enable systemd-timesyncd.service
Created symlink /etc/systemd/system/dbus-org.freedesktop.timesync1.service →
/usr/lib/systemd/system/systemd-timesyncd.service.
Created symlink /etc/systemd/system/sysinit.target.wants/systemd-
timesyncd.service → /usr/lib/systemd/system/systemd-timesyncd.service.
# systemctl start systemd-timesyncd.service
```

Set the hardware clock

Here's what one of my systems looked like after starting timesyncd:

```
# timedatectl
               Local time: Sat 2020-05-16 14:34:54 EDT
           Universal time: Sat 2020-05-16 18:34:54 UTC
                 RTC time: Sat 2020-05-16 14:34:53
                Time zone: America/New_York (EDT, -0400)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

The RTC time is around a second off from local time (EDT), and the discrepancy grows by a couple more seconds over the next few days. Because RTC does not have the concept of time zones, the `timedatectl` command must do a comparison to determine which time zone is a match. If the RTC time does not match local time exactly, it is not considered to be in the local time zone.

In search of a bit more information, I checked the status of systemd-timesync.service and found:

```
# systemctl status systemd-timesyncd.service
● systemd-timesyncd.service - Network Time Synchronization
     Loaded: loaded (/usr/lib/systemd/system/systemd-timesyncd.service; enabled;
vendor preset: disabled)
```

```
     Active: active (running) since Sat 2020-05-16 13:56:53 EDT; 18h ago
       Docs: man:systemd-timesyncd.service(8)
   Main PID: 822 (systemd-timesyn)
     Status: "Initial synchronization to time server 163.237.218.19:123
(2.fedora.pool.ntp.org)."
      Tasks: 2 (limit: 10365)
     Memory: 2.8M
        CPU: 476ms
     CGroup: /system.slice/systemd-timesyncd.service
             └─822 /usr/lib/systemd/systemd-timesyncd

May 16 09:57:24 testvm2.both.org systemd[1]: Starting Network Time
Synchronization...
May 16 09:57:24 testvm2.both.org systemd-timesyncd[822]: System clock time unset
or jumped backwards, restoring from recorded timestamp: Sat 2020-05-16 13:56:53
May 16 13:56:53 ...m2.both.org systemd[1]: Started Network Time Synchronization.
May 16 13:57:56 testvm2.both.org systemd-timesyncd[822]: Initial synchronization
to time server 163.237.218.19:123 (2.fedora.pool.ntp.org).
```

Notice the log message that says the system clock time was unset or jumped backward. The timesync service sets the system time from a timestamp. Timestamps are maintained by the timesync daemon and are created at each successful time synchronization.

The `timedatectl` command does not have the ability to set the value of the hardware clock from the system clock; it can only set the time and date from a value entered on the command line. However, you can set the RTC to the same value as the system time by using the `hwclock` command:

```
# /sbin/hwclock --systohc --localtime
# timedatectl
               Local time: Mon 2020-05-18 13:56:46 EDT
           Universal time: Mon 2020-05-18 17:56:46 UTC
                 RTC time: Mon 2020-05-18 13:56:46
                Time zone: America/New_York (EDT, -0400)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: yes
```

The `--localtime` option ensures that the hardware clock is set to local time, not UTC.

Do you really need RTC?

Any NTP implementation will set the system clock during the startup sequence, so is RTC necessary? Not really, so long as you have a network connection to a time server. However, many systems do not have full-time access to a network connection, so the hardware clock is useful so that Linux can read it and set the system time. This is a better solution than having to set the time by hand, even if it might drift away from the actual time.

Summary

This article explored the use of some systemd tools for managing date, time, and time zones. The systemd-timesyncd tool provides a decent NTP client that can keep time on a local host synchronized with an NTP server. However, systemd-timesyncd does not provide a server service, so if you need an NTP server on your network, you must use something else, such as Chrony, to act as a server.

I prefer to have a single implementation for any service in my network, so I use Chrony. If you do not need a local NTP server, or if you do not mind dealing with Chrony for the server and systemd-timesyncd for the client and you do not need Chrony's additional capabilities, then systemd-timesyncd is a serviceable choice for an NTP client.

There is another point I want to make: You do not have to use systemd tools for NTP implementation. You can use the old ntpd or Chrony or some other NTP implementation. systemd is composed of a large number of services; many of them are optional, so they can be disabled and something else used in its place. It is not the huge, monolithic monster that some make it out to be. It is OK to not like systemd or parts of it, but you should make an informed decision.

I don't dislike systemd's implementation of NTP, but I much prefer Chrony because it meets my needs better. And that is what Linux is all about.

Use systemd timers instead of cronjobs

I am in the process of converting my cron jobs to systemd timers. I have used timers for a few years, but usually, I learned just enough to perform the task I was working on. While doing research for this systemd series, I learned that systemd timers have some very interesting capabilities.

Like cron jobs, systemd timers can trigger events—shell scripts and programs—at specified time intervals, such as once a day, on a specific day of the month (perhaps only if it is a Monday), or every 15 minutes during business hours from 8am to 6pm. Timers can also do some things that cron jobs cannot. For example, a timer can trigger a script or program to run a specific amount of time after an event such as boot, startup, completion of a previous task, or even the previous completion of the service unit called by the timer.

System maintenance timers

When Fedora or any systemd-based distribution is installed on a new system, it creates several timers that are part of the system maintenance procedures that happen in the background of any Linux host. These timers trigger events necessary for common maintenance tasks, such as updating system databases, cleaning temporary directories, rotating log files, and more.

As an example, I'll look at some of the timers on my primary workstation by using the `systemctl status *timer` command to list all the timers on my host. The asterisk symbol works the same as it does for file globbing, so this command lists all systemd timer units:

```
# systemctl status *timer
● mlocate-updatedb.timer - Updates mlocate database every day
     Loaded: loaded (/usr/lib/systemd/system/mlocate-updatedb.timer; enabled;
vendor preset: enabled)
```

Creative Commons Attribution Share-alike 4.0

```
     Active: active (waiting) since Tue 2020-06-02 08:02:33 EDT; 2 days ago
    Trigger: Fri 2020-06-05 00:00:00 EDT; 15h left
   Triggers: ● mlocate-updatedb.service

Jun 02 08:02:33 testvm1.both.org systemd[1]: Started Updates mlocate database
every day.

● logrotate.timer - Daily rotation of log files
     Loaded: loaded (/usr/lib/systemd/system/logrotate.timer; enabled; vendor
preset: enabled)
     Active: active (waiting) since Tue 2020-06-02 08:02:33 EDT; 2 days ago
    Trigger: Fri 2020-06-05 00:00:00 EDT; 15h left
   Triggers: ● logrotate.service
       Docs: man:logrotate(8)
             man:logrotate.conf(5)

Jun 02 08:02:33 testvm1.both.org systemd[1]: Started Daily rotation of log files.
[...]
Jun 02 08:02:33 testvm1.both.org systemd[1]: Started Run system activity
accounting tool every 10 minutes.
```

Each timer has at least six lines of information associated with it:

- The first line has the timer's file name and a short description of its purpose.
- The second line displays the timer's status, whether it is loaded, the full path to the timer unit file, and the vendor preset.
- The third line indicates its active status, which includes the date and time the timer became active.
- The fourth line contains the date and time the timer will be triggered next and an approximate time until the trigger occurs.
- The fifth line shows the name of the event or the service that is triggered by the timer.
- Some (but not all) systemd unit files have pointers to the relevant documentation. Three of the timers in my virtual machine's output have pointers to documentation. This is a nice (but optional) bit of data.
- The final line is the journal entry for the most recent instance of the service triggered by the timer.

Depending upon your host, you probably have a different set of timers.

Create a timer

Although we can deconstruct one or more of the existing timers to learn how they work, let’s create our own service unit and a timer unit to trigger it. We will use a fairly trivial example in

Creative Commons Attribution Share-alike 4.0

order to keep this simple. After we have finished this, it will be easier to understand how the other timers work and to determine what they are doing.

First, create a simple service that will run something basic, such as the `free` command. For example, you may want to monitor free memory at regular intervals. Create the following `myMonitor.service` unit file in the `/etc/systemd/system` directory. It does not need to be executable:

```
# This service unit is for testing timer units
# By David Both
# Licensed under GPL V2
#

[Unit]
Description=Logs system statistics to the systemd journal
Wants=myMonitor.timer

[Service]
Type=oneshot
ExecStart=/usr/bin/free

[Install]
WantedBy=multi-user.target
```

Now let’s look at the status and test our service unit to ensure that it works as we expect it to.

```
# systemctl status myMonitor.service
● myMonitor.service - Logs system statistics to the systemd journal
     Loaded: loaded (/etc/systemd/system/myMonitor.service; disabled; vendor
preset: disabled)
     Active: inactive (dead)
# systemctl start myMonitor.service
```

Where is the output? By default, the standard output (`STDOUT`) from programs run by systemd service units is sent to the systemd journal, which leaves a record you can view now or later—up to a point. (I will look at systemd journaling and retention strategies in a future article in this series.) Look at the journal specifically for your service unit and for today only. The `-S` option, which is the short version of `--since`, allows you to specify the time period that the `journalctl` tool should search for entries. This isn't because you don't care about previous results—in this case, there won't be any—it is to shorten the search time if your host has been running for a long time and has accumulated a large number of entries in the journal:

```
# journalctl -S today -u myMonitor.service
```

```
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Thu 2020-06-11 09:40:47 EDT.
--
Jun 11 09:12:09 testvm1.both.org systemd[1]: Starting Logs system statistics to the systemd journal...
Jun 11 09:12:09 testvm1.both.org free[377966]: total     used    free   shared  buff/cache   available
Jun 11 09:12:09 testvm1.both.org free[377966]: Mem:       12635740      522868    11032860        8016
1080012    11821508
Jun 11 09:12:09 testvm1.both.org free[377966]: Swap:       8388604           0     8388604
Jun 11 09:12:09 testvm1.both.org systemd[1]: myMonitor.service: Succeeded.
```

A task triggered by a service can be a single program, a series of programs, or a script written in any scripting language. Add another task to the service by adding the following line to the end of the `[Service]` section of the `myMonitor.service` unit file:

```
ExecStart=/usr/bin/lsblk
```

Start the service again and check the journal for the results, which should look like this. You should see the results from both commands in the journal:

```
Jun 11 15:42:18 testvm1.both.org systemd[1]: Starting Logs system statistics to the systemd journal...
Jun 11 15:42:18 testvm1.both.org free[379961]:               total        used        free      shared
buff/cache   available
Jun 11 15:42:18 testvm1.both.org free[379961]: Mem:       12635740      531788    11019540     8024...
Jun 11 15:42:18 testvm1.both.org free[379961]: Swap:       8388604           0     8388604
Jun 11 15:42:18 testvm1.both.org lsblk[379962]: NAME          MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
Jun 11 15:42:18 testvm1.both.org lsblk[379962]: sda             8:0    0  120G  0 disk
Jun 11 15:42:18 testvm1.both.org lsblk[379962]: ├─sda1          8:1    0    4G  0 part /boot
Jun 11 15:42:18 testvm1.both.org lsblk[379962]: └─sda2          8:2    0  116G  0 part
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   ├─VG01-root 253:0    0    5G  0 lvm  /
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   ├─VG01-swap 253:1    0    8G  0 lvm  [SWAP]
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   ├─VG01-usr  253:2    0   30G  0 lvm  /usr
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   ├─VG01-tmp  253:3    0   10G  0 lvm  /tmp
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   ├─VG01-var  253:4    0   20G  0 lvm  /var
Jun 11 15:42:18 testvm1.both.org lsblk[379962]:   └─VG01-home 253:5    0   10G  0 lvm  /home
Jun 11 15:42:18 testvm1.both.org lsblk[379962]: sr0            11:0    1 1024M  0 rom
Jun 11 15:42:18 testvm1.both.org systemd[1]: myMonitor.service: Succeeded.
Jun 11 15:42:18 testvm1.both.org systemd[1]: Finished Logs system statistics to the systemd journal.
```

Now that you know your service works as expected, create the timer unit file, `myMonitor.timer` in `/etc/systemd/system`, and add the following:

```
# This timer unit is for testing
# By David Both
# Licensed under GPL V2

[Unit]
Description=Logs some system statistics to the systemd journal
Requires=myMonitor.service

[Timer]
Unit=myMonitor.service
OnCalendar=*-*-* *:*:00
```

```
[Install]
WantedBy=timers.target
```

The `OnCalendar` time specification in the `myMonitor.timer file`, `*-*-* *:*:00`, should trigger the timer to execute the `myMonitor.service` unit every minute. I will explore `OnCalendar` settings a bit later in this article.

For now, observe any journal entries pertaining to running your service when it is triggered by the timer. You could also follow the timer, but following the service allows you to see the results in near real time. Run `journalctl` with the `-f` (follow) option:

```
# journalctl -S today -f -u myMonitor.service
-- Logs begin at Mon 2020-06-08 07:47:20 EDT. --
```

Start but do not enable the timer, and see what happens after it runs for a while:

```
# systemctl start myMonitor.service
```

One result shows up right away, and the next ones come at—sort of—one-minute intervals. Watch the journal for a few minutes and see if you notice the same things I did:

```
# journalctl -S today -f -u myMonitor.service
-- Logs begin at Mon 2020-06-08 07:47:20 EDT. --
Jun 13 08:39:18 testvm1.both.org systemd[1]: Starting Logs system statistics to the systemd journal...
Jun 13 08:39:18 testvm1.both.org systemd[1]: myMonitor.service: Succeeded.
Jun 13 08:39:19 testvm1.both.org free[630566]:               total        used        free   shared...
Jun 13 08:39:19 testvm1.both.org free[630566]: Mem:       12635740      556604    10965516     8036...
Jun 13 08:39:19 testvm1.both.org free[630566]: Swap:       8388604           0     8388604
Jun 13 08:39:18 testvm1.both.org systemd[1]: Finished Logs system statistics to the systemd journal.
Jun 13 08:39:19 testvm1.both.org lsblk[630567]: NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
Jun 13 08:39:19 testvm1.both.org lsblk[630567]: sda             8:0    0  120G  0 disk
Jun 13 08:39:19 testvm1.both.org lsblk[630567]: ├─sda1          8:1    0    4G  0 part /boot
Jun 13 08:39:19 testvm1.both.org lsblk[630567]: └─sda2          8:2    0  116G  0 part
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   ├─VG01-root 253:0    0    5G  0 lvm  /
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   ├─VG01-swap 253:1    0    8G  0 lvm  [SWAP]
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   ├─VG01-usr  253:2    0   30G  0 lvm  /usr
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   ├─VG01-tmp  253:3    0   10G  0 lvm  /tmp
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   ├─VG01-var  253:4    0   20G  0 lvm  /var
Jun 13 08:39:19 testvm1.both.org lsblk[630567]:   └─VG01-home 253:5    0   10G  0 lvm  /home
Jun 13 08:39:19 testvm1.both.org lsblk[630567]: sr0            11:0    1 1024M  0 rom
Jun 13 08:40:46 testvm1.both.org systemd[1]: Starting Logs system statistics to the systemd journal...
Jun 13 08:40:46 testvm1.both.org free[630572]:               total        used        free   shared...
Jun 13 08:40:46 testvm1.both.org free[630572]: Mem:       12635740      555228    10966836     8036...
Jun 13 08:40:46 testvm1.both.org free[630572]: Swap:       8388604           0     8388604
Jun 13 08:40:46 testvm1.both.org lsblk[630574]: NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
Jun 13 08:40:46 testvm1.both.org lsblk[630574]: sda             8:0    0  120G  0 disk
Jun 13 08:40:46 testvm1.both.org lsblk[630574]: ├─sda1          8:1    0    4G  0 part /boot
Jun 13 08:40:46 testvm1.both.org lsblk[630574]: └─sda2          8:2    0  116G  0 part
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   ├─VG01-root 253:0    0    5G  0 lvm  /
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   ├─VG01-swap 253:1    0    8G  0 lvm  [SWAP]
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   ├─VG01-usr  253:2    0   30G  0 lvm  /usr
```

```
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   ├─VG01-tmp  253:3    0   10G  0 lvm  /tmp
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   ├─VG01-var  253:4    0   20G  0 lvm  /var
Jun 13 08:40:46 testvm1.both.org lsblk[630574]:   └─VG01-home 253:5    0   10G  0 lvm  /home
Jun 13 08:40:46 testvm1.both.org lsblk[630574]: sr0           11:0    1 1024M  0 rom
Jun 13 08:40:46 testvm1.both.org systemd[1]: myMonitor.service: Succeeded.
Jun 13 08:40:46 testvm1.both.org systemd[1]: Finished Logs system statistics to the systemd journal.
Jun 13 08:41:46 testvm1.both.org systemd[1]: Starting Logs system statistics to the systemd journal...
Jun 13 08:41:46 testvm1.both.org free[630580]:               total        used        free   shared...
Jun 13 08:41:46 testvm1.both.org free[630580]: Mem:       12635740      553488    10968564     8036...
Jun 13 08:41:46 testvm1.both.org free[630580]: Swap:       8388604           0     8388604
Jun 13 08:41:47 testvm1.both.org lsblk[630581]: NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
Jun 13 08:41:47 testvm1.both.org lsblk[630581]: sda             8:0    0  120G  0 disk
Jun 13 08:41:47 testvm1.both.org lsblk[630581]: ├─sda1          8:1    0    4G  0 part /boot
Jun 13 08:41:47 testvm1.both.org lsblk[630581]: └─sda2          8:2    0  116G  0 part
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   ├─VG01-root 253:0    0    5G  0 lvm  /
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   ├─VG01-swap 253:1    0    8G  0 lvm  [SWAP]
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   ├─VG01-usr  253:2    0   30G  0 lvm  /usr
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   ├─VG01-tmp  253:3    0   10G  0 lvm  /tmp
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   ├─VG01-var  253:4    0   20G  0 lvm  /var
Jun 13 08:41:47 testvm1.both.org lsblk[630581]:   └─VG01-home 253:5    0   10G  0 lvm  /home
Jun 13 08:41:47 testvm1.both.org lsblk[630581]: sr0           11:0    1 1024M  0 rom
Jun 13 08:41:47 testvm1.both.org systemd[1]: myMonitor.service: Succeeded.
Jun 13 08:41:47 testvm1.both.org systemd[1]: Finished Logs system statistics to the systemd journal.
```

Be sure to check the status of both the timer and the service.

You probably noticed at least two things in the journal. First, you do not need to do anything special to cause the `STDOUT` from the `ExecStart` triggers in the `myMonitor.service` unit to be stored in the journal. That is all part of using systemd for running services. However, it does mean that you might need to be careful about running scripts from a service unit and how much `STDOUT` they generate.

The second thing is that the timer does not trigger exactly on the minute at :00 seconds or even exactly one minute from the previous instance. This is intentional, but it can be overridden if necessary (or if it just offends your sysadmin sensibilities).

The reason for this behavior is to prevent multiple services from triggering at exactly the same time. For example, you can use time specifications such as Weekly, Daily, and more. These shortcuts are all defined to trigger at 00:00:00 hours on the day they are triggered. When multiple timers are specified this way, there is a strong likelihood that they would attempt to start simultaneously.

systemd timers are intentionally designed to trigger somewhat randomly around the specified time to try to prevent simultaneous triggers. They trigger semi-randomly within a time window that starts at the specified trigger time and ends at the specified time plus one minute. This trigger time is maintained at a stable position with respect to all other defined timer units, according to the `systemd.timer` man page. You can see in the journal entries above that the timer triggered immediately when it started and then about 46 or 47 seconds after each minute.

Most of the time, such probabilistic trigger times are fine. When scheduling tasks such as backups to run, so long as they run during off-hours, there will be no problems. A sysadmin can select a deterministic start time, such as 01:05:00 in a typical cron job specification, to not conflict with other tasks, but there is a large range of time values that will accomplish that. A one-minute bit of randomness in a start time is usually irrelevant.

However, for some tasks, exact trigger times are an absolute requirement. For those, you can specify greater trigger time-span accuracy (to within a microsecond) by adding a statement like this to the `Timer` section of the timer unit file:

```
AccuracySec=1us
```

Time spans can be used to specify the desired accuracy as well as to define time spans for repeating or one-time events. It recognizes the following units:

- usec, us, µs
- msec, ms
- seconds, second, sec, s
- minutes, minute, min, m
- hours, hour, hr, h
- days, day, d
- weeks, week, w
- months, month, M (defined as 30.44 days)
- years, year, y (defined as 365.25 days)

All the default timers in `/usr/lib/systemd/system` specify a much larger range for accuracy because exact times are not critical. Look at some of the specifications in the system-created timers:

```
# grep Accur /usr/lib/systemd/system/*timer
/usr/lib/systemd/system/fstrim.timer:AccuracySec=1h
/usr/lib/systemd/system/logrotate.timer:AccuracySec=1h
/usr/lib/systemd/system/logwatch.timer:AccuracySec=12h
/usr/lib/systemd/system/mlocate-updatedb.timer:AccuracySec=24h
/usr/lib/systemd/system/raid-check.timer:AccuracySec=24h
/usr/lib/systemd/system/unbound-anchor.timer:AccuracySec=24h
```

View the complete contents of some of the timer unit files in the `/usr/lib/systemd/system` directory to see how they are constructed.

You do not have to enable the timer in this experiment to activate it at boot time, but the command to do so would be:

```
# systemctl enable myMonitor.timer
```

The unit files you created do not need to be executable. You also did not enable the service unit because it is triggered by the timer. You can still trigger the service unit manually from the command line, should you want to. Try that and observe the journal.

See the man pages for `systemd.timer` and `systemd.time` for more information about timer accuracy, event-time specifications, and trigger events.

Timer types

systemd timers have other capabilities that are not found in cron, which triggers only on specific, repetitive, real-time dates and times. systemd timers can be configured to trigger based on status changes in other systemd units. For example, a timer might be configured to trigger a specific elapsed time after system boot, after startup, or after a defined service unit activates. These are called monotonic timers. Monotonic refers to a count or sequence that continually increases. These timers are not persistent because they reset after each boot.

Table 1 lists the monotonic timers along with a short definition of each, as well as the `OnCalendar` timer, which is not monotonic and is used to specify future times that may or may not be repetitive. This information is derived from the `systemd.timer` man page with a few minor changes.

Timer	Monotonic	Definition
OnActiveSec=	X	This defines a timer relative to the moment the timer is activated.
OnBootSec=	X	This defines a timer relative to when the machine boots up.
OnStartupSec=	X	This defines a timer relative to when the service manager first starts. For system timer units, this is very similar to OnBootSec=, as the system service manager generally starts very early at boot. It's primarily useful when configured in units running in the per-user service manager, as the user service manager generally starts on first login only, not during boot.

Timer	Monotonic	Definition
OnUnitActiveSec=	X	This defines a timer relative to when the timer that is to be activated was last activated.
OnUnitInactiveSec=	X	This defines a timer relative to when the timer that is to be activated was last deactivated.
OnCalendar=		This defines real-time (i.e., wall clock) timers with calendar event expressions. See systemd.time(7) for more information on the syntax of calendar event expressions. Otherwise, the semantics are similar to OnActiveSec= and related settings. This timer is the one most like those used with the cron service.

Table 1: systemd timer definitions

Monotonic timers can use the same shortcut names for their time spans as the `AccuracySec` statement mentioned before, but systemd normalizes those names to seconds. For example, you might want to specify a timer that triggers an event one time, five days after the system boots; that might look like: `OnBootSec=5d`. If the host booted at `2020-06-15 09:45:27`, the timer would trigger at `2020-06-20 09:45:27` or within one minute after.

Calendar event specifications

Calendar event specifications are a key part of triggering timers at desired repetitive times. Start by looking at some specifications used with the `OnCalendar` setting.

systemd and its timers use a different style for time and date specifications than the format used in crontab. It is more flexible than crontab and allows fuzzy dates and times in the manner of the `at` command. It should also be familiar enough that it will be easy to understand.

The basic format for systemd timers using `OnCalendar=` is `DOW YYYY-MM-DD HH:MM:SS`. DOW (day of week) is optional, and other fields can use an asterisk (*) to match any value for that position. All calendar time forms are converted to a normalized form. If the time is not specified, it is assumed to be 00:00:00. If the date is not specified but the time is, the next match might be today or tomorrow, depending upon the current time. Names or numbers can be used for the month and day of the week. Comma-separated lists of each unit can be specified. Unit ranges can be specified with `..` between the beginning and ending values.

There are a couple interesting options for specifying dates. The Tilde (~) can be used to specify the last day of the month or a specified number of days prior to the last day of the month. The "/" can be used to specify a day of the week as a modifier.

Here are some examples of some time specifications used in `OnCalendar` statements.

Calendar event specification	Description
DOW YYYY-MM-DD HH:MM:SS	
--* 00:15:30	Every day of every month of every year at 15 minutes and 30 seconds after midnight
Weekly	Every Monday at 00:00:00
Mon *-*-* 00:00:00	Same as weekly
Mon	Same as weekly
Wed 2020-*-*	Every Wednesday in 2020 at 00:00:00
Mon..Fri 2021-*-*	Every weekday in 2021 at 00:00:00
2022-6,7,8-1,15 01:15:00	The 1st and 15th of June, July, and August of 2022 at 01:15:00am
Mon *-05~03	The next occurrence of a Monday in May of any year which is also the 3rd day from the end of the month.
Mon..Fri *-08~04	The 4th day preceding the end of August for any years in which it also falls on a weekday.
*-05~03/2	The 3rd day from the end of the month of May and then again two days later. Repeats every year. Note that this expression uses the Tilde (~).
*-05-03/2	The third day of the month of may and then every 2nd day for the rest of May. Repeats every year. Note that this expression uses the dash (-).

Table 2: Sample `OnCalendar` event specifications

Test calendar specifications

systemd provides an excellent tool for validating and examining calendar time event specifications in a timer. The `systemd-analyze calendar` tool parses a calendar time event specification and provides the normalized form as well as other interesting information such as the date and time of the next "elapse," i.e., match, and the approximate amount of time before the trigger time is reached.

First, look at a date in the future without a time (note that the times for `Next elapse` and `UTC` will differ based on your local time zone):

```
# systemd-analyze calendar 2030-06-17
  Original form: 2030-06-17
Normalized form: 2030-06-17 00:00:00
    Next elapse: Mon 2030-06-17 00:00:00 EDT
```

Creative Commons Attribution Share-alike 4.0

```
       (in UTC): Mon 2030-06-17 04:00:00 UTC
       From now: 10 years 0 months left
```

Now add a time. In this example, the date and time are analyzed separately as non-related entities:

```
# systemd-analyze calendar 2030-06-17 15:21:16
  Original form: 2030-06-17
Normalized form: 2030-06-17 00:00:00
    Next elapse: Mon 2030-06-17 00:00:00 EDT
       (in UTC): Mon 2030-06-17 04:00:00 UTC
       From now: 10 years 0 months left

  Original form: 15:21:16
Normalized form: *-*-* 15:21:16
    Next elapse: Mon 2020-06-15 15:21:16 EDT
       (in UTC): Mon 2020-06-15 19:21:16 UTC
       From now: 3h 55min left
```

To analyze the date and time as a single unit, enclose them together in quotes. Be sure to remove the quotes when using them in the `OnCalendar=` event specification in a timer unit or you get errors:

```
# systemd-analyze calendar "2030-06-17 15:21:16"
Normalized form: 2030-06-17 15:21:16
    Next elapse: Mon 2030-06-17 15:21:16 EDT
       (in UTC): Mon 2030-06-17 19:21:16 UTC
       From now: 10 years 0 months left
```

Now test the entries in Table 2. I like the last one, especially:

```
# systemd-analyze calendar "2022-6,7,8-1,15 01:15:00"
  Original form: 2022-6,7,8-1,15 01:15:00
Normalized form: 2022-06,07,08-01,15 01:15:00
    Next elapse: Wed 2022-06-01 01:15:00 EDT
       (in UTC): Wed 2022-06-01 05:15:00 UTC
       From now: 1 years 11 months left
```

Look at this example, in which I list the next five elapses for the timestamp expression:

```
# systemd-analyze calendar --iterations=5 "Mon *-05~3"
  Original form: Mon *-05~3
Normalized form: Mon *-05~03 00:00:00
    Next elapse: Mon 2023-05-29 00:00:00 EDT
       (in UTC): Mon 2023-05-29 04:00:00 UTC
       From now: 2 years 11 months left
       Iter. #2: Mon 2028-05-29 00:00:00 EDT
       (in UTC): Mon 2028-05-29 04:00:00 UTC
       From now: 7 years 11 months left
```

```
        Iter. #3: Mon 2034-05-29 00:00:00 EDT
        (in UTC): Mon 2034-05-29 04:00:00 UTC
        From now: 13 years 11 months left
        Iter. #4: Mon 2045-05-29 00:00:00 EDT
        (in UTC): Mon 2045-05-29 04:00:00 UTC
        From now: 24 years 11 months left
        Iter. #5: Mon 2051-05-29 00:00:00 EDT
        (in UTC): Mon 2051-05-29 04:00:00 UTC
        From now: 30 years 11 months left
```

This gives you enough information to start testing your `OnCalendar` time specifications. The `systemd-analyze` tool can be used for other interesting analyses, which I will begin to explore in the next article in this series.

Summary

systemd timers can be used to perform the same kinds of tasks as the cron tool but offer more flexibility in terms of the calendar and monotonic time specifications for triggering events.

Even though the service unit you created for this experiment is usually triggered by the timer, you can also use the `systemctl start myMonitor.service` command to trigger it at any time. Multiple maintenance tasks can be scripted in a single timer; these can be Bash scripts or Linux utility programs. You can run the service triggered by the timer to run all the scripts, or you can run individual scripts as needed.

I have not yet seen any indication that `cron` and `at` will be deprecated. I hope that does not happen because `at` is much easier to use for one-off task scheduling than systemd timers!

Creative Commons Attribution Share-alike 4.0

systemd calendar and timespans

systemd uses calendar time, specifying one or more moments in time to trigger events (such as a backup program), as well as timestamped entries in the journal. It can also use timespans, which define the amount of time between two events but are not directly tied to specific calendar times.

In this chapter, I will look in more detail at how time and date are used and specified in systemd. Also, because systemd uses two slightly different, non-compatible time formats, I will explain how and when they are used.

Definitions

Here are some important time- and calendar-related systemd terms to understand:

- **Absolute timestamp:** A single unambiguous and unique point in time defined in the format `YYYY-MM-DD HH:MM:SS`. The timestamp format specifies points in time when events are triggered by timers. An absolute timestamp can represent only a single point in time, such as `2025-04-15 13:21:05`.
- **Accuracy** is the quality of closeness to the true time; in other words, how close to the specified calendar time an event is triggered by a timer. The default accuracy for systemd timers is defined as a one-minute timespan that starts at the defined calendar time. For example, an event specified to occur at the `OnCalendar` time of 09:52:17 might be triggered at any time between then and 09:53:17.
- **Calendar events** are one or more specific times specified by a systemd timestamp in the format `YYYY-MM-DD HH:MM:SS`. It can be a single point in time or a series of points that are well-defined and for which the exact times can be calculated. systemd journals use timestamps to mark each event with the exact time it occurred.

In systemd, exact time is specified in the timestamp format `YYYY-MM-DD HH:MM:SS`. When only the `YYYY-MM-DD` portion is specified, the time defaults to 00:00:00. When only the `HH:MM:SS` portion is specified, the date is the next calendar instance of that time. If the time specified is before the current time, the next instance will be tomorrow, and if the specified time is later than the current time, the next instance will be today. This is the format systemd timers use to express `OnCalendar` times.

Recurring calendar events can be specified using special characters and formats that represent fields with multiple value matches. For example, `2026-08-15..25 12:15:00` represents 12:15pm on the 15th through the 25th of August 2026 and would trigger 11 matches. Calendar events can also be specified with an absolute timestamp.

- **Timespan** is the amount of time between two events or the duration of something like an event or the time between two events. Timespans can be used to specify the desired accuracy for an event to be triggered by a timer and to define the time to elapse between events. systemd recognizes the following time units:
  - usec, us, µs
  - msec, ms
  - seconds, second, sec, s
  - minutes, minute, min, m
  - hours, hour, hr, h
  - days, day, d
  - weeks, week, w
  - months, month, M (defined as 30.44 days)
  - years, year, y (defined as 365.25 days)

The `systemd.time(7)` man page has a complete description of time and date expressions in timers and other systemd tools.

Calendar event expressions

Calendar event expressions are a key part of triggering timers at repetitive times. systemd and its timers don't use the same style for time and date expressions as crontab uses. systemd is also more flexible than crontab and allows fuzzy dates and times similar to the `at` command.

The format `OnCalendar=` uses for calendar event expressions is `DOW YYYY-MM-DD HH:MM:SS`. DOW (day of the week) is optional, and other fields can use an asterisk (`*`) to match any value for that position. If the time is not specified, it is assumed to be 00:00:00. If the date is not specified but the time is, the next match might be today or tomorrow, depending upon the current time. All the various calendar time-expression formats are converted to a normalized form, and the `systemd-analyze calendar` command shows the normalized form of the time expression.

systemd provides an excellent tool for validating and examining calendar events used in an expression. The `systemd-analyze calendar` tool parses a calendar time event expression and provides the normalized form and other information, such as the date and time of the next "elapse" (match) and the approximate amount of time before it reaches the trigger time.

> **Note:** All the following commands can be performed by non-root users and the times for "Next elapse" and "UTC" differ based on your local time zone.

First, look at the syntax of the systemd-analyze calendar command. Start with a date in the future without a time. Because all the date unit fields are explicitly specified, this is a one-time event:

```
$ systemd-analyze calendar 2030-06-17
  Original form: 2030-06-17
Normalized form: 2030-06-17 00:00:00
    Next elapse: Mon 2030-06-17 00:00:00 EDT
       (in UTC): Mon 2030-06-17 04:00:00 UTC
       From now: 10 years 0 months left
```

Add a time (in this example, the date and time are analyzed separately as non-related entities):

```
$ systemd-analyze calendar 2030-06-17 15:21:16
  Original form: 2030-06-17
Normalized form: 2030-06-17 00:00:00
    Next elapse: Mon 2030-06-17 00:00:00 EDT
       (in UTC): Mon 2030-06-17 04:00:00 UTC
       From now: 10 years 0 months left

  Original form: 15:21:16
Normalized form: *-*-* 15:21:16
    Next elapse: Mon 2020-06-15 15:21:16 EDT
       (in UTC): Mon 2020-06-15 19:21:16 UTC
       From now: 3h 55min left
```

To analyze the date and time as a single entity, enclose them together in quotes:

```
$ systemd-analyze calendar "2030-06-17 15:21:16"
Normalized form: 2030-06-17 15:21:16
    Next elapse: Mon 2030-06-17 15:21:16 EDT
       (in UTC): Mon 2030-06-17 19:21:16 UTC
       From now: 10 years 0 months left
```

Specify one time earlier than the current time and one later. In this example, the current time is 16:16 on 2019-05-15:

```
$ systemd-analyze calendar 15:21:16 22:15
  Original form: 15:21:16
Normalized form: *-*-* 15:21:16
    Next elapse: Fri 2019-05-17 15:21:16 EDT
       (in UTC): Fri 2019-05-17 19:21:16 UTC
       From now: 23h left

  Original form: 22:15
Normalized form: *-*-* 22:15:00
    Next elapse: Thu 2019-05-16 22:15:00 EDT
       (in UTC): Fri 2019-05-17 02:15:00 UTC
       From now: 5h 59min left
```

The `systemd-analyze calendar` tool does not work on timestamps. So things like "tomorrow" or "today" will cause errors if you use them with the calendar sub-command because they are timestamps rather than `OnCalendar` time formats:

```
$ systemd-analyze calendar "tomorrow"
Failed to parse calendar expression 'tomorrow': Invalid argument
Hint: this expression is a valid timestamp. Use 'systemd-analyze timestamp
"tomorrow"' instead?
```

The term "tomorrow" always resolves to tomorrow's date and a time of 00:00:00. You must use the normalized expression format, `YYYY-MM-DD HH:MM:SS`, for this tool to work in calendar mode. Despite this, the `systemd-analyze calendar` tool can still help you understand the structure of the calendar time expressions used by systemd timers. I recommend reading the `systemd.time(7)` man page for a better understanding of the time formats that can be used with systemd timers.

Why would using a statement like `OnCalendar=tomorrow` fail when used in a timer?

Timestamps

Whereas calendar times can be used to match single or multiple points in time, timestamps unambiguously represent a single point in time. For example, timestamps in the systemd journal refer to a precise moment when each logged event occurs:

```
$ journalctl -S today
Hint: You are currently not seeing messages from other users and the system.
      Users in groups 'adm', 'systemd-journal', 'wheel' can see all messages.
      Pass -q to turn off this notice.
-- Logs begin at Wed 2020-06-17 10:08:41 EDT, end at Wed 2020-06-17 10:13:55 EDT.
--
Jun 17 10:08:41 testvm1.both.org systemd[1137785]: Started Mark boot as
successful after the user session has run 2 minutes.
Jun 17 10:08:41 testvm1.both.org systemd[1137785]: Started Daily Cleanup of
User's Temporary Directories.
Jun 17 10:08:41 testvm1.both.org systemd[1137785]: Reached target Paths.
Jun 17 10:08:41 testvm1.both.org systemd[1137785]: Reached target Timers.
[...]
Jun 17 10:13:55 testvm1.both.org systemd[1137785]: systemd-tmpfiles-
clean.service: Succeeded.
Jun 17 10:13:55 testvm1.both.org systemd[1137785]: Finished Cleanup of User's
Temporary Files and Directories.
```

The `systemd-analyze timestamp` command can be used to analyze timestamp expressions the same way it analyzes calendar expressions. Here is an example from the journal data stream:

```
$ systemd-analyze timestamp "Jun 17 10:08:41"
Failed to parse "Jun 17 10:08:41": Invalid argument
[student@testvm1 ~]$ systemd-analyze timestamp Jun 17 10:08:41
Failed to parse "Jun": Invalid argument

Failed to parse "17": Invalid argument
Hint: this expression is a valid timespan. Use 'systemd-analyze timespan "17"'
instead?

  Original form: 10:08:41
Normalized form: Wed 2020-06-17 10:08:41 EDT
       (in UTC): Wed 2020-06-17 14:08:41 UTC
   UNIX seconds: @1592402921
       From now: 11min ago
```

Why is the data copied from the journal not considered to be a valid timestamp? I have no idea. It seems pretty dumb to print the timestamps for the journal in a format that cannot be used by the tool designed to analyze timestamps. But look at the time specified as the beginning of the log:

```
$ systemd-analyze timestamp "Wed 2020-06-17 10:08:41"
  Original form: Wed 2020-06-17 10:08:41
Normalized form: Wed 2020-06-17 10:08:41 EDT
       (in UTC): Wed 2020-06-17 14:08:41 UTC
   UNIX seconds: @1592402921
       From now: 15min ago
```

OK, so that time is a valid timestamp. The key is that the `systemd-analyze` tool only recognizes the `DOW YYYY-MM-DD HH:MM:SS` format. As the error messages state, it does not recognize month names or standalone days of the month, such as 17. It is very exacting because the only way to ensure events are triggered by timers at desired points in time or intervals is to be completely accurate with how they are specified.

Any unambiguously expressed time, such as `2020-06-17 10:08:41`, is a timestamp because it can only occur once. A timestamp that will occur in the future can also be used in a systemd timer, and that timer will only trigger its defined action once.

A time expressed somewhat more ambiguously, such as `2025-*-* 22:15:00`, can only be a calendar time used in the `OnCalendar` statement in a timer unit file. This expression will trigger an event every day in the year 2025 at 22:15:00 (10:15:00pm).

But still—why not use valid timestamps in the journal output? I *still* don't know, but with a bit of command-line manipulation, you can convert the default time display into valid timestamps. The `journalctl` command tool has some options that can display the timestamps in a format you can easily use with the `systemd-analyze` tool:

```
# journalctl -o short-full
[...]
Fri 2020-06-26 12:51:36 EDT testvm1.both.org systemd[1]: Finished Update UTMP
about System Runlevel Changes.
Fri 2020-06-26 12:51:36 EDT testvm1.both.org systemd[1]: Startup finished in
2.265s (kernel) + 4.883s (initrd) + 22.645s (userspace) = 29.793s.
Fri 2020-06-26 12:51:36 EDT testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=systemd-update-utmp-runlevel>
Fri 2020-06-26 12:51:36 EDT testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=systemd-update-utmp-runlevel >
Fri 2020-06-26 12:51:36 EDT testvm1.both.org crond[959]: (CRON) INFO (running
with inotify support)
Fri 2020-06-26 12:51:36 EDT testvm1.both.org ModemManager[759]: <info>  Couldn't
check support for device '/sys/devices/pci0000:00/0000:00:03.0': not >
Fri 2020-06-26 12:51:37 EDT testvm1.both.org VBoxService[804]: 16:51:37.196960
timesync vgsvcTimeSyncWorker: Radical guest time change: -14 388 436 32>
```

```
Fri 2020-06-26 12:51:39 EDT testvm1.both.org chronyd[827]: Selected source
192.168.0.52
Fri 2020-06-26 12:51:39 EDT testvm1.both.org chronyd[827]: System clock TAI
offset set to 37 seconds
[...]
```

You can now use the timestamps like this:

```
# systemd-analyze timestamp "2020-06-26 12:51:36"
  Original form: 2020-06-26 12:51:36
Normalized form: Fri 2020-06-26 12:51:36 EDT
       (in UTC): Fri 2020-06-26 16:51:36 UTC
   UNIX seconds: @1593190296
       From now: 2h 37min ago
```

You can also display the journal timestamps in a monotonic format that shows the number of seconds since boot:

```
# journalctl -o short-monotonic
[    0.000000] testvm1.both.org kernel: Linux version 5.6.6-300.fc32.x86_64
(mockbuild@bkernel03.phx2.fedoraproject.org) (gcc version 10.0.1 20200328 >
[    0.000000] testvm1.both.org kernel: Command line:
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.6.6-300.fc32.x86_64 root=/dev/mapper/VG01-root
ro resume=/dev/>
[    0.000000] testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x001:
'x87 floating point registers'
[...]
[    0.000000] testvm1.both.org kernel: x86/fpu: xstate_offset[2]:  576,
xstate_sizes[2]:  256
[    0.000000] testvm1.both.org kernel: x86/fpu: Enabled xstate features 0x7,
context size is 832 bytes, using 'standard' format.
[    0.000000] testvm1.both.org kernel: BIOS-provided physical RAM map:
[    0.000000] testvm1.both.org kernel: BIOS-e820: [mem 0x0000000000000000-
0x000000000009fbff] usable
[    0.000000] testvm1.both.org kernel: BIOS-e820: [mem 0x000000000009fc00-
0x000000000009ffff] reserved
[...]
[29337.443404] testvm1.both.org systemd[1]: sysstat-collect.service: Succeeded.
[29337.443504] testvm1.both.org systemd[1]: Finished system activity accounting
tool.
[29394.784506] testvm1.both.org CROND[2253]: (root) CMD (run-parts
/etc/cron.hourly)
[29394.792113] testvm1.both.org run-parts[2256]: (/etc/cron.hourly) starting
0anacron
[29394.799468] testvm1.both.org run-parts[2262]: (/etc/cron.hourly) finished
0anacron
```

Read the `journalctl` man page for a complete list of the timestamp format options (among other things).

Timespans

Timespans are primarily used in systemd timers to define a specific span of time between events. This could be used to trigger events so that they occur a specified amount of time after system startup or after a previous instance of the same event. For example, here is a sample expression in the timer unit file to trigger an event 32 minutes after system startup:

```
OnStartupSec=32m
```

The default accuracy for triggering systemd timers is a time window starting at the specified time and lasting one minute. You can increase trigger timespan accuracy to within a microsecond by adding a statement like the following to the Timer section of the timer-unit file:

```
AccuracySec=1us
```

The `systemd-analyze timespan` command can help ensure you are using a valid timespan in the unit file. These samples will get you started:

```
$ systemd-analyze timespan 15days
Original: 15days
      μs: 1296000000000
   Human: 2w 1d
[student@testvm1 ~]$ systemd-analyze timespan "15days 6h 32m"
Original: 15days 6h 32m
      μs: 1319520000000
   Human: 2w 1d 6h 32min
```

Experiment with these and some of your own:

- `"255days 6h 31m"`
- `"255days 6h 31m 24.568ms"`

Final thoughts

Timespans are used to schedule timer events a specified interval after a defined event such as startup. Calendar timestamps can be used to schedule timer events on specific calendar days and times, either as one-offs or repeating. Timestamps are also used on systemd journal entries, although not in a default format that can be used directly in tools like `systemd-analyze`.

All of this was more than just a little confusing to me when I started working with systemd timers and creating calendar and timestamp expressions to trigger events. That was partly because of the similar—but not quite identical—formats used for specifying timestamps and calendar event trigger times. I hope that this has helped to clarify all of this for you.

Using systemd journals to troubleshoot transient problems

Problem determination can be as much an art as a science, and sometimes, it seems even a little magic can be useful. Everyone has encountered situations where a reported failure could not be reproduced, which is always frustrating for both the user and the system administrator. Even home appliances and automobiles can be obstinate and refuse to fail when the service person shows up.

Anthropomorphism aside, sysadmins have some tools that can show what has been happening in their Linux computers with varying degrees of granularity. There are tools, like top, htop, glances, sar, iotop, tcpdump, traceroute, mtr, iptraf-ng, df, du, and many more, all of which can display a host's current state, and several of which can produce logs of various levels of detail.

While these tools can be used to find ongoing problems, they are not particularly helpful for transient problems or those with no directly observable symptoms—not observable, at least, until some major and possibly catastrophic problem occurs.

An important tool I use for problem determination is the system logs—and with systemd, the system journals. The systemd journal is always one of the first tools I turn to when solving a problem, especially the problems that don't seem to happen when I am watching. It took me a long time at the beginning of my sysadmin career to realize the wealth of information in the log files, and this discovery greatly improved my speed in resolving problems.

In this chapter, I explore some details about the systemd journal, how it works, and ways to use `journalctl` to locate and identify problems.

About journals

The purpose of any log or journal is to maintain a time-sequenced history of the normal activities of the services and programs that run on a host and to record any errors or warning messages that occur. The log messages used to be maintained in separate files in `/var/log`, usually one file for the kernel and separate ones for most of the services running on the host. Unfortunately, the large number of log files could spread out necessary information and delay the discovery of a problem's root cause. This could be especially time-consuming when you're trying to determine what was happening in a system when an error occurred.

The old `/var/log/dmesg` file was usually used for the kernel, but that file was discontinued several years ago in favor of using the `dmesg` command to display the same information and integrating those messages (and more) into the `/var/log/messages` file. This merger of other logs into the `messages` file helped speed problem determination by keeping much of the data in one file, but there were still many services whose logs were not integrated into the more central `messages` file.

The systemd journal is designed to collect all messages into a single, unified structure that can show a complete picture of everything that happened in a system at and around a specific time or event. Because the events, regardless of the source, are in one place and in time sequence, it is possible to see at a glance everything happening at a specific point or range of times. In my opinion, this is one of the main benefits of systemd journaling.

About the systemd journal

The systemd journaling service is implemented by the `systemd-journald` daemon. According to the [`systemd-journald` man page]:

> systemd-journald is a system service that collects and stores logging data. It creates and maintains structured, indexed journals based on logging information that is received from a variety of sources:
>
> - Kernel log messages, via kmsg
> - Simple system log messages, via the libc syslog(3) call
> - Structured system log messages via the native Journal API, see sd_journal_print(3)
> - Standard output and standard error of service units. For further details see below.
> - Audit records, originating from the kernel audit subsystem

> The daemon will implicitly collect numerous metadata fields for each log messages in a secure and unfakeable way. See systemd.journal-fields(7) for more information about the collected metadata.
>
> Log data collected by the journal is primarily text-based but can also include binary data where necessary. Individual fields making up a log record stored in the journal may be up to 2^64-1 bytes in size.

Configuration changes

The systemd journal daemon can be configured using the `/etc/systemd/journald.conf` file. For many hosts, this file does not need any changes because the defaults are quite reasonable. Look at your `journald.conf` file now, if you have not already.

The most common configuration changes you might consider would specify the maximum journal file size, the number of older journal files, and the maximum file-retention times. The primary reason to make those changes would be to reduce the storage space used by the journal if you have a small storage device. In a mission-critical environment, you may also want to reduce the amount of time between syncing journal data stored in RAM memory to the storage device.

The `journald.conf` man page has more details.

Controversies about the data format

One of the controversies surrounding systemd is the binary format in which the journal contents are stored. Some arguments against systemd are based on the systemd journal being stored in a binary format. This would seem to be contrary to the Unix/Linux philosophy to use ASCII text for data, which is one argument people use to justify their dislike of systemd. For example, Doug McIlroy, the inventor of the pipes, said:

> "This is the Unix Philosophy: Write programs that do one thing well. Write programs to work together. Write programs to handle text steams, because that is a universal interface." —Doug McIlroy, quoted in Eric S. Raymond's book *The Art of Unix Programming*

However, these arguments seem to be based on at least a partial misconception because the man page clearly states that the data "is primarily text-based," although it allows for binary data forms. Linux kernel creator Linus Torvalds, who is always clear about his feelings, said:

> "I don't actually have any particularly strong opinions on systemd itself. I've had issues with some of the core developers that I think are much too cavalier about bugs and compatibility, and I think some of the design details are insane (I dislike the binary logs, for example), but those are details, not big issues." —Linus Torvalds, quoted in ZDNet's "[Linus Torvalds and others on Linux's systemd]()" in 2014

The systemd journal files are stored in one or more subdirectories of `/var/log/journal`. Log into a test system where you have root access, and make `/var/log/journal` the present working directory (PWD). List the subdirectories there, choose one, and make it the PWD. You can look at these files in a number of ways. I started with the `stat` command (note that the journal file names on your host will be different from mine):

```
# stat system@7ed846aadf1743139083681ec4042037-0000000000000001-
0005a99c0280fd5f.journal
  File: system@7ed846aadf1743139083681ec4042037-0000000000000001-
0005a99c0280fd5f.journal
  Size: 8388608         Blocks: 16392      IO Block: 4096   regular file
Device: fd04h/64772d    Inode: 524384      Links: 1
Access: (0640/-rw-r-----)  Uid: (    0/    root)   Gid: (  190/systemd-journal)
Access: 2020-07-13 08:30:05.764291231 -0400
Modify: 2020-07-04 07:33:52.916001110 -0400
Change: 2020-07-04 07:33:52.916001110 -0400
 Birth: -
```

The journal file is identified as a "regular" file, which is not especially helpful. The `file` command identifies it as a "journal" file, but you already know that. Look inside the file with the `dd` command. The following command sends the output data stream to STDOUT; you may want to pipe it through the `less` pager:

```
# dd if=system@7ed846aadf1743139083681ec4042037-0000000000000001-
0005a99c0280fd5f.journal | less
[...]
9�P1�8��9_SOURCE_MONOTONIC_TIMESTAMP=191726���/��P����9MESSAGE=Inode-cache hash
table entries: 1048576 (order: 11, 8388608 bytes, linear)�hx
9�P1�p�9��/
��P�b������9��9_SOURCE_MONOTONIC_TIMESTAMP=191825�pdXY�7p�9��9MESSAGE=mem auto-
init: stack:off, heap alloc:off, heap free:off�i��
��(n�O���@Y�    �����Zu���82����7X�8�������8��ⅅR������8<~B�4�<�
�8tM⌁8$����8��Ж��h9�&������9������`@��9�pdXY�7b���WV��9��9_SOURCE_MONOTONIC_TIM
ESTAMP=234745����4�h@�9��9MESSAGE=Memory: 12598028K/12963384K available (14339K
kernel code, 2406K rwdata, 8164K rodata, 2468K init, 5072K b
ss, 365356K reserved, 0K cma-reserved)�j����(n�O���@Y�
����]��m�82����7X�8�������8��ⅅR������8<~B�4�<�
�8tM⌁8$����8��Ж��h9�&�����9���WV��9���
```

```
4�hbB���a��^��9���9_SOURCE_MONOTONIC_TIMESTAMP=234758��3�����9���9MESSAGE=random:
get_random_u64 called from __kmem_cache_create+0x3e/0x610 wi
th crng_init=0�k���(n�O����@Y�
����j������82����7X��8C�X�Y"��8��������8��ⅅR������8<~B�4�<�
�8tM⸙8$����8��Ж�à�9B���a���9�3���b�8�õ������9h�9_SO�9h�9MESSAGE=SLUB: HWalign=64,
Order=0-3, MinObjects=0, CPUs=4, Nodes=1�l����(n�O����@Y�
������z��X�82����7X�8�������8��ⅅR������8<~B�4�<� �8tM⸙
b�(+I)�x�9�9_SOURCE_MONOTONIC_TIMESTAMP=235444r�℃%/p��9�9MESSAGE=Kernel/User page
tables isolation: enabled�m����(n�O����@Y�     ����k��B0��8
2����7X�8�������8��ⅅR������8<~B�4�<�
�8tM⸙8$����8��Ж��h9�&�����8�9�(+I)Ϗ�9�℃%/pb8O{ W����8�9��9_SOURCE_MONOTONIC_TIMESTA
MP=235464u�N`�FP    M��9
��9MESSAGE=ftrace: allocating 41361 entries in 162 pages�n����(n�O����@Y�
[...]
```

Even this small portion of the data stream from the `dd` command shows an interesting mixture of ASCII text and binary data. Another useful tool is the `strings` command, which simply displays all the ASCII text strings contained in a file and ignores the binary data:

```
# strings system@7ed846aadf1743139083681ec4042037-0000000000000001-
0005a99c0280fd5f.journal
[...]
MESSAGE=Linux version 5.7.6-201.fc32.x86_64
(mockbuild@bkernel01.iad2.fedoraproject.org) (gcc version 10.1.1 20200507 (Red
Hat 10.1.1-1) (GC
C), GNU ld version 2.34-3.fc32) #1 SMP Mon Jun 29 15:15:52 UTC 2020
MESSAGE
_BOOT_ID=6e944f99ebd9405984090f829a927fa4
_BOOT_ID
_MACHINE_ID=3bccd1140fca488187f8a1439c832f07
_MACHINE_ID
_HOSTNAME=testvm1.both.org
_HOSTNAME
PRIORITY=6
MESSAGE=Command line: BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.7.6-201.fc32.x86_64
root=/dev/mapper/VG01-root ro resume=/dev/mapper/VG01-swap rd.lv
m.lv=VG01/root rd.lvm.lv=VG01/swap rd.lvm.lv=VG01/usr selinux=0
MESSAGE=x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
MESSAGE=x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
MESSAGE=x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
Z_g3;
MESSAGE=x86/fpu: Enabled xstate features 0x7, context size is 832 bytes, using
'standard' format.
[...]
```

This data can be interpreted by humans, and this particular segment looks very similar to the output data stream from the `dmesg` command. I'll leave you to explore further on your own, but my conclusion is that the journal files are clearly a mixture of binary and ASCII text. That mix makes it cumbersome to use traditional text-based Linux tools to extract usable data. But there is a better way that provides many possibilities for extracting and viewing journal data.

About journalctl

The `journalctl` command is designed to extract usable information from the systemd journals using powerful and flexible criteria for identifying the desired data. Previous articles in this series have described `journalctl`, and there is more to know.

I'll review a bit first and start with some basics in case you have not read the previous articles or just need a refresher.

You can use the `journalctl` command without any options or arguments to view the systemd journal that contains all journal and log information:

```
# journalctl
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Thu 2020-07-16 10:30:43 EDT.
--
Jun 08 07:47:20 testvm1.both.org kernel: Linux version 5.6.6-300.fc32.x86_64
(mockbuild@bkernel03.phx2.fedoraproject.org) (gcc version 10.0>
Jun 08 07:47:20 testvm1.both.org kernel: Command line:
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.6.6-300.fc32.x86_64 root=/dev/mapper/VG01-root
ro >
Jun 08 07:47:20 testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x001:
'x87 floating point registers'
Jun 08 07:47:20 testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x002:
'SSE registers'
[...]
Jul 16 10:00:42 testvm1.both.org systemd[1]: sysstat-collect.service: Succeeded.
Jul 16 10:00:42 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=sysstat-collect comm="systemd>
Jul 16 10:00:42 testvm1.both.org systemd[1]: Finished system activity accounting
tool.
Jul 16 10:00:42 testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=sysstat-collect comm="systemd">
Jul 16 10:01:01 testvm1.both.org CROND[378562]: (root) CMD (run-parts
/etc/cron.hourly)
Jul 16 10:01:01 testvm1.both.org run-parts[378565]: (/etc/cron.hourly) starting
0anacron
[...]
```

You can also explicitly show the same data the `dmesg` command presents. Open two terminal sessions next to each other and issue the `dmesg` command in one and the following command in the other:

```
# journalctl --dmesg
```

The only difference you should see is the time format. The `dmesg` command is in a monotonic format that shows the number of seconds since the system boot. The `journalctl` output is in a date and time format. The short-monotonic option displays the time since boot:

```
# journalctl --dmesg -o short-monotonic
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Mon 2020-07-20 13:01:01 EDT.
--
[    0.000000] testvm1.both.org kernel: Linux version 5.7.6-201.fc32.x86_64
(mockbuild@bkernel01.iad2.fedoraproject.org) (gcc version 10.1.>
[    0.000000] testvm1.both.org kernel: Command line:
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.7.6-201.fc32.x86_64 root=/dev/mapper/VG01-root
ro r>
[    0.000000] testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x001:
'x87 floating point registers'
[    0.000000] testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x002:
'SSE registers'
[    0.000000] testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x004:
'AVX registers'
[    0.000000] testvm1.both.org kernel: x86/fpu: xstate_offset[2]:  576,
xstate_sizes[2]:  256
[    0.000000] testvm1.both.org kernel: x86/fpu: Enabled xstate features 0x7,
context size is 832 bytes, using 'standard' format.
<snip>
[    0.000002] testvm1.both.org kernel: clocksource: kvm-clock: mask:
0xffffffffffffffff max_cycles: 0x1cd42e4dffb, max_idle_ns: 8815905914>
[    0.000005] testvm1.both.org kernel: tsc: Detected 2807.988 MHz processor
[    0.001157] testvm1.both.org kernel: e820: update [mem 0x00000000-0x00000fff]
usable ==> reserved
[...]
lines 624-681/681 (END)
```

The `journalctl` command has many options, including the `-o` (output) option with several suboptions that allow you to select a time and date format that meets different sets of requirements. I have listed most of them below, along with a short description that I expanded or modified from the `journalctl` man page. Note that the primary difference between most of these is the format of the date and time, and the other information remains the same.

journalctl time and date formats

- **short:** This is the default format and generates an output that is most closely like the formatting of classic syslog files, showing one line per journal entry. This option shows journal metadata including the monotonic time since boot, the fully qualified hostname, and the unit name such as the kernel, DHCP, etc.

```
Jul 20 08:43:01 testvm1.both.org kernel: Inode-cache hash table entries:
1048576 (order: 11, 8388608 bytes, linear)
```

- **short-full:** This format is very similar to the default but shows timestamps in the format the `--since=` and `--until=` options accept. Unlike the timestamp information shown in short output mode, this mode includes weekday, year, and timezone information in the output and is locale-independent.

```
Mon 2020-06-08 07:47:20 EDT testvm1.both.org kernel: x86/fpu: Supporting
XSAVE feature 0x004: 'AVX registers'
```

- **short-iso:** The short-iso format is very similar to the default, but shows ISO 8601 wallclock timestamps.

```
2020-06-08T07:47:20-0400 testvm1.both.org kernel: kvm-clock: Using msrs
4b564d01 and 4b564d00
```

- **short-iso-precise:** This format is the same as short-iso but includes full microsecond precision.

```
2020-06-08T07:47:20.223738-0400 testvm1.both.org kernel: Booting
paravirtualized kernel on KVM
```

- **short-monotonic:** Very similar to the default but shows monotonic timestamps instead of wallclock timestamps.

```
[    2.091107] testvm1.both.org kernel: ata1.00: ATA-6: VBOX HARDDISK, 1.0,
max UDMA/133
```

- **short-precise:** This format is also similar to the default but shows classic syslog timestamps with full microsecond precision.

```
Jun 08 07:47:20.223052 testvm1.both.org kernel: BIOS-e820: [mem
0x000000000009fc00-0x000000000009ffff] reserved
```

- **short-unix:** Like the default, but shows seconds passed since January 1, 1970, UTC instead of wallclock timestamps ("Unix time"). The time is shown with microsecond accuracy.

```
1591616840.232165 testvm1.both.org kernel: tcp_listen_portaddr_hash hash
table entries: 8192
```

- **cat:** Generates a very terse output only showing the message of each journal entry with no metadata, not even a timestamp.

```
ohci-pci 0000:00:06.0: irq 22, io mem 0xf0804000
```

- **verbose:** This format shows the full data structure for all the entry items with all fields. This is the format option that is most different from all the others.

```
Mon 2020-06-08 07:47:20.222969 EDT
[s=d52ddc9f3e8f434b9b9411be2ea50b1e;i=1;b=dcb6dcc0658e4a8d8c781c21a2c6360d;
m=242d7f;t=5a7912c6148f9;x=8f>
    _SOURCE_MONOTONIC_TIMESTAMP=0
    _TRANSPORT=kernel
    PRIORITY=5
    SYSLOG_FACILITY=0
    SYSLOG_IDENTIFIER=kernel
    MESSAGE=Linux version 5.6.6-300.fc32.x86_64
(mockbuild@bkernel03.phx2.fedoraproject.org) (gcc version 10.0.1 20200328
(Red Hat 10.0.1-0>
    _BOOT_ID=dcb6dcc0658e4a8d8c781c21a2c6360d
    _MACHINE_ID=3bccd1140fca488187f8a1439c832f07
    _HOSTNAME=testvm1.both.org
```

Other choices, available with the `-o` option, allow exporting the data in various formats such as binary or JSON. I also find the `-x` option illuminating because it can show additional explanatory messages for some journal entries. If you try this option, be aware that it can greatly increase the output data stream. For example, look at the additional information for an entry like this:

```
[    4.503737] testvm1.both.org systemd[1]: Starting File System Check on
/dev/mapper/VG01-root...
[    4.691555] testvm1.both.org systemd-fsck[548]: root: clean, 1813/327680
files, 48555/1310720 blocks
[    4.933065] testvm1.both.org systemd[1]: Finished File System Check on
/dev/mapper/VG01-root.
```

That expands to:

```
[    4.503737] testvm1.both.org systemd[1]: Starting File System Check on
/dev/mapper/VG01-root...
-- Subject: A start job for unit systemd-fsck-root.service has begun execution
-- Defined-By: systemd
-- Support: https://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- A start job for unit systemd-fsck-root.service has begun execution.
--
-- The job identifier is 36.
[    4.691555] testvm1.both.org systemd-fsck[548]: root: clean, 1813/327680
files, 48555/1310720 blocks
[    4.933065] testvm1.both.org systemd[1]: Finished File System Check on
/dev/mapper/VG01-root.
-- Subject: A start job for unit systemd-fsck-root.service has finished
successfully
-- Defined-By: systemd
-- Support: https://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- A start job for unit systemd-fsck-root.service has finished successfully.
--
-- The job identifier is 36
```

There is some new information here, but I think the main benefit is that the information is contextualized to clarify the original terse messages to some degree.

Narrowing the search

Most of the time, it is not necessary or even desirable to list all the journal entries and manually search through them. Sometimes I look for entries related to a specific service, and other times I look for entries that happened at specific times. The `journalctl` command provides powerful options that allow you to see only the data you are interested in finding.

Start with the `--list-boots` option, which lists all the boots during the time period when journal entries exist. Note that the `journalctl.conf` file may specify that journal entries are discarded after they reach a certain age or after the storage device (HDD/SSD) space taken by the journals reaches a specified maximum amount:

```
# journalctl --list-boots
-10 dcb6...360d Mon 2020-06-08 07:47:20 EDT—Mon 2020-06-08 07:53:05 EDT
 -9 7d61...c2a4 Fri 2020-07-03 15:50:06 EDT—Fri 2020-07-03 18:21:23 EDT
 -8 1b3a...2206 Fri 2020-07-03 18:21:58 EDT—Fri 2020-07-03 22:10:54 EDT
 -7 5fef...61ae Fri 2020-07-03 22:18:41 EDT—Sat 2020-07-04 06:50:19 EDT
```

```
 -6 6e94...7fa4 Sat 2020-07-04 07:33:25 EDT—Sat 2020-07-04 07:58:59 EDT
 -5 ec8b...32959 Sat 2020-07-04 08:12:06 EDT—Sat 2020-07-04 09:12:47 EDT
 -4 cb17...9a99 Sat 2020-07-04 10:19:53 EDT—Sat 2020-07-04 11:31:03 EDT
 -3 4fe3...dbb0 Sat 2020-07-04 07:59:58 EDT—Sun 2020-07-05 09:39:30 EDT
 -2 06fb...446c Mon 2020-07-06 06:27:05 EDT—Mon 2020-07-13 08:50:06 EDT
 -1 33db...ac37 Mon 2020-07-13 04:50:33 EDT—Thu 2020-07-16 13:49:32 EDT
  0 75c9...ee50 Mon 2020-07-20 08:43:01 EDT—Fri 2020-07-24 12:44:06 EDT
```

The most recent boot ID appears at the bottom; it is the long, random hex number. You can use this data to view the journals for a specific boot. This can be specified using the boot offset number in the left-most column or the UUID in the second column. This command displays the journal for the boot instance with the offset of `-2`—the second previous boot from the current one:

```
# journalctl -b -2
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Fri 2020-07-24 12:44:06 EDT.
--
Jul 06 06:27:05 testvm1.both.org kernel: Linux version 5.7.6-201.fc32.x86_64
(mockbuild@bkernel01.iad2.fedoraproject.org) (gcc version 10.1>
Jul 06 06:27:05 testvm1.both.org kernel: Command line:
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.7.6-201.fc32.x86_64 root=/dev/mapper/VG01-root
ro >
Jul 06 06:27:05 testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x001:
'x87 floating point registers'
Jul 06 06:27:05 testvm1.both.org kernel: x86/fpu: Supporting XSAVE feature 0x002:
'SSE registers'
<SNIP>
```

Or you could use the UUID for the desired boot. The offset numbers change after each boot, but the UUID does not:

```
# journalctl -b 06fb81f1b29e4f68af9860844668446c
```

The `-u` option allows you to select specific units to examine. You can use a unit name or a pattern for matching, and you can use this option multiple times to match multiple units or patterns. In this example, I used it in combination with `-b` to show chronyd journal entries for the current boot:

```
# journalctl -u chronyd -b
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Sun 2020-07-26 09:10:47 EDT.
--
Jul 20 12:43:31 testvm1.both.org systemd[1]: Starting NTP client/server...
```

```
Jul 20 12:43:31 testvm1.both.org chronyd[811]: chronyd version 3.5 starting
(+CMDMON +NTP +REFCLOCK +RTC +PRIVDROP +SCFILTER +SIGND +ASYNCD>
Jul 20 12:43:31 testvm1.both.org chronyd[811]: Frequency -0.021 +/- 0.560 ppm
read from /var/lib/chrony/drift
Jul 20 12:... chronyd[811]: Using right/UTC timezone to obtain leap second data
Jul 20 12:43:31 testvm1.both.org systemd[1]: Started NTP client/server.
Jul 20 12:44:00 testvm1.both.org chronyd[811]: Selected source 192.168.0.52
Jul 20 12:44:00 testvm1.both.org chronyd[811]: System clock TAI offset set to 37s
Jul 20 12:44:00 testvm1.both.org chronyd[811]: System clock wrong by 1.412227
seconds, adjustment started
Jul 20 12:44:01 testvm1.both.org chronyd[811]: System clock was stepped by
1.412227 seconds
```

Suppose you want to look at events that were recorded between two arbitrary times. You can also use `-S` (`--since`) and `-U` (`--until`) to specify the beginning and ending times. The following command displays journal entries starting at 15:36:00 on July 24, 2020, through the current time:

```
# journalctl -S "2020-07-24 15:36:00"
```

And this command displays all journal entries starting at 15:36:00 on July 24, 2020, until 16:00:00 on July 25:

```
# journalctl -S "2020-07-24 15:36:00" -U "2020-07-25 16:00:00"
```

This command combines `-S`, `-U`, and `-u` to give journal entries for the NetworkManager service unit starting at 15:36:00 on July 24, 2020, until 16:00:00 on July 25:

```
# journalctl -S "2020-07-24 15:36:00" -U "2020-07-25 16:00:00" -u NetworkManager
```

Some syslog facilities, such as cron, auth, mail, daemon, user, and more, can be viewed with the `--facility` option. You can use `--facility=help` to list the available facilities. In this example, the mail facility is not the Sendmail service that would be used for an email service, but the local client used by Linux to send email to root as event notifications. Sendmail actually has two parts, the server, which (for Fedora and related distributions) is not installed by default, and the client, which is always installed so that it can be used to deliver system emails to local recipients, especially root:

```
# journalctl --facility=mail
```

The `journalctl` man page lists all the options that can be used to narrow searches. The table below summarizes some of the options I use most frequently. Most of these options can be used in various combinations to further narrow a search.

Options to narrow searches of the journal

Option	Description
--list-boots	This displays a list of boots. The information can be used to show journal entries only for a particular boot.
-b [offset|boot ID]	This specifies which boot to display information for. It includes all journal entries from that boot through shutdown or reboot.
--facility=[facility name]	This specifies the facility names as they're known to syslog. Use --facility=help to list the valid facility names.
-k, --dmesg	These display only kernel messages and are equivalent to using the dmesg command.
-S, --since [timestamp]	These show all journal entries since (after) the specified time. They can be used with --until to display an arbitrary range of time. Fuzzy times such as "yesterday" and "2 hours ago"—with quotes—are also allowed.
-u [unit name]	The -u option allows you to select specific units to examine. You can use a unit name or a pattern for matching. This option can be used multiple times to match multiple units or patterns.
-U, --until [timestamp]	These show all journal entries until (prior to) the specified time. They can be used with --since to display an arbitrary range of time. Fuzzy times such as "yesterday" and "2 hours ago"—with quotes—are also allowed.

Other interesting options

The `journalctl` program offers some other interesting options, as well. These are useful for refining the data search, specifying how the journal data is displayed, and managing the journal files.

Option	Description
-f, --follow	This journalctl option is similar to using the tail -f command. It shows the most recent entries in the journal that match whatever other options have been specified and also displays new entries as they occur. This can be useful when watching for events and

	when testing changes.
-e, --pager-end	The -e option displays the end of the data stream instead of the beginning. This does not reverse the order of the data stream, rather it causes the pager to jump to the end.
--file [journal filename]	This names a specific journal file in /var/log/journal/<journal subdirectory>.
-r, --reverse	This option reverses the order of the journal entries in the pager so that the newest are at the top rather than the bottom.
-n, --lines=[X]	This shows the most recent X number of lines from the journal.
--utc	This displays times in UTC rather than local time.
-g, --grep=[REGEX]	I like the -g option because it enables me to search for specific patterns in the journal data stream. This is just like piping a text data stream through the grep command. This option uses Perl-compatible regular expressions.
--disk-usage	This option displays the amount of disk storage used by the current and archived journals. It might not be as much as you think.
--flush	Journal data stored in the virtual filesystem /run/log/journal/, which is volatile storage, is written to /var/log/journal/ which is persistent storage. This option ensures that all data is flushed to /run/log/journal/ at the time it returns.
--sync	This writes all unwritten journal entries (still in RAM but not in /run/log/journal) to the persistent filesystem. All journal entries known to the journaling system at the time the command is entered are moved to persistent storage.
--vacuum-size= --vacuum-time= --vacuum-files=	These can be used singly or in combination to remove the oldest archived journal files until the specified condition is met. These options only consider archived files, and not active files, so the result might not be exactly what was specified.

Journal files

If you have not already, be sure to list the files in the journal directory on your host. Remember that the name of the directory containing the journal files is a long, random number. This directory contains multiple active and archived journal files, including some for users:

```
# cd /var/log/journal/ad8f29ed15044f8ba0458c846300c2a4/
[root@david ad8f29ed15044f8ba0458c846300c2a4]# ll
total 352308
```

```
-rw-r-----+ 1 root systemd-journal  33554432 May 28 13:07
system@0c91aaef57c441859ea5e421edff6528-0000000000000001-0005a6703120d448.journal
-rw-r-----+ 1 root systemd-journal 109051904 Jun 23 21:24
system@0c91aaef57c441859ea5e421edff6528-0000000000008238-0005a6b85e4e03c6.journal
-rw-r-----+ 1 root systemd-journal 100663296 Jul 21 18:39
system@0c91aaef57c441859ea5e421edff6528-0000000000021f3e-0005a8ca55efa66a.journal
-rw-r-----+ 1 root systemd-journal  41943040 Jul 30 09:34 system.journal
-rw-r-----+ 1 root systemd-journal   8388608 May 28 13:07 user-
1000@037bcc7935234a5ea243b3af304fd08a-0000000000000c45-0005a6705ac48a3c.journal
-rw-r-----+ 1 root systemd-journal  16777216 Jun 23 21:24 user-
1000@bc90cea5294447fba2c867dfe40530db-0000000000008266-0005a6b85e910761.journal
-rw-r-----+ 1 root systemd-journal  41943040 Jul 21 16:08 user-
1000@bc90cea5294447fba2c867dfe40530db-0000000000021f4b-0005a8ca68c83ab7.journal
-rw-r-----+ 1 root systemd-journal   8388608 Jul 30 09:34 user-1000.journal
[root@david ad8f29ed15044f8ba0458c846300c2a4]#
```

You can see the user files in this listing for the user ID (UID) 1000, which is my Linux account. The `--files` option allows me to see the content of specified files, including the user files:

```
# journalctl --file user-1000.journal
[...]
Jul 29 14:13:32 david.both.org tumblerd[145509]: Registered thumbnailer
/usr/bin/gdk-pixbuf-thumbnailer -s %s %u %o
Jul 29 14:13:32 david.both.org Thunar[2788]: ThunarThumbnailer: got 0 handle
(Queue)
Jul 29 14:13:32 david.both.org Thunar[2788]: ThunarThumbnailer: got 0 handle
(Error or Ready)
Jul 29 14:13:32 david.both.org Thunar[2788]: ThunarThumbnailer: got 0 handle
(Finished)
Jul 29 14:15:33 david.both.org tumblerd[145552]: error: Broken zip file structure
Jul 29 14:20:34 david.both.org systemd[2466]: dbus-:1.2-
org.freedesktop.thumbnails.Thumbnailer1@11.service: Succeeded.
Jul 29 14:34:17 david.both.org systemd[2466]: Starting Cleanup of User's
Temporary Files and Directories...
Jul 29 14:34:17 david.both.org systemd[2466]: systemd-tmpfiles-clean.service:
Succeeded.
Jul 29 14:34:17 david.both.org systemd[2466]: Finished Cleanup of User's
Temporary Files and Directories.
Jul 29 14:48:26 david.both.org systemd[2466]: Started dbus-:1.2-
org.freedesktop.thumbnails.Thumbnailer1@12.service.
Jul 29 14:48:26 david.both.org tumblerd[145875]: Registered thumbnailer gsf-
office-thumbnailer -i %i -o %o -s %s
[...]
```

This output shows, among other things, temporary file cleanup for the UID1000 user. Data relating to individual users may be helpful in locating the root cause of problems originating in user space. I found a number of interesting entries in this output. Try it on your host and see what you find.

Adding journal entries

It can be useful to add your own entries to the journal. This is accomplished with the `systemd-cat` program that allows piping the STDOUT of a command or program to the journal. This command can be used as part of a pipeline on the command line or in a script:

```
# echo "Hello world" | systemd-cat -p info -t myprog
# journalctl -n 10
Jul 27 09:01:01...both.org CROND[9742]: (root) CMD (run-parts /etc/cron.hourly)
Jul 27 09:01:01...both.org run-parts[...]: (/etc/cron.hourly) starting 0anacron
Jul 27 09:01:01...both.org run-parts[451]: (/etc/cron.hourly) finished 0anacron
Jul 27 09:07:53 testvm1.both.org unknown[976501]: Hello world
Jul 27 09:10:47...both.org systemd[1]: Starting system activity accounting
tool...
Jul 27 09:10:47 testvm1.both.org systemd[1]: sysstat-collect.service: Succeeded.
Jul 27 09:10:47...both.org systemd[1]: Finished system activity accounting tool.
Jul 27 09:10:47 testvm1.both.org audit[1]: SERVICE_START pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=sysstat-collect comm="systemd"
exe="/usr/lib/syst>
Jul 27 09:10:47 testvm1.both.org audit[1]: SERVICE_STOP pid=1 uid=0
auid=4294967295 ses=4294967295 msg='unit=sysstat-collect comm="systemd"
exe="/usr/lib/syste>
Jul 27 09:17:10 testvm1.both.org myprog[976516]: Hello world
```

The `-p` option specifies a priority, emerg, alert, crit, err, warning, notice, info, debug, or a value between 0 and 7 that represents each of those named levels. These priority values are the same as those defined by `syslog(3)`. The default is info. The `-t` option is an identifier, which can be any arbitrary short string, such as a program or script name. This string can be used for searches by the `journalctl` command:

```
# journalctl -t myprog
-- Logs begin at Mon 2020-06-08 07:47:20 EDT, end at Mon 2020-07-27 09:21:57 EDT.
--
Jul 27 09:17:10 testvm1.both.org myprog[976516]: Hello world
```

Journal management

I use the `--disk-usage` option to check on journal sizes, along with other commands relating to disk usage, to ensure that my `/var` filesystem is not filling up:

```
# journalctl --disk-usage
```

```
Archived and active journals take up 136.0M in the file system.
```

The disk usage for the journals on the testvm1 host is about 136MB. The result on my primary workstation is 328MB, and the host I use for my firewall and router uses 2.8GB for the journals. Journal sizes depend greatly on the host's use and daily run time. My physical hosts all run 24x7.

The `/etc/systemd/journald.conf` file can be used to configure the journal file sizes, rotation, and retention times to meet any needs not met by the default settings. You can also configure the journal storage location—you can specify a directory on the storage device or whether to store everything in RAM, which is volatile storage. If the journals are stored in RAM, they will not be persistent between boots.

The default time unit in the `journald.conf` file is seconds, but it can be overridden using the suffixes `year`, `month`, `week`, `day`, `h`, or `m`.

Suppose you want to limit the total amount of storage space allocated to journal files to 1GB, store all journal entries in persistent storage, keep a maximum of 10 files, and delete any journal archive files that are more than a month old. You can configure this in `/etc/systemd/journald.conf` using:

```
SystemMaxUse=1G
Storage=persistent
SystemMaxFiles=10
MaxRetentionSec=1month
```

By default, the `SystemMaxUse` is 10% of available disk space. The default settings have been fine for the systems I work with, and I have not needed to change any of them. The `journald.conf` man page states that the time-based settings for specifying how long to store journal entries in a single file or to retain older files are normally not necessary. This is because file number and size configurations usually force rotation and deletion of old files before any time settings might be needed.

The `SystemKeepFree` option ensures a specific amount of space is kept free for other data. Many databases and application programs use the `/var` filesystem to store data, so ensuring enough available storage space may be critical in systems with smaller hard drives and minimum space allocated to `/var`.

If you make changes to this configuration, be sure to monitor the results carefully for an appropriate period of time to ensure they are performing as expected.

Journal file rotation

The journal files are typically rotated automatically based upon the configuration in the `/etc/systemd/journald.conf` file. Files are rotated whenever one of the specified conditions is met. So if, for example, the space allocated to journal files is exceeded, the oldest file(s) is deleted, the active file is made into an archive, and a new active file is created.

Journal files can also be rotated manually. I suggest using the `--flush` option to ensure current data is moved to persistent storage before you run the command:

```
# journalctl --rotate
```

There is another way to purge old journal files without performing a file rotation. The `vacuum-size=`, `vacuum-files=`, and `vacuum-time=` commands can be used to delete old archive files down to a specified total size, number of files, or prior time. The option values consider only the archive files and not the active ones, so the resulting reduction in total file size might be somewhat less than expected.

The following command purges old archive files so that only ones that are less than one month old are left. You can use the `s`, `m`, `h`, `days`, `months`, `weeks`, and `years` suffixes:

```
# journalctl --vacuum-time=1month
```

This command deletes all archive files except for the four most recent ones. If there are fewer than four archive files, nothing will happen, and the original number of files remains:

```
# journalctl --vacuum-files=4
```

This last `vacuum` command deletes archive files until 200MB or less of archived files are left:

```
# journalctl --vacuum-size=200M
```

Only complete files are deleted. The vacuum commands do not truncate archive files to meet the specification. They also work only on archive files, not active ones.

Final thoughts

This article looked at using the `journalctl` command to extract various types of data from the systemd journal in different formats. It also explored managing journal files and how to add entries to the log from commands and scripts.

The systemd journal system provides a significant amount of metadata and context for entries compared to the old syslogd program. This additional data and the context available from the other journal entries around the time of an incident can help the sysadmin locate and resolve problems much faster than having to search multiple syslog files.

In my opinion, the `journalctl` command meets the Unix philosophy that programs should do one thing and do it well. The only thing `journalctl` does is extract data from the journal and provide many options for selecting and formatting that data. At about 85K, it is not very big. Of course, that does not include shared libraries, but those are, by definition, shared with other programs.

You should now have enough information to use the systemd journal more effectively. If you would like to know more than what I covered here, look in the man pages for `journalctl` and `systemd-cat`.

Resolve systemd-resolved name-service failures with Ansible

Most people tend to take name services for granted. They are necessary to convert human-readable names, such as `www.example.com`, into IP addresses, like `93.184.216.34`. It is easier for humans to recognize and remember names than IP addresses, and name services allow us to use names, and they also convert them to IP addresses for us.

The Domain Name System (DNS) is the global distributed database that maintains the data required to perform these lookups and reverse lookups, in which the IP address is known and the domain name is needed.

I installed Fedora 33 the first day it became available in October 2020. One of the major changes was a migration from the ancient Name Service Switch (NSS) resolver to systemd-resolved. Unfortunately, after everything was up and running, I couldn't connect to or even ping any of the hosts on my network by name, although using IP addresses did work.

The problem

I run my own name server using BIND on my network server, and all has been good for over 20 years. I've configured my DHCP server to provide the IP address of my name server to every workstation connected to my network, and that (along with a couple of backup name servers) is stored in `/etc/resolv.conf`.

Michael Catanzaro describes how systemd-resolved is supposed to work, but the introduction of systemd-resolved caused various strange resolution problems on my network hosts. The symptoms varied depending upon the host's purpose. The trouble mostly presented as an inability to find IP addresses for hosts inside the network on most systems. On other systems, it failed to resolve any names at all. For example, even though nslookup sometimes returned the correct IP addresses for hosts inside and outside networks, ping would not contact the designated host, nor could I SSH to that same host. Most of the time, neither the lookup, the ping, nor SSH would work unless I used the IP address in the command.

The new resolver allegedly has four operational modes, described in this [Fedora Magazine article](). None of the options seems to work correctly when the network has its own name server designed to perform lookups for internal and external hosts.

In theory, systemd-resolved is supposed to fix some corner issues around the NSS resolver failing to use the correct name server when a host is connected to a VPN, which has become a common problem with so many more people working from home.

The new resolver is supposed to use the fact that `/etc/resolv.conf` is now a symlink to determine how it is supposed to work based on which resolve file it is linked to. systemd-resolved's man page includes details about this behavior. The man page says that setting `/etc/resolv.conf` as a symlink to `/run/systemd/resolve/resolv.conf` should cause the new resolver to work the same way the old one does, but that didn't work for me.

My fix

I have seen many options on the internet for resolving this problem, but the only thing that works reliably for me is to stop and disable the new resolver. First, I deleted the existing link for `resolv.conf`, copied my preferred `resolv.conf` file to `/run/NetworkManager/resolv.conf`, and added a new link to that file in `/etc`:

```
# rm -f /etc/resolv.conf
# ln -s /run/NetworkManager/resolv.conf /etc/resolv.conf
```

These commands stop and disable the systemd-resolved service:

```
# systemctl stop systemd-resolved.service ; systemctl disable systemd-
resolved.service
Removed /etc/systemd/system/multi-user.target.wants/systemd-resolved.service.
Removed /etc/systemd/system/dbus-org.freedesktop.resolve1.service.
```

There's no reboot required. The old resolver takes over, and name services work as expected.

Make it easy with Ansible

I set up this Ansible playbook to make the necessary changes if I install a new host or an update that reverts the resolver to systemd-resolved, or if an upgrade to the next release of

Fedora reverts the resolver. The `resolv.conf` file you want for your network should be located in `/root/ansible/resolver/files/`:

```
# fixResolver.yml                                 #
# This playbook configures the old nss resolver on systems that have the new   #
# systemd-resolved service installed. It copies the resolv.conf file for my    #
# network to /run/NetworkManager/resolv.conf and places a link to that file    #
# as /etc/resolv.conf. It then stops and disables systemd-resolved which       #
# activates the old nss resolver.                                              #
################################################################################
---
- name: Revert to old NSS resolver and disable the systemd-resolved service
  hosts: all_by_IP
################################################################################

  tasks:
    - name: Copy resolv.conf for my network
      copy:
        src: /root/ansible/resolver/files/resolv.conf
        dest: /run/NetworkManager/resolv.conf
        mode: 0644
        owner: root
        group: root

    - name: Delete existing /etc/resolv.conf file or link
      file:
        path: /etc/resolv.conf
        state: absent

    - name: Create link from /etc/resolv.conf to /run/NetworkManager/resolv.conf
      file:
        src: /run/NetworkManager/resolv.conf
        dest: /etc/resolv.conf
        state: link

    - name: Stop and disable systemd-resolved
      systemd:
        name: systemd-resolved
        state: stopped
        enabled: no
```

Whichever Ansible inventory you use must have a group that uses IP addresses instead of hostnames. This command runs the playbook and specifies the name of the inventory file I use for hosts by IP address:

```
# ansible-playbook -i /etc/ansible/hosts_by_IP fixResolver.yml
```

Final thoughts

Sometimes the best answer to a tech problem is to fall back to what you know. When systemd-resolved is more robust, I'll likely give it another try, but for now I'm glad that open source infrastructure allows me to quickly identify and resolve network problems. Using Ansible to automate the process is a much appreciated bonus. The important lesson here is to do your research when troubleshooting, and to never be afraid to void your warranty.

Analyze Linux startup performance

Part of the system administrator's job is to analyze the performance of systems and to find and resolve problems that cause poor performance and long startup times. Sysadmins also need to check other aspects of systemd configuration and usage.

The systemd init system provides the `systemd-analyze` tool that can help uncover performance problems and other important systemd information. In a previous chapter, I used `systemd-analyze` to analyze timestamps and timespans in systemd timers, but this tool has many other uses, some of which I explore in this article.

Startup overview

The Linux startup sequence is a good place to begin exploring because many `systemd-analyze` tool functions are targeted at startup. But first, it is important to understand the difference between boot and startup. The boot sequence starts with the BIOS power-on self test (POST) and ends when the kernel is finished loading and takes control of the host system, which is the beginning of startup and the point when the systemd journal begins.

Earlier in this book, I discussed startup in more detail with respect to what happens and in what sequence. In this article, I want to examine the startup sequence to look at the amount of time it takes to go through startup and which tasks take the most time.

The results I'll show below are from my primary workstation, which is much more interesting than a virtual machine's results. This workstation consists of an ASUS TUF X299 Mark 2 motherboard, an Intel i9-7960X CPU with 16 cores and 32 CPUs (threads), and 64GB of RAM. Some of the commands below can be run by a non-root user, but I will use root in this article to prevent having to switch between users.

Creative Commons Attribution Share-alike 4.0

There are several options for examining the startup sequence. The simplest form of the `systemd-analyze` command displays an overview of the amount of time spent in each of the main sections of startup, the kernel startup, loading and running `initrd` (i.e., initial ramdisk, a temporary system image that is used to initialize some hardware and mount the `/` [root] filesystem), and userspace (where all the programs and daemons required to bring the host up to a usable state are loaded). If no subcommand is passed to the command, `systemd-analyze time` is implied:

```
# systemd-analyze
Startup finished in 53.921s (firmware) + 2.643s (loader) + 2.236s (kernel) +
4.348s (initrd) + 10.082s (userspace) = 1min 13.233s
graphical.target reached after 10.071s in userspace
```

The most notable data in this output is the amount of time spent in firmware (BIOS): almost 54 seconds. This is an extraordinary amount of time, and none of my other physical systems take anywhere near as long to get through BIOS.

My System76 Oryx Pro laptop spends only 8.506 seconds in BIOS, and all of my home-built systems take a bit less than 10 seconds. After some online searches, I found that this motherboard is known for its inordinately long BIOS boot time. My motherboard never "just boots." It always hangs, and I need to do a power off/on cycle, and then BIOS starts with an error, and I need to press F1 to enter BIOS configuration, from where I can select the boot drive and finish the boot. This is where the extra time comes from.

Not all hosts show firmware data. My unscientific experiments lead me to believe that this data is shown only for Intel generation 9 processors or above. But that could be incorrect.

This overview of the boot startup process is interesting and provides good (though limited) information, but there is much more information available about startup, as I'll describe below.

Assigning blame

You can use `systemd-analyze blame` to discover which systemd units take the most time to initialize. The results are displayed in order by the amount of time they take to initialize, from most to least:

```
# systemd-analyze blame
       5.417s NetworkManager-wait-online.service
       3.423s dracut-initqueue.service
       2.715s systemd-udev-settle.service
       2.519s fstrim.service
       1.275s udisks2.service
```

```
       1.271s smartd.service
        996ms upower.service
        637ms lvm2-monitor.service
        533ms lvm2-pvscan@8:17.service
        520ms dmraid-activation.service
        460ms vboxdrv.service
        396ms initrd-switch-root.service
[...] # I removed lots of entries with increasingly small times
```

Because many of these services start in parallel, the numbers may add up to significantly more than the total given by `systemd-analyze time` for everything after the BIOS. All of these are small numbers, so I cannot find any significant savings here.

The data from this command can provide indications about which services you might consider to improve boot times. Services that are not used can be disabled. There does not appear to be any single service that is taking an excessively long time during this startup sequence. You may see different results for each boot and startup.

Critical chains

Like the critical path in project management, a *critical chain* shows the time-critical chain of events that take place during startup. These are the systemd units you want to look at if startup is slow, as they are the ones that would cause delays. This tool does not display all the units that start, only those in this critical chain of events:

```
# systemd-analyze critical-chain
The time when unit became active or started is printed after the "@" character.
The time the unit took to start is printed after the "+" character.

graphical.target @10.071s
└─lxdm.service @10.071s
  └─plymouth-quit.service @10.047s +22ms
    └─systemd-user-sessions.service @10.031s +7ms
      └─remote-fs.target @10.026s
        └─remote-fs-pre.target @10.025s
          └─nfs-client.target @4.636s
            └─gssproxy.service @4.607s +28ms
              └─network.target @4.604s
                └─NetworkManager.service @4.383s +219ms
                  └─dbus-broker.service @4.434s +136ms
                    └─dbus.socket @4.369s
                      └─sysinit.target @4.354s
                        └─systemd-update-utmp.service @4.345s +9ms
                          └─auditd.service @4.301s +42ms
                            └─systemd-tmpfiles-setup.service @4.254s +42ms
```

```
                              └─import-state.service @4.233s +19ms
                                └─local-fs.target @4.229s
                                  └─Virtual.mount @4.019s +209ms
                                    └─systemd-fsck@dev-mapper-vg_david2\
x2dVirtual.service @3.742s +274ms
                                      └─local-fs-pre.target @3.726s
                                        └─lvm2-monitor.service @356ms +637ms
                                          └─dm-event.socket @319ms
                                            └─-.mount
                                              └─system.slice
                                                └─-.slice
```

The numbers preceded with @ show the absolute number of seconds since startup began when the unit becomes active. The numbers preceded by + show the amount of time it takes for the unit to start.

System state

Sometimes you need to determine the system's current state. The `systemd-analyze dump` command dumps a *massive* amount of data about the current system state. It starts with a list of the primary boot timestamps, a list of each systemd unit, and a complete description of the state of each:

```
# systemd-analyze dump
Timestamp firmware: 1min 7.983523s
Timestamp loader: 3.872325s
Timestamp kernel: Wed 2020-08-26 12:33:35 EDT
Timestamp initrd: Wed 2020-08-26 12:33:38 EDT
Timestamp userspace: Wed 2020-08-26 12:33:42 EDT
Timestamp finish: Wed 2020-08-26 16:33:56 EDT
Timestamp security-start: Wed 2020-08-26 12:33:42 EDT
Timestamp security-finish: Wed 2020-08-26 12:33:42 EDT
Timestamp generators-start: Wed 2020-08-26 16:33:42 EDT
Timestamp generators-finish: Wed 2020-08-26 16:33:43 EDT
Timestamp units-load-start: Wed 2020-08-26 16:33:43 EDT
Timestamp units-load-finish: Wed 2020-08-26 16:33:43 EDT
Timestamp initrd-security-start: Wed 2020-08-26 12:33:38 EDT
Timestamp initrd-security-finish: Wed 2020-08-26 12:33:38 EDT
Timestamp initrd-generators-start: Wed 2020-08-26 12:33:38 EDT
Timestamp initrd-generators-finish: Wed 2020-08-26 12:33:38 EDT
Timestamp initrd-units-load-start: Wed 2020-08-26 12:33:38 EDT
Timestamp initrd-units-load-finish: Wed 2020-08-26 12:33:38 EDT
-> Unit system.slice:
        Description: System Slice
        Instance: n/a
        Unit Load State: loaded
```

```
        Unit Active State: active
        State Change Timestamp: Wed 2020-08-26 12:33:38 EDT
        Inactive Exit Timestamp: Wed 2020-08-26 12:33:38 EDT
        Active Enter Timestamp: Wed 2020-08-26 12:33:38 EDT
        Active Exit Timestamp: n/a
        Inactive Enter Timestamp: n/a
        May GC: no
[...] # I've deleted a bazillion lines of output
```

On my main workstation, this command generated a stream of 49,680 lines and about 1.66MB. This command is very fast, so you don't need to wait for the results.

I do like the wealth of detail provided for the various connected devices, such as storage. Each systemd unit has a section with details such as modes for various runtimes, cache, and log directories, the command line used to start the unit, the process ID (PID), the start timestamp, as well as memory and file limits.

The man page for `systemd-analyze` shows the `systemd-analyze --user dump` option, which is intended to display information about the internal state of the user manager. This fails for me, and internet searches indicate that there may be a problem with it. In systemd, `--user` instances are used to manage and control the resources for the hierarchy of processes belonging to each user. The processes for each user are part of a control group, which I'll cover in a future article.

Analytic graphs

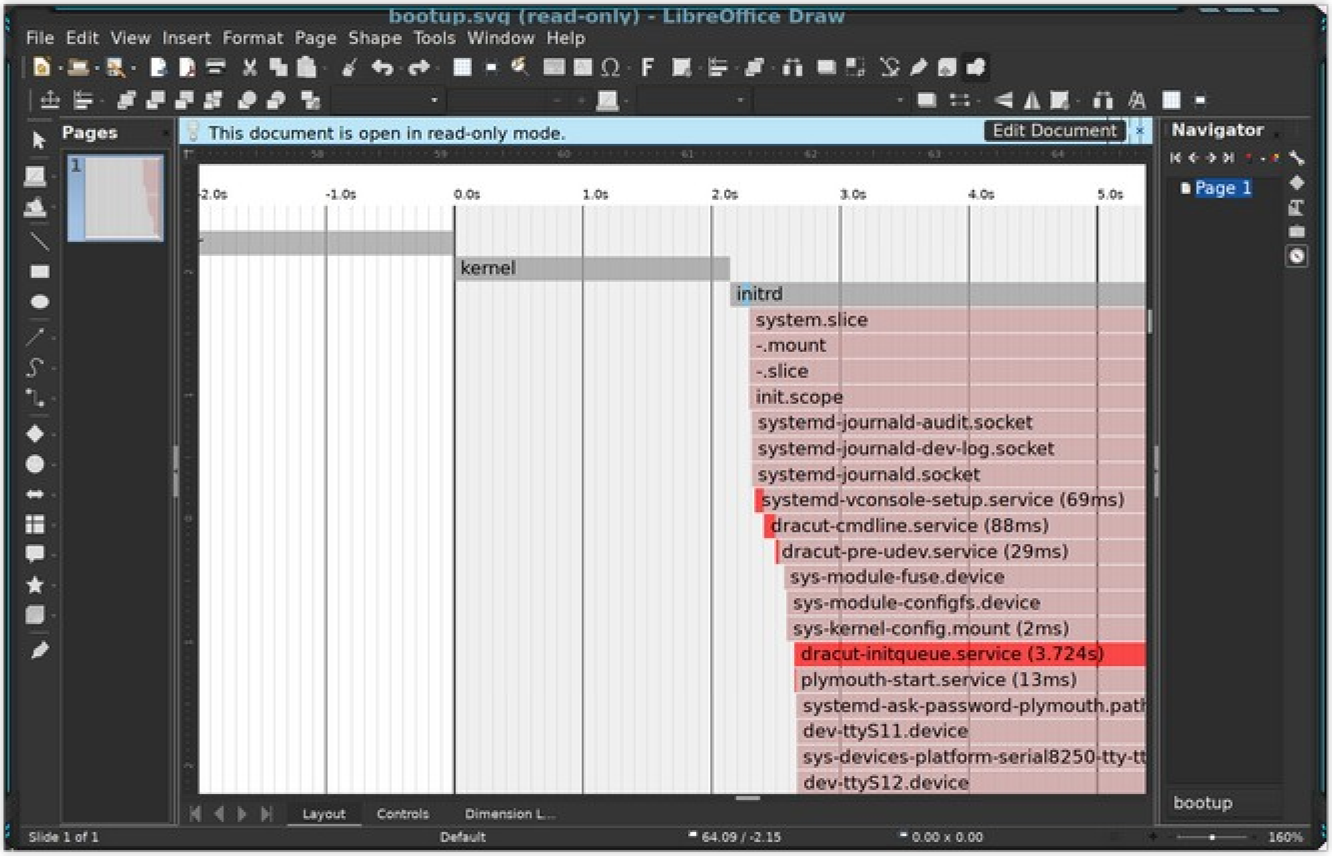
Most pointy-haired-bosses (PHBs) and many good managers find pretty graphs easier to read and understand than the text-based system performance data I usually prefer. Sometimes, though, even I like a good graph, and `systemd-analyze` provides the capability to display startup data in an SVG vector graphics chart.

The following command generates a vector graphics file that displays the events that take place during boot and startup. It only takes a few seconds to generate this file:

```
# systemd-analyze plot > /tmp/bootup.svg
```

This command creates an SVG, which is a text file that defines a series of graphic vectors that applications, including Image Viewer, Ristretto, Okular, Eye of Mate, LibreOffice Draw, and others, use to generate a graph. These applications process SVG files to create an image.

I used LibreOffice Draw to render a graph. The graph is huge, and you need to zoom in considerably to make out any detail. Here is a small portion of it:



The bootup sequence is to the left of the zero (0) on the timeline in the graph, and the startup sequence is to the right of zero. This small portion shows the kernel, `initrd`, and the processes `initrd` started.

This graph shows at a glance what started when, how long it took to start up, and the major dependencies. The critical path is highlighted in red.

Another command that generates graphical output is `systemd-analyze dot`. It generates textual dependency graph descriptions in [DOT](https://en.wikipedia.org/wiki/DOT_(graph_description_language)) format. The resulting data stream is then piped through the `dot` utility, which is part of a family of programs that can be used to generate vector graphic files from various types of data. These SVG files can also be processed by the tools listed above.

First, generate the file. This took almost nine minutes on my primary workstation:

```
# time systemd-analyze dot | dot -Tsvg > /tmp/test.svg
   Color legend: black     = Requires
```

```
                 dark blue = Requisite
                 dark grey = Wants
                 red       = Conflicts
                 green     = After

real    8m37.544s
user    8m35.375s
sys     0m0.070s
```

I won't reproduce the output here because the resulting graph is pretty much spaghetti. But you should try it and view the result to see what I mean.

Conditionals

One of the more interesting, yet somewhat generic, capabilities I discovered while reading the `systemd-analyze(1)` man page is the `condition` subcommand. (Yes—I do read the man pages, and it is amazing what I have learned this way!) This `condition` subcommand can be used to test the conditions and asserts that can be used in systemd unit files.

It can also be used in scripts to evaluate one or more conditions—it returns a zero (0) if all are met or a one (1) if any condition is not met. In either case, it also spews text about its findings.

The example below, from the man page, is a bit complex. It tests for a kernel version between 4.0 and 5.1, that the host is running on AC power, that the system architecture is anything but ARM, and that the directory `/etc/os-release` exists. I added the `echo $?` statement to print the return code.

```
# systemd-analyze condition 'ConditionKernelVersion = ! <4.0' \
                    'ConditionKernelVersion = >=5.1' \
                    'ConditionACPower=|false' \
                    'ConditionArchitecture=|!arm' \
                    'AssertPathExists=/etc/os-release' ; \
echo $?
test.service: AssertPathExists=/etc/os-release succeeded.
Asserts succeeded.
test.service: ConditionArchitecture=|!arm succeeded.
test.service: ConditionACPower=|false failed.
test.service: ConditionKernelVersion=>=5.1 succeeded.
test.service: ConditionKernelVersion=!<4.0 succeeded.
Conditions succeeded.
0
```

The list of conditions and asserts starts around line 600 on the `systemd.unit(5)` man page.

Listing configuration files

The `systemd-analyze` tool provides a way to send the contents of various configuration files to `STDOUT`, as shown here. The base directory is `/etc/`:

```
# systemd-analyze cat-config systemd/system/display-manager.service
# /etc/systemd/system/display-manager.service
[Unit]
Description=LXDM (Lightweight X11 Display Manager)
#Documentation=man:lxdm(8)
Conflicts=getty@tty1.service
After=systemd-user-sessions.service getty@tty1.service plymouth-quit.service
livesys-late.service
#Conflicts=plymouth-quit.service

[Service]
ExecStart=/usr/sbin/lxdm
Restart=always
IgnoreSIGPIPE=no
#BusName=org.freedesktop.lxdm

[Install]
Alias=display-manager.service
```

This is a lot of typing to do nothing more than a standard `cat` command does. I find the next command a tiny bit helpful. It can search out files with the specified pattern within the standard systemd locations:

```
# systemctl cat backup*
# /etc/systemd/system/backup.timer
# This timer unit runs the local backup program
# (C) David Both
# Licensed under GPL V2
#

[Unit]
Description=Perform system backups
Requires=backup.service

[Timer]
Unit=backup.service
OnCalendar=*-*-* 00:15:30

[Install]
WantedBy=timers.target


# /etc/systemd/system/backup.service
```

```
# This service unit runs the rsbu backup program
# By David Both
# Licensed under GPL V2
#

[Unit]
Description=Backup services using rsbu
Wants=backup.timer

[Service]
Type=oneshot
Environment="HOME=/root"
ExecStart=/usr/local/bin/rsbu -bvd1
ExecStart=/usr/local/bin/rsbu -buvd2

[Install]
WantedBy=multi-user.target
```

Both of these commands preface the contents of each file with a comment line containing the file's full path and name.

Unit file verification

After creating a new unit file, it can be helpful to verify that its syntax is correct. This is what the `verify` subcommand does. It can list directives that are spelled incorrectly and call out missing service units:

```
# systemd-analyze verify /etc/systemd/system/backup.service
```

Adhering to the Unix/Linux philosophy that "silence is golden," a lack of output messages means that there are no errors in the scanned file.

Security

The `security` subcommand checks the security level of specified services. It only works on service units and not on other types of unit files:

```
# systemd-analyze security display-manager
  NAME                                                 DESCRIPTION
>
✗ PrivateNetwork=                                      Service has access
to the host's network                        >
✗ User=/DynamicUser=                                   Service runs as
root user                                       >
```

```
✗ CapabilityBoundingSet=~CAP_SET(UID|GID|PCAP)               Service may change
UID/GID identities/capabilities             >
✗ CapabilityBoundingSet=~CAP_SYS_ADMIN                       Service has
administrator privileges                           >
✗ CapabilityBoundingSet=~CAP_SYS_PTRACE                      Service has
ptrace() debugging abilities                       >
✗ RestrictAddressFamilies=~AF_(INET|INET6)                   Service may
allocate Internet sockets                          >
✗ RestrictNamespaces=~CLONE_NEWUSER                          Service may create
user namespaces                              >
✗ RestrictAddressFamilies=~…                                 Service may
allocate exotic sockets                            >
[...]
✗ CapabilityBoundingSet=~CAP_SYS_TTY_CONFIG                  Service may issue
vhangup()                                     >
✗ CapabilityBoundingSet=~CAP_WAKE_ALARM                      Service may program
timers that wake up the system              >
✗ RestrictAddressFamilies=~AF_UNIX                           Service may
allocate local sockets                             >

→ Overall exposure level for backup.service: 9.6 UNSAFE ?
lines 34-81/81 (END)
```

Of course, many services need pretty much complete access to everything in order to do their work. I ran this program against several services, including my own backup service; the results may differ, but the bottom line seems to be mostly the same.

This tool would be very useful for checking and fixing userspace service units in security-critical environments. I don't think it has much to offer for most of us.

Final thoughts

This powerful tool offers some interesting and amazingly useful options. Much of what this article explores is about using `systemd-analyze` to provide insights into Linux's startup performance using systemd. It can also analyze other aspects of systemd.

Some of these tools are of limited use, and a couple should be forgotten completely. But most can be used to good effect when resolving problems with startup and other systemd functions.

Managing resources with cgroups in systemd

There is little more frustrating to me as a sysadmin than unexpectedly running out of a computing resource. On more than one occasion, I have filled all available disk space in a partition, run out of RAM, and not had enough CPU time to perform my tasks in a reasonable amount of time. Resource management is one of the most important tasks that sysadmins do.

The point of resource management is to ensure that all processes have relatively equal access to the system resources they need. Resource management also involves ensuring that RAM, hard drive space, and CPU capacity are added when necessary or rationed when that is not possible. In addition, users who hog system resources, whether intentionally or accidentally, should be prevented from doing so.

There are tools that enable sysadmins to monitor and manage various system resources. For example, top and similar tools allow you to monitor the use of memory, I/O, storage (disk, SSD, and so on), network, swap space, CPU usage, and more. These tools, particularly those that are CPU-centric, are mostly based on the paradigm that the running process is the unit of control. At best, they provide a way to adjust the nice number–and through that, the priority—or to kill a running process. (For information about nice numbers, see *Monitoring Linux and Windows hosts with Glances*.)

Other tools based on traditional resource management in a SystemV environment are managed by the `/etc/security/limits.conf` file and the local configuration files located in the `/etc/security/limits.d` directory. Resources can be limited in a fairly crude but useful manner by user or group. Resources that can be managed include various aspects of RAM, total CPU time per day, total amount of data, priority, nice number, number of concurrent logins, number of processes, maximum file size, and more.

Creative Commons Attribution Share-alike 4.0

Using cgroups for process management

One major difference between systemd and SystemV is how they handle processes. SystemV treats each process as an entity unto itself. systemd collects related processes into control groups, called cgroups (short for control groups), and manages system resources for the cgroup as a whole. This means resources can be managed per application rather than by the individual processes that make up an application.

The control units for cgroups are called slice units. Slices are a conceptualization that allows systemd to order processes in a tree format for ease of management.

Viewing cgroups

I'll start with some commands that allow you to view various types of information about cgroups. The `systemctl status <service>` command displays slice information about a specified service, including its slice. This example shows the `at` daemon:

```
# systemctl status atd.service
● atd.service - Deferred execution scheduler
     Loaded: loaded (/usr/lib/systemd/system/atd.service; enabled; vendor preset:
enabled)
     Active: active (running) since Wed 2020-09-23 12:18:24 EDT; 1 day 3h ago
       Docs: man:atd(8)
   Main PID: 1010 (atd)
      Tasks: 1 (limit: 14760)
     Memory: 440.0K
        CPU: 5ms
     CGroup: /system.slice/atd.service
             └─1010 /usr/sbin/atd -f

Sep 23 12:18:24 testvm1.both.org systemd[1]: Started Deferred execution
scheduler.
```

This is an excellent example of one reason that I find systemd more usable than SystemV and the old init program. There is so much more information here than SystemV could provide. The cgroup entry includes the hierarchical structure where the `system.slice` is systemd (PID 1), and the `atd.service` is one level below and part of the `system.slice`. The second line of the cgroup entry also shows the process ID (PID) and the command used to start the daemon.

The `systemctl` command shows multiple cgroup entries. The `--all` option shows all slices, including those that are not currently active:

```
# systemctl -t slice --all
  UNIT                             LOAD   ACTIVE   SUB    DESCRIPTION
  -.slice                          loaded active   active Root Slice
  system-getty.slice               loaded active   active system-getty.slice
  system-lvm2\x2dpvscan.slice      loaded active   active system-lvm2\
x2dpvscan.slice
  system-modprobe.slice            loaded active   active system-modprobe.slice
  system-sshd\x2dkeygen.slice      loaded active   active system-sshd\
x2dkeygen.slice
  system-systemd\x2dcoredump.slice loaded inactive dead   system-systemd\
x2dcoredump.slice
  system-systemd\x2dfsck.slice     loaded active   active system-systemd\
x2dfsck.slice
  system.slice                     loaded active   active System Slice
  user-0.slice                     loaded active   active User Slice of UID 0
  user-1000.slice                  loaded active   active User Slice of UID 1000
  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.

11 loaded units listed.
To show all installed unit files use 'systemctl list-unit-files'.
```

The first thing to notice about this data is that it shows user slices for UIDs 0 (root) and 1000, which is my user login. This shows only the slices and not the services that are part of each slice. This data shows that a slice is created for each user at the time they log in. This can provide a way to manage all of a user's tasks as a single cgroup entity.

Explore the cgroup hierarchy

All is well and good so far, but cgroups are hierarchical, and all of the service units run as members of one of the cgroups. Viewing that hierarchy is easy and uses one old command and one new one that is part of systemd.

The `ps` command can be used to map the processes and their locations in the cgroup hierarchy. Note that it is necessary to specify the desired data columns when using the `ps` command. I significantly reduced the volume of output from this command below, but I tried to leave enough so you can get a feel for what you might find on your systems:

```
# ps xawf -eo pid,user,cgroup,args
    PID USER     CGROUP                          COMMAND
      2 root     -                               [kthreadd]
      3 root     -                                \_ [rcu_gp]
      4 root     -                                \_ [rcu_par_gp]
      6 root     -                                \_ [kworker/0:0H-kblockd]
      9 root     -                                \_ [mm_percpu_wq]
     10 root     -                                \_ [ksoftirqd/0]
     11 root     -                                \_ [rcu_sched]
     12 root     -                                \_ [migration/0]
     13 root     -                                \_ [cpuhp/0]
     14 root     -                                \_ [cpuhp/1]
<SNIP>
 625406 root     -                                \_ [kworker/3:0-ata_sff]
 625409 root     -                                \_ [kworker/u8:0-events_unbound]
      1 root     0::/init.scope                  /usr/lib/systemd/systemd --switched-
root --system --deserialize 30
    588 root     0::/system.slice/systemd-jo /usr/lib/systemd/systemd-journald
    599 root     0::/system.slice/systemd-ud /usr/lib/systemd/systemd-udevd
    741 root     0::/system.slice/auditd.ser /sbin/auditd
    743 root     0::/system.slice/auditd.ser  \_ /usr/sbin/sedispatch
    764 root     0::/system.slice/ModemManag /usr/sbin/ModemManager
    765 root     0::/system.slice/NetworkMan /usr/sbin/NetworkManager --no-daemon
    767 root     0::/system.slice/irqbalance /usr/sbin/irqbalance --foreground
    779 root     0::/system.slice/mcelog.ser /usr/sbin/mcelog --ignorenodev --
daemon --foreground
    781 root     0::/system.slice/rngd.servi /sbin/rngd -f
    782 root     0::/system.slice/rsyslog.se /usr/sbin/rsyslogd -n
<SNIP>
    893 root     0::/system.slice/sshd.servi sshd: /usr/sbin/sshd -D [listener] 0
of 10-100 startups
   1130 root     0::/user.slice/user-0.slice  \_ sshd: root [priv]
   1147 root     0::/user.slice/user-0.slice  |   \_ sshd: root@pts/0
   1148 root     0::/user.slice/user-0.slice  |       \_ -bash
   1321 root     0::/user.slice/user-0.slice  |           \_ screen
   1322 root     0::/user.slice/user-0.slice  |               \_ SCREEN
   1323 root     0::/user.slice/user-0.slice  |                   \_ /bin/bash
 498801 root     0::/user.slice/user-0.slice  |                   |   \_ man
systemd.resource-control
[...]
```

You can view the entire hierarchy with the `systemd-cgls` command, which is a bit simpler because it does not require any complex options.

I have shortened this tree view considerably. as well, but I left enough to give you some idea of the amount of data as well as the types of entries you should see when you do this on your system. I did this on one of my virtual machines, and it is about 200 lines long; the amount of data from my primary workstation is about 250 lines:

```
# systemd-cgls
Control group /:
-.slice
├─user.slice
│ ├─user-0.slice
│ │ ├─session-1.scope
│ │ │ ├─  1130 sshd: root [priv]
│ │ │ ├─  1147 sshd: root@pts/0
│ │ │ ├─  1148 -bash
│ │ │ ├─  1321 screen
│ │ │ ├─  1322 SCREEN
│ │ │ ├─  1323 /bin/bash
│ │ │ ├─  1351 /bin/bash
│ │ │ ├─  1380 /bin/bash
│ │ │ ├─123293 man systemd.slice
│ │ │ ├─123305 less
│ │ │ ├─246795 /bin/bash
│ │ │ ├─371371 man systemd-cgls
│ │ │ ├─371383 less
│ │ │ ├─371469 systemd-cgls
│ │ │ └─371470 less
│ │ └─user@0.service …
│ │   ├─dbus-broker.service
│ │   │ ├─1170 /usr/bin/dbus-broker-launch --scope user
│ │   │ └─1171 dbus-broker --log 4 --controller 12 --machine-id
3bccd1140fca488187f8a1439c832f07 --max-bytes 100000000000000 --max-fds
25000000000000 --max->
│ │   ├─gvfs-daemon.service
│ │   │ └─1173 /usr/libexec/gvfsd
│ │   └─init.scope
│ │     ├─1137 /usr/lib/systemd/systemd --user
│ │     └─1138 (sd-pam)
│ └─user-1000.slice
│   ├─user@1000.service …
│   │ ├─dbus\x2d:1.2\x2dorg.xfce.Xfconf.slice
│   │ │ └─dbus-:1.2-org.xfce.Xfconf@0.service
│   │ │   └─370748 /usr/lib64/xfce4/xfconf/xfconfd
[...]
```

This tree view shows all of the user and system slices and the services and programs running in each cgroup. Notice the units called "scopes," which group related programs into a management unit, within the `user-1000.slice` in the listing above. The `user-1000.slice/session-7.scope` cgroup contains the GUI desktop program hierarchy, starting with the LXDM display manager session and all of its subtasks, including things like the Bash shell and the Thunar GUI file manager.

Scope units are not defined in configuration files but are generated programmatically as the result of starting groups of related programs. Scope units do not create or start the processes running as part of that cgroup. All processes within the scope are equal, and there is no internal hierarchy. The life of a scope begins when the first process is created and ends when the last process is destroyed.

Open several windows on your desktop, such as terminal emulators, LibreOffice, or whatever you want, then switch to an available virtual console and start something like `top` or Midnight Commander. Run the `systemd-cgls` command on your host, and take note of the overall hierarchy and the scope units.

The `systemd-cgls` command provides a more complete representation of the cgroup hierarchy (and details of the units that make it up) than any other command I have found. I prefer its cleaner representation of the tree than what the `ps` command provides.

More information

After covering these basics, I had planned to go into more detail about cgroups and how to use them, but I discovered a series of four excellent articles by Red Hat's Steve Ovens on Opensource.com's sister site Enable Sysadmin. Rather then basically rewriting Steve's articles, I decided it would be much better to take advantage of his cgroup expertise by linking to them:

1. A Linux sysadmin's introduction to cgroups
2. How to manage cgroups with CPUShares
3. Managing cgroups the hard way—manually
4. Managing cgroups with systemd

Enjoy and learn from them, as I did.