White Paper

Server

systemd in SUSE® Linux Enterprise 12

A kinder, gentler introduction from SUSE

Table of Contents

SUSE
We adapt. You succeed.

Abstract

Introduced with SUSE® Linux Enterprise 12, systemd is the new system startup and service manager for Linux, replacing the old System V init (SysV init). Now you can forget everything you ever knew about how Linux systems start up and manage services. systemd is *different*, and you need to understand how to work with it.

Because of this, it has quite a learning curve, especially because it introduces a number of new concepts and commands. However, once you get over that initial hump, you will find systemd a powerful and flexible tool.

According to the developers, systemd "provides aggressive parallelization capabilities, uses socket and D-Bus activation[1] for starting services, offers on-demand starting of daemons, keeps track of processes using Linux cgroups, supports snapshotting and restoring of the system state, maintains mount and automount points and implements an elaborate transactional dependency-based service control logic."[2] It also promotes run-on sentences and excessive fervor in its adherents.

This paper covers the basics of systemd with an emphasis on providing correlations between how things were done with SysV init and how they need to be done now with systemd.

1 *If you don't know what D-Bus is, don't worry. It's essentially an abstraction layer for inter-process communications (IPC). You can think of it as a more structured means to perform IPC without every process wishing to use it having to figure it out for themselves.*

2 **www.freedesktop.org/wiki/Software/systemd/**

Introduction

The move to systemd has been very controversial in the broader Linux community. Critics have questioned why a replacement for SysV init is needed in the first place, the design of systemd and its implementation. While this paper won't delve into all that background, it will talk about a couple of points.

Why Replace SysV init?

The SysV init concept has been around for a long time. For the most part, it works well. There are a number of areas where it could be improved, however:

- *It's slow.*
- *It's hard to parallelize.*
- *The concept of runlevels is rather coarse.*
- *Linux Standard Base (LSB) dependencies only do part of what most system administrators need.*
- *It lacks automatic restart of services (apart from what is in* `/etc/inittab`*).*
- *It lacks unified logging.*
- *It lacks unified resource limit handling.*

Many Independent Software Vendors (ISVs) don't provide init scripts for their products. While that behavior probably won't change for systemd, the hope is that writing systemd service descriptions will be more straightforward and reliable than writing shell scripts.

SUSE understands the need for some level of backward compatibility to help users and administrators alike adopt systemd more easily. This paper highlights some of these enhancements as they are mentioned.

Basic systemd Concepts

Not only is systemd a replacement for the init process itself, but also all the infrastructure that is built on top of it.

The Linux kernel by default starts one and only one "user space" process: `/sbin/init`. (It's a little more complex than that, but for here it's accurate enough.) This process is then responsible for the bring-up of all services in the system, all connected ttys on which to log in, which file systems need to be mounted, etc. With SysV init, terminals (or "consoles") are usually defined by adding an entry to the `/etc/inittab` file. Most system services are started by init using scripts that reside in `/etc/init.d`. These init scripts are executed by a variety of helper scripts as the system is brought up or change from one runlevel to another. In addition to these services, there are boot scripts that reside in `/etc/init.d/boot.d`, which are executed only once during the initial bring-up stages.

With systemd, `/sbin/init` is a symbolic link to systemd. As a result, just as before, systemd is started as the first process and runs with a process ID (PID) of 1. But unlike SysV init, systemd strives to unify the way system resources are handled. To this end, systemd introduces the concept of a unit file, capable of describing just about any sort of boot activity, system service or resource that needs to be brought up and managed, and `/etc/inittab` no longer even exists. (More on this later.)

systemd does still support init scripts. Thus, if you have written your own init scripts to bring up your own services, there's probably nothing or only very little you will have to do. If you want, you can wait until later to convert them to systemd service files. SUSE Linux Enterprise 12 itself is still using scripts for a few services. To be clear, though, there are some limitations and differences in how systemd handles them.

Refer to **www.freedesktop.org/wiki/Software/systemd/Incompatibilities/** for details.

Additionally, SUSE Linux Enterprise 12 still supports `/etc/sysconfig` files to control the runtime behavior of services; the majority of these variables haven't changed from what was supported in SUSE Linux Enterprise 11.

Runlevels and Targets

As mentioned previously, SysV init uses the concept of runlevels. These runlevels were used to define different system configurations or system states. Typical Linux systems support runlevels 1, 3 and 5. Runlevel 1 (or S) describes the so-called single-user configuration which can be used by the administrator to perform intrusive management tasks; runlevel 3 is a multi-user configuration with the network and all services active, but no graphical console. Finally, runlevel 5 is typically used for a graphical workstation type of configuration.

systemd supports a similar concept, called "targets." In general, targets are used as synchronization points in the boot process and can be assigned almost arbitrary names. A good number of pre-defined targets are included with the systemd package, so take a look at those before deciding you need to create your own.

Dependencies and Parallelization

One the more important aspects of both init scripts and systemd unit files is that they need to express dependencies between services and boot activities. For instance, there's not much use in trying to mount a remote file system over NFS before the network has been brought up. In fact, trying to do so will likely be a painful experience. Both SysV init scripts and systemd unit files allow you to create those dependencies, albeit in a different way. However, in the systemd world, the dependency information is much more pervasive and fine-grained, allowing systemd to be much more aggressive in parallelizing system startup. Conversely, systemd also allows the system administrator more control in ensuring the order that services are started matches their intent.

Unit Files

As mentioned, the way to describe things to systemd is via unit files. Unit files take a much more standardized approach to describing a service than the LSB header: while the latter was added as a bit of an afterthought to make scripts more manageable, systemd's unit files are an integral part of the design. Unit files primarily serve as a collection of meta information, with perhaps a few lines of shell code that actually start, stop or restart the service.

For example, see Figure 1 for the systemd unit file for the cron service, which can be found at `/usr/lib/systemd/system/cron.service`.

```
[Unit]
Description=Command Scheduler
After=ypbind.service nscd.service network.target
After=postfix.service sendmail.service exim.
service

[Service]
ExecStart=/usr/sbin/cron -n
Restart=on-abort

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

Figure 1

Contrast that with the init script for cron, which has over 140 lines of shell script and comments. As you can see, you can be quite specific as to what other services you want to be started before (or after) the service in this unit file. This is a big improvement over the largely frustrating process of trying to do the same thing with the LSB header in an init script.

Also, notice how the Service section specifies a restart option. This is something that was tedious to do for SysV init scripts and intrusive as well because it had to be done manually inside each init script. This perhaps was the main reason why some ISVs would have their services controlled through an entry in `/etc/inittab`: SysV init allows you to specify automatic restart for entries in `inittab`.

systemd provides that facility "out of the box" for any kind of service; you just have to enable it with a single statement in the unit file. Of course, this is not the only useful facility built into systemd. You can also specify a userid for a service to use when it runs, restrict its security capabilities and even place limits on its use of memory or the device files it can access, by using control groups and name spaces. More on this later.

Although this paper only discusses the service and target types of unit files in any detail, Figure 2 shows there is a rather wide variety of unit file types.

automount	scope	swap
busname	service	target
device	slice	timer
mount	snapshot	
path	socket	

Figure 2

There are man pages that go into great detail on each of these types of unit files. You can access them by `man systemd.<unittype>`. For example, `man systemd.mount` or `man systemd.service`. The documentation for busname unit files is included in the man page for systemd.service.

There is a naming convention for just about all unit files. Some of the conventions are more obvious than others, e.g., unit files for services are named after the services themselves, such as `sshd.service` or `rpcbind.service`.

Mount units must be named after the mount point directories they control, e.g., `var-tmp.mount` for `/var/tmp/` or `var-spool.mount` for `/var/spool/`.

The systemd Journal

The systemd journal is a new logging scheme that provides "structured, indexed and reliable logging on systemd systems."[3] In addition to recording the regular syslog messages generated by a service, it also captures the standard output of daemons, which previously went to the null device.

While it is possible to use the journal to completely replace the classic syslog implementations, that has not been done in SUSE Linux Enterprise 12. Instead, it uses the provided capabilities of the journal to send the data to rsyslog.

One of the nice features of the journal is that `systemctl status` will also show the last ten log messages of the daemon. If nothing else, it avoids having to search for the file that contains the logs for a daemon just to see those last lines. There are many other features of the journal, which can be found in the man page for the `journalctl` command.

Very Quick and Dirty Overview of Useful Commands

If you're getting impatient by this point, you don't have to read all the way to the end before you can do something useful. If you're the type of person that likes to try something and see how it works, check Appendix A. If you want more details, keep reading here.

Basic Operations with systemd

Your main interface to systemd is the `systemctl` command. You'll likely find yourself typing that a *lot* so it might make sense to create a shorter alias for it in your `~/.alias` file.

3 **http://0pointer.de/blog/projects/systemctl-journal.html**

Runlevels and Targets

As documented in `/etc/inittab` on systems using SysV init:

Runlevel 0 is System halt.
Runlevel 1 is Single user mode
Runlevel 2 is Local multiuser without remote network (e.g. NFS)
Runlevel 3 is Full multiuser with network
Runlevel 4 is Not used
Runlevel 5 is Full multiuser with network and xdm
Runlevel 6 is System reboot

The equivalent targets in systemd look like this:

Runlevel	Equivalent systemd target
0	`poweroff.target`
1	`rescue.target`
2, 3, 4	`multi-user.target`
5	`graphical.target`
6	`reboot.target`

To change to a new target (runlevel), you will use the systemctl command. For example, to change to single-user mode:

```
# systemctl isolate rescue.target
```

Similarly, to change to the non-graphical mode:

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

Note that ".target" *must* be specified on the command. If you don't, you'll see this:

```
# systemctl isolate rescue
Failed to start rescue.service: Operation refused,
unit may not be isolated.
```

There are some apparent exceptions to this, which really aren't. For example, you can issue commands such as this:

```
# systemctl default
```

which will put the system in the default target configuration, or

```
# systemctl reboot
```

which will reboot the system. These are in fact systemd "system commands" and are documented in the `systemctl` man page under "System Commands." There are about a dozen of them listed there. For things like `default, halt, reboot` and some others, the result is the same as issuing `systemctl isolate default.target, systemctl isolate halt.target`, etc. The difference is that the system command forms will also send a `wall` message to all users whereas the isolate form will not.

For compatibility purposes, aliases to various targets have been provided in systemd:
runlevel0.target -> poweroff.target
runlevel1.target -> rescue.target
runlevel2.target -> multi-user.target
runlevel3.target -> multi-user.target
runlevel4.target -> multi-user.target
runlevel5.target -> graphical.target
runlevel6.target -> reboot.target

This will allow you to do things like

```
# systemctl isolate runlevel5.target
```

where with SysV init you would execute

```
# telinit 5
```

(Actually, if you have the systemd-sysvinit RPM installed, you can still use the `telinit` command to do this.)

SETTING AND QUERYING THE DEFAULT TARGET (RUNLEVEL)

You can determine the default target with `systemctl get-default`

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

You can set the default target with `systemctl set-default`

```
# systemctl set-default graphical.target
# systemctl set-default multi-user.target
```

Just as with SysV init, you must *not* set the default target to poweroff.target or reboot.target. If you do, you'll get the same result with systemd: a system that either immediately shuts itself down or reboots endlessly.

Enabling and Disabling Services

Over the years, multiple ways have been created to enable or disable system services. The two most used on SUSE Llinux Enterprise are chkconfig and insserv. With the introduction of systemd, the functionality of those two tools is rather limited.

For example, `chkconfig` can really only be used to manage services that have an init script in `/etc/init.d/`. For packages that have a systemd service unit file, `chkconfig on` and `chkconfig off` can be used to enable or disable the service and not much else. The insserv command will only work with init scripts in `/etc/init.d/`. Any attempt to use insserv with a systemd unit file will result in messages such as this:

```
# insserv -r nscd
Warning: /etc/init.d/nscd is masked by
/usr/lib/systemd/system/nscd.service.
Try 'chkconfig nscd off' instead
```

The more realistic recommendation would be to use `systemctl disable nscd` instead.

Enabling or disabling services should not be confused with starting (activating) or stopping (deactivating) services, as done by the start and stop commands. Enabling/disabling and starting/stopping services are completely separate actions. Start/stop only affects the currently running system and is not persistent. Enabling/disabling affects the system at the next runlevel (target) change and *is* persistent.

The systemd equivalent to chkconfig and insserv is, you guessed it, `systemctl`.

```
systemctl enable xinetd.service
systemctl disable xinetd.service
```

Also note that we're not specifying which targets we want xinetd to run in. The reason is the unit file itself tells systemd when it should be running or not.

To determine if a service is enabled or not

```
systemctl is-enabled xinetd.service
```

will print one of a number of possible values:

Printed string	Meaning	Return value
enabled **enabled-runtime**	The service is enabled and will be started at the next runlevel (target) change.	0
disabled	The service is not enabled, but can be started and stopped manually.	1
masked **masked-runtime**	The service is disabled and cannot be started, even manually.	1
linked **linked-runtime**	The service is enabled and will be started at the next runlevel (target) change.	1
static[4]	The service is *not* enabled and has no provisions in the [Install] section for being enabled. It can be started manually and will be started automatically if another unit file that is started needs it.	0

In the three "-runtime" status entries, -"runtime" indicates that a status is only temporary and will be lost at the next reboot.

Also note the masked status. This is the result of using `systemctl mask` for a unit file. As noted, this will prevent the unit from ever being started, even by a `systemctl start` command. You probably won't use this very often, but it can be helpful in situations where you absolutely want to be sure a service won't be started automatically or by accident. The converse of the command is `systemctl unmask`.

Starting, Stopping and Checking Service Status

The way you would start and stop the cron service shown above is thus:

```
systemctl start cron.service
systemctl stop cron.service
systemctl restart cron.service
```

4 Units that are shown as "static" can neither be enabled nor disabled. These are units that will be activated if another service needs them to run. Instead of `systemctl enable` or `systemctl disable`, use `systemctl unmask` or `systemctl mask` respectively.

Normally, these commands do not display much in terms of feedback. To be sure, follow the start command with `systemctl status` to view the actual status of the service.

If there is a problem, there are a variety of messages you might get:

```
Job for cron.service failed.  See 'systemctl status
cron.service' and 'journalctl -xn' for details.

Failed to reload cron.service: Job type reload is
not applicable for unit cron.service.
```

The `systemctl status` command will provide the current status of the service as well as some entries from the system log for it:

```
# systemctl status cron.service
cron.service - Command Scheduler
   Loaded: loaded (/usr/lib/systemd/system/cron.
service; enabled)
   Active: active (running) since Thu 2014-06-26
19:35:09 EDT; 50s ago
 Main PID: 30596 (cron)
   CGroup: /system.slice/cron.service
           └─30596 /usr/sbin/cron -n

Jun 26 19:35:09 s390vsl210 systemd[1]: Starting
Command Scheduler...
Jun 26 19:35:09 s390vsl210 systemd[1]: Started
Command Scheduler.
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(RANDOM_DELAY will be scaled with ...d.)
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(running with inotify support)
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(@reboot jobs will be run at compu...p.)
Hint: Some lines were ellipsized, use -l to show in
full.

# systemctl -l status cron.service
cron.service - Command Scheduler
   Loaded: loaded (/usr/lib/systemd/system/cron.
service; enabled)
   Active: active (running) since Thu 2014-06-26
```

```
19:35:09 EDT; 1s ago
 Main PID: 30596 (cron)
   CGroup: /system.slice/cron.service
           └─30596 /usr/sbin/cron -n

Jun 26 19:35:09 s390vsl210 systemd[1]: Starting
Command Scheduler...
Jun 26 19:35:09 s390vsl210 systemd[1]: Started
Command Scheduler.
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(RANDOM_DELAY will be scaled with factor 66% if
used.)
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(running with inotify support)
Jun 26 19:35:09 s390vsl210 cron[30596]: (CRON) INFO
(@reboot jobs will be run at computer's startup.)
```

Some services, such as cron, still provide an init script, along with a systemd unit file. These init scripts should be sourcing the `/etc/rc.status` script very early on. The `/etc/rc.status` script will determine if systemd is running and redirect the command to it. If you really want the init script to be executed, simply set the `SYSTEMD_NO_WRAP` environment variable to any non-null value when invoking it. Be aware, though, that using this environment variable means systemd will *not* be monitoring the service started by the init script. Therefore, this should only be used for debugging purposes and not in production. A good way to debug an init script startup/shutdown sequence would be to change its "shebang" to `#!/bin/sh -x` or `#!/bin/bash -x` and check `journalctl` output.

```
# rccron stop
redirecting to systemctl stop cron.service

# SYSTEMD_NO_WRAP=x rccron stop
Shutting down CRON daemon                          done

# SYSTEMD_NO_WRAP= rccron stop
redirecting to systemctl stop cron.service

# SYSTEMD_NO_WRAP= /etc/init.d/cron stop
redirecting to systemctl stop cron.service

# SYSTEMD_NO_WRAP=z /etc/init.d/cron stop
Shutting down CRON daemon
```

Managing the systemd Daemon

With SysV init, when the system administrator would need to modify `/etc/inittab`, they had to always remember to tell init to re-examine the contents of the file. With systemd that is frequently done implicitly as the result of other commands being issued. However, when it does need to be done manually, that can be accomplished as follows:

SysV init command	systemd command
`telinit q` `telinit Q`	`systemctl daemon-reload`

For those cases where the daemon needs to be restarted, e.g., when the binary has been replaced by maintenance, then:

SysV init command	systemd command
`telinit u` `telinit U`	`systemctl daemon-rexec`

Socket Activation for Services

systemd provides for activating services when a socket connection is made to a listening port, much like inetd and xinetd do for network services. systemd takes the concept further, extending it to include local "bus-based" services such as D-bus as well as others such as multipathd, dm-event, syslog, etc. Looking in `/usr/lib/systemd/system/` you'll see a number of socket unit files. In most cases, but not all, you'll notice that there is a corresponding service unit file:

- `dm-event.service` and `dm-event.socket`
- `iscsid.service` and `iscsid.socket`
- `rpcbind.service` and `rpcbind.socket`

The reasoning behind this separation of the socket and the program that "services" the socket is somewhat sophisticated[5]. It reduces the time it takes a system to boot by starting fewer services, but more of them in parallel:

5 See **http://0pointer.de/blog/projects/systemd.html** and **http://0pointer.de/blog/projects/socket-activation.html**

- *systemd can create all the sockets ahead of time without having to wait for each daemon to initialize.*
- *Services that wait until a socket from another daemon, say syslog, is available can start sooner and have activity over the socket buffered until the syslog daemon is fully initialized.*
- *By not starting a service—for example, sshd—until a connection is made, infrequently used daemons won't be consuming system resources during boot or normal system activity.*

Having this logical separation can lead to some confusion, however. In many cases the socket unit files can be enabled, disabled, started and queried independent of the service unit. Enabling, disabling and starting the service should result in the same action being taken for the socket, since the service depends on the socket. Because of that dependency, stopping a socket should result in the service being stopped. Stopping the service, though, does not mean the socket will be stopped. The rpcbind service is a good example of this:

```
# systemctl stop rpcbind.service
Warning: Stopping rpcbind.service, but it can still
be activated by:
  rpcbind.socket
```

What that message means is that systemd will remain listening for connections on the rpcbind socket. If a connections is received, systemd will start the rpcbind daemon to handle it.. To truly stop rpcbind from running, you must stop the rpcbind. socket unit. In many cases, it may be easiest to try and remember to "Start the *service*" and "Stop the *socket*."

Local File System Handling

If you're used to adding file systems via the YaST® tool or manually editing `/etc/fstab`, then you don't need to do anything differently with systemd. YaST will still update `/etc/fstab` with the information necessary to mount the file system. Using YaST is the preferred method for adding/changing file system mounts. The systemd developers recommend using `fstab` entries wherever possible. However, if you need more control over what file systems get mounted and when, or need a feature that isn't supported in `fstab`, then you can create a systemd mount unit file.

For example, assume you've added a new disk to a system, partitioned it and created an ext3 file system. The new disk is named `/dev/sdb`, and you want to mount the first partition on `/sysdtest`. The unit file to have it mounted at boot time would be `/etc/systemd/system/sysdtest.mount` and it might look like Figure 3.

```
[Unit]
Description=systemd testing directory
Before=local-fs.target
# skip mounting if the directory does not exist
or is a symlink
ConditionPathIsDirectory=/sysdtest
ConditionPathIsSymbolicLink=!/sysdtest

[Mount]
What=/dev/sdb1
Where=/sysdtest
Type=ext3

[Install]
WantedBy = local-fs.target
```

Figure 3

Understand that simply creating the unit file will not cause the file system to be automatically mounted at the next boot. You must also enable the mount:

```
# systemctl enable sysdtest.mount
```

This will cause systemd to create a symbolic link to the unit file in the `/etc/systemd/system/local-fs.target.wants` directory.

Once the unit has been enabled, the actual mount will be performed the next time the system reaches the local-fs.target during the boot process. If you want it to happen immediately, then

```
# systemctl start sysdtest.mount
```

will accomplish that.

Conversely,

```
# systemctl stop sysdtest.mount
```

will cause the file system to be unmounted immediately, and

```
# systemctl disable sysdtest.mount
```

will cause the symbolic link to be deleted and the file system will not be mounted again automatically.

See `man systemd.mount` for the other entries that are specific to mount unit files.

Remote File System Handling

Working with remote file systems, such as over NFS, is similar to working with local file systems. Of course there are differences due to the need for a working network connection, etc. The preferred method here also is to continue to use YaST and entries such as the following will be added to `/etc/fstab`:

```
remote.host.name:/nfs/export /sysdtest      nfs
defaults   0 0
```

The normal system startup will ensure that the network and the necessary RPC-related daemons are started before the mount is attempted.

Again, if you believe you need to create a mount unit file, it might look like Figure 4.

```
[Unit]
Description=systemd testing directory
Before=remote-fs.target
After=nfs.service
# skip mounting if the directory does not exist
or is a symlink
ConditionPathIsDirectory=/sysdtest
ConditionPathIsSymbolicLink=!/sysdtest

[Mount]
What=remote.host.name:/nfs/export
Where=/sysdtest
Type=nfs

[Install]
WantedBy = remote-fs.target
```

Figure 4

Note the addition of the `After=` value for the nfs service. If you're writing your own unit files, you are reponsible for knowing what has to be started before or after your unit and describing those dependencies in the unit file.

Backward Compatible Support for init Scripts

To ease the transition to systemd, SUSE is providing backward compatible support for init scripts in a number of ways. For example, in the SysV init environment, init scripts can have a special header that provides some meta information on the service: a short name, the preferred runlevels to be run in, which other services need to be started before this one in order for it to work, etc. This header is called the "LSB header," named after the "Linux Standard Base" working group that defined it[6]. *(See Appendix D for an example from SUSE Linux Enterprise 11.)* SUSE will continue to support these headers in systemd, including the use of the special "LSB targets" or dependencies such as $network, $remote_fs, $portmap, etc.

Furthermore, many services in SUSE Linux Enterprise have had symbolic links in `/sbin` or `/usr/sbin` pointing to the actual init script in `/etc/init.d`, for example, rccron, rcpostfix, rcsshd. Those symbolic links still exist (even if the package no longer contains an init script), but may point either to an init script in `/etc/init.d`, or to `/usr/sbin/service`. If the symbolic link points to `/usr/sbin/service`, it will determine if the command should be redirected to `systemctl` or not and proceed accordingly.

A significant caveat applies to the use of the "rcfoo" names to start services, however. When a service is invoked by its rc* symbolic link, no attempt is made to ensure that any prerequisite services are started first, just as happens with SUSE Linux Enterprise 11. So, for example, if you enter `rcnfs start` and rpcbind is not running, the command will fail. In contrast, when using `systemctl start nfs.service`, systemd *will* start any prerequisite services first.

Finally, use of chkconfig, insserv and the service command will still be provided. Be aware that their functionality has changed somewhat and, in most cases, will be calling on systemd to perform the actual work requested.

6 **www.linuxfoundation.org/collaborate/workgroups/lsb**

systemd Can Do More

Control Groups, and How systemd Uses Them

systemd puts each service and each session into a separate control group (cgroup). These control groups are a kernel feature and can be used in a variety of useful ways. Perhaps most importantly, control groups provide a means of tracking which processes belong to a given service. If you've ever had to deal with the unclean shutdown of, say, an application server, and had to hunt down all its helper processes and agents and kill them manually, you will probably appreciate the `systemd-cgls` command and its companion, the `systemctl kill` command.

Sessions also get assigned an audit ID matching their cgroup ID. You can restrict these cgroups in all the ways the kernel supports: I/O bandwidth, memory or CPU consumption, etc.

Very much like the `pstree` command, `systemd-cgls` provides an ASCII tree representation of all the control groups on the system. See Figure 5.

```
# systemd-cgls /sys/fs/cgroup/systemd/system.
slice/postfix.service
/sys/fs/cgroup/systemd/system.slice/postfix.
service:
└─2619 /usr/lib/postfix/master -w
└─2621 pickup -l -t fifo -u
└─2622 qmgr -l -t fifo -u
```

Figure 5

In this example, you can see the control group associated with the postfix service, and all processes that were started as part of this service. That's much more useful than pstree, isn't it? But there's more. Assume your service got stuck in some weird error state, and you're no longer able to shut it down cleanly. Rather than killing off processes manually by referring to their PID (and keeping your fingers crossed that you're not mistyping anything), you can use a command like this:

```
# systemctl kill postfix.service
```

This will kill *all* processes in the control group associated with this service. Done! And if you did mistype the service name, you're not going to kill some other innocent process. Instead systemd will tell you:

```
Failed to kill unit typo.service: Unit typo.service
is not loaded.
```

Of course, there's more to control groups than this. The kernel supports a number of so-called *cgroup controllers* that can limit the resource usage of any given control group. This gives you the ability to restrict the amount of CPU and memory used by a service or its amount of I/O bandwidth. All of these can be set using simple statements in a unit file. Of course, you can also set ulimits as you used to (and you can also do that in unit files). But one of the big benefits of control groups over ulimits is that ulimits apply to single processes only. Control groups allow you to enforce a limit on the *aggregate* usage of all processes in a control group, no matter how many helper processes the service may spawn.

See the files under `/usr/src/linux/Documentation/croups/` in the kernel-source RPM for more details.

You can also tune some cgroup properties for specific services by using the following options in the `[Service]` section.

For example, to manage CPU usage in a cgroup:

```
CpuShares=1500
```

Or to manage I/O bandwidth:

```
BlockIOWeight=(optional path) 500

BlockIORead(Write)Bandwidth=/var/log 5M
```

Or memory (RAM) consumption:

```
MemoryLimit=1G
```

See man `5 systemd.resource-control` for a full list.

Security Features

systemd has a number of features that you can easily exploit to control resources consumption for performance reasons and to better assure security. Except as noted, you can find more information on these features in `systemd.exec(5)`.

RESTRICT SERVICES AND SESSIONS USING NAMESPACES

Linux kernel namespaces are the technology underlying Linux containers. In the case of systemd, the network namespace feature is used to limit/isolate network access for a service.

```
PrivateNetwork=yes
```

This will set up a new network namespace for the executed processes and configures only the loopback network device `lo` inside it. No other network devices will be available to the executed process.

BLACKLIST DIRECTORIES SO THAT THEY CANNOT BE ACCESSED

You can prevent a service from any access to particular directories, or restrict it to read-only access.

```
InaccessibleDirectories=/home
ReadOnlyDirectories=/var
```

REQUIRE A PRIVATE DIRECTORY FOR A SERVICE'S TEMPORARY FILES

If you don't want a service's temporary files accessed by other processes, you can force that service to use a separate, private directory for `/tmp` and `/var/tmp` without having to modify the service itself.

```
PrivateTmp=yes
```

WHITELIST ACCESS TO DEVICES IN `/dev`

By default, processes are allowed to "see" all devices under `/dev`. As always, normal UNIX/Linux permissions control what can be done with those devices. For more control, you can limit what devices are seen by coding which device nodes you want them to access and what type of permissions they have for them. See `systemd.resource-control(5)`.

```
DeviceAllow=/dev/null rw
```

SPECIFY THE USER AND GROUP TO BE USED BY A SERVICE

If you need a service to run as a specific user or group, you can easily force that in the unit file.

```
User=
Group=
```

START THE SERVICE IN A CHROOTED ENVIRONMENT

```
RootDirectory=
```

Note that If this is used, systemd will not automagically ensure that the process and all its auxiliary files are available in the `chroot()` jail. That is still the job of a human to set up ahead of time.

ASSIGN OR PROHIBIT LINUX KERNEL CAPABILITIES `(CAP_FOOBAR)`

There are a lot of kernel capabilities that can be exploited by a process. See `capabilities(7)` for the full list. Many of these capabilities require the process that invokes them to be privileged, but not all. You can grant access to those capabilities, or remove access as needed by your local requirements.

```
CapabilitiesBoundingSet=CAP_CHOWN CAP_KILL
CapabilitiesBoundingSet=~CAP_PTRACE (all but this one)
```

SET ULIMIT VALUES

As mentioned previously, specifying ulimits in a unit file has the advantage of limiting the *aggregate* resource usage of all processes in a control group. All of the same limits that are normally available can be specified.

```
LimitNPROC=1
LimitFSIZE=0
```

Useful Tips and Tricks

Does booting with "`init=/bin/bash`" still work?

Yes, it does. How you enter that is a little different, due to grub2 being the bootloader for SUSE Linux Enterprise Server 12.

- *Use the up/down arrows on the keyboard to select the kernel/initrd combination you want to use to boot.*
- *Press the "e" key on the keyboard.*
- *Use the up/down arrow keys to find the line that begins with "linux /boot/vmlinux..."*
- *Use either the left/right arrow keys or the "end" key to get to the end of that line.*
- *Add* `init=/bin/bash` *and press the F10 key, or Control-x to initiate the boot.*

How can I boot into runlevel 1 for system repair?
This is similar to the technique described for adding "`init=/bin/bash`" to the kernel command line. Simply add `systemd.unit=rescue.target` or just `1` to it.

How can I debug startup problems?
Again, similar to the technique described for adding "`init=/bin/bash`" to the kernel command line, there are keywords you can add instead:

- *systemd.log_level=debug will cause additional logging to be written to the system journal (log). Use the* `journalctl -ab` *command to examine the output.*
- *debug (The systemd developers decided to interpret this keyword as belonging to systemd in addition to the kernel.)*
- *If the problem doesn't seem to be with systemd but dracut, you can use the rd.debug parameter to get more information written to the system journal.*

For more information, you can refer to the web page at: **http://freedesktop.org/wiki/Software/systemd/Debugging/**

How do I override or supplement systemd defaults?
First, understand that you should never modify a file in `/usr/lib/systemd/system/`. Any changes you make there will be lost when systemd maintenance is installed. To avoid that, systemd has implemented a way of "stacking" and merging configuration information. This means that changes you make are safe from being overwritten and are easily identifiable.

By design, any files in `/etc/systemd/system/` take precedence over those in `/usr/lib/systemd/system/`. Therefore, you can create a copy in `/etc/systemd/system/` and modify it there. Understand that the original file in `/usr/lib/systemd/system/` will be completely ignored thereafter, so be careful of the modifications you make.

Perhaps the easiest (and safer) way is to use one or more "drop-in" files. Using the cron service as an example:

- *Create the* `/etc/systemd/system/cron.service.d/` *directory with mkdir*
- *Create files in that directory, such as* `/etc/systemd/system/cron.service.d/mychanges.conf` *containing only those parameters you want to add or override:*

```
[Service]
CPUShares=1500
BlockIOWeight=500
MemoryLimit=1G
```

In this scenario, the service file in `/usr/lib/systemd/system/` *will still be read and used, and the parameters you specified will be merged with and/or override what is there.*

In both cases, after the changes have been made, run `systemctl daemon-reload` to have them used by systemd.

How do I find out what other units are needed for a particular unit?
Using the wickedd.service as an example:

```
# systemctl show -p "Wants" wickedd.service
Wants=wickedd-nanny.service wickedd-auto4.service
wickedd-dhcp4.service wickedd-dhcp6.service system.
slice
```

If you leave off the `-p "Wants"` parameter, systemd will display all the properties it knows about a unit, which is quite a list (over 120 lines of output). Looking through that output should give you a good idea what other properties you might be interested in displaying.

How do I tell what changes from the systemd defaults have been made?
To check all changes applied (in `/etc/systemd/system`) on a system, you can use the `systemd-delta` command, which will output a diff of those changes.

Appendix A

Very Quick and Dirty Overview of Useful Commands

STARTING AND STOPPING A SERVICE

Previously, if you wanted to start or stop a service manually, you would call its init script using the corresponding verb as the parameter. For your convenience, init scripts shipped by SUSE are usually accessible via an "rc*serviceName*" command that you could use as shorthand instead of the longer `/etc/init.d/`*`serviceName`* path name.

Things aren't much different with systemd. To begin, where we supported a shorthand command `rcfoobar`, we still do so today. The difference is that it is now mapped to a call to `systemctl`, which is one of the two or three really central systemd utilities you will come across as an administrator.

Its use is pretty straightforward:

```
# systemctl start sshd.service
# systemctl stop sshd.service
# systemctl status sshd.service
```

These are really the three most important commands you'll have to remember. Note that while you *should* include the `.service` suffix, it's not 100 percent required. If you don't use the suffix, systemd will check to see if a service file exists and use it.

If you try `systemctl status`, you will notice that it is verbose in its output. Where a classic LSB script would only print a fairly terse message that would tell you whether the service is running or not, `systemctl` tries to give you all relevant information at a glance. However, once you've become used to it, you may miss it dearly when you need to go back to a SysV init based system.

Want to see more? How about this:

Enabling and Disabling a Service

```
# systemctl enable sshd.service
# systemctl disable sshd.service
```

This is how you enable a service so that it gets activated automatically on the next reboot or runlevel change. Disabling does just the reverse, as you've surely figured out.

Rebooting or Shutting Down the System

To reboot the system:

```
# systemctl reboot
```

To shut down the system:

```
# systemctl halt
# systemctl poweroff
```

Changing Runlevels

To get to runlevel 1 or S:

```
# systemctl rescue
```

To get to the default runlevel:

```
# systemctl default
```

Looking at System Logs with `journalctl`

Looking at the output from various services is quite easy with `journalctl`.

```
# journalctl -u sshd
```

Compare the output of that command with this:

```
# journalctl -x -u sshd
```

With the second, you'll notice a *lot* of additional output. This extra output is to augment log lines with explanatory text from the message catalog.

Instead of tail `-f /var/log/something` you can run

```
# journalctl -f
```

Like to see what's in the kernel ring buffer with `dmesg`?

```
# journalctl -k
```

will do that. Even better than the `dmesg` command, you can do this:

```
# journalctl -k -f
```

and see new messages as they're added to the buffer.

Appendix B

Terminology

- **Unit file**—*Encodes information about things such as a service, socket, device, mount, automount, target, snapshot, etc.*
- **Target**—*A unit configuration file whose name ends in ".target" encodes information about a target unit of systemd, which is used for grouping units and as well-known synchronization points during startup.*
- **Slice**—*A concept for hierarchically managing resources of a group of processes.*
- **Seat**—*The set of hardware available at one work place (graphics card, keyboard, mouse, usb devices).*
  - This doesn't seem to be relevant to System z
- **Session**—*A session is created once a user is logged on, using a specific seat:*
  - Only one session can be active per seat.
  - The default seat (for Linux consoles) is seat0.
- *Hardware is assigned to seats.*
  - This replaces ConsoleKit

Appendix C

Websites and Other Documentation for Reference

The man pages that are packaged with systemd are pretty verbose, which can be both good and bad. As most man pages do, they tend to assume some background you may nor may not possess. The following links can help fill in that background:

www.suse.com/documentation/sles-12/book_sle_admin/data/cha_systemd.html

www.suse.com/documentation/sles-12/pdfdoc/book_sle_admin/book_sle_admin.pdf

www.freedesktop.org/wiki/Software/systemd/
Scroll down a bit to get to some of the more interesting stuff.

http://freedesktop.org/wiki/Software/systemd/Debugging/

www.linux.com/learn/tutorials/788613-understanding-and-using-systemd

http://0pointer.de/blog/projects/systemd.html

http://0pointer.de/blog/projects/systemd-update.html

http://0pointer.de/blog/projects/systemd-update-2.html

http://0pointer.de/blog/projects/systemd-update-3.html

http://en.wikipedia.org/wiki/Systemd

http://0pointer.de/blog/projects/why.html

Appendix D

Sample LSB header for cron init script

```
### BEGIN INIT INFO
# Provides:       cron
# Required-Start: $remote_fs $syslog $time
# Should-Start:   $network smtp
# Required-Stop:  $remote_fs $syslog
# Should-Stop:    $network smtp
# Default-Start:  2 3 5
# Default-Stop:   0 1 6
# Short-Description: Cron job service
# Description:    Cron job service
### END INIT INFO
```

Contact your local SUSE Solutions Provider, or call SUSE at:

1 800 796 3700 U.S./Canada
1 801 861 4500 Worldwide

SUSE
Maxfeldstrasse 5
90409 Nuremberg
Germany

www.suse.com