INITIALIZING AND MANAGING SERVICES IN LINUX: PAST, PRESENT AND FUTURE

systemd is the new init system used by many of the top Linux distributions, but do you know the history behind it and how we got here? Learn about the history of init systems in Linux and their UNIX legacy. Gain a better perspective about how Linux manages services and other support processes.

Jonas Gorauskas

One of the most crucial pieces of any UNIX-like operating system is the init dæmon process. In Linux, this process is started by the kernel, and it's the first userspace process to spawn and the last one to die during shutdown.

During the history of UNIX and Linux, many init systems have gained popularity and then faded away. In this article, I focus on the history of the init system as it relates to Linux, and I talk about the role of init in a modern Linux system. I also relate some of the history of the System V Init (SysV) scheme, which was the de facto standard for many Linux distributions for a long time. Then I cover a couple more modern approaches to system initialization, such as Upstart and systemd. Finally, I pay some attention to how things work in systemd, as this seems to be the popular choice at the moment for several of the largest distributions.

The Role of Init

Init is short for initializer, and it's both a startup manager and a session manager for Linux and other UNIXes. It's a startup manager, because it plays a crucial role in the startup of Linux. It's the process that creates or initializes userspace and, ultimately, all userspace processes. It also may be considered a session manager, because it takes care of many aspects of userspace and its processes once the system is up and running.

The call to start this process is, in fact, hard-coded in the Linux kernel. Download the latest kernel sources and look for a function called kernel_init in the file init/main.c. Among the files that the Linux kernel will try to execute is /sbin/init. If Linux cannot find one of these processes, it throws a kernel panic and halts.

The kernel gives the init process an ID of 1 or PID 1. All other userspace processes are forked from init, and therefore, PID 1 claims ancestral rights to all other userspace processes. PID 1 also automatically will become the direct parent process of any userspace process that is orphaned.

A Little Bit of History

Now that I have set the stage for the article and given you a very basic understanding of what init is and does, I'd like to digress into a little bit of UNIX history.

There has been a lot of diversity in the initialization schemes for UNIX-like operating systems over time. Two of the most important init schemes that had a historical impact on how different Linux distributions do things are the rc scheme used in the 4.4 BSD

A Linux distribution implementing a SysV scheme can be in one of many distinct states in which a predetermined number of processes may be running.

and the SysV scheme used in SunOS and Solaris.

The 4.4 BSD init system is pretty simple and monolithic. When booting, the kernel runs /sbin/init, which would spawn a shell to run the /etc/rc script. The /etc/rc script contained commands to check the integrity of hard drives and mount them, start other processes, and start the networking subsystem. This scheme was contained completely within a few scripts: namely /etc/rc, /etc/rc.local and /etc/netstart. This scheme also had no specific shutdown procedure. Init would receive a SIGTERM signal and send a SIGHUP and/or a SIGTERM to its children, and after all processes exited, it would drop to single-user mode and shut down.

Today, the systems that have inherited the rc initialization scheme are Free-BSD, Net-BSD and the Slackware Linux distribution. These modern systems have improved quite a bit on the original 4.4 BSD scheme and are much more modular than the original.

Most other Linux distributions have, historically, been adepts of the SysV scheme, which originally was implemented in AT&T UNIX and derivative systems like Solaris.

System V Init

A Linux distribution implementing a SysV scheme can be in one of many distinct states in which a predetermined number of processes may be running. These states are called runlevels and to get to a certain runlevel means that the system is in a certain operational stage.

The meaning for each runlevel may vary based on your distribution of Linux. For example, there are a few distributions (such as Ubuntu) that use runlevel 2 to mean multi-user graphical mode with networking enabled. Others (like Fedora) use runlevel 5 to mean the same thing.

In a SysV Linux machine, the kernel runs /sbin/init as usual, which in turn will load parameters and execute

directives defined in /etc/inittab. This file defines the default runlevel for the whole system, describes what happens when Ctrl-Alt-Del is pressed, loads keymap files, defines which terminals to spawn gettys for, spawns terminal login processes, runs the /etc/init.d/rcS script, and it also influences the order of execution of other runlevel scripts.

The /etc/init.d/rcS script will put the system in a single-user mode in order to finish probing hardware, mount disks, set hostname, set up networking and so on. Take a look at /etc/rcS.d/ in a Debian 7 system for all the gory details. Next, /sbin/init will switch itself to the default runlevel to start all the system services. The default runlevel value is defined in the `initdefault` line of /etc/inittab.

This actually translates into a call to the /etc/init.d/rc script with the parameter of 2 for the runlevel value. The rc script will then execute all of the K* (for Kill) and S* (for Start) scripts in the /etc/rc2.d/ directory. These are actually links to the real scripts in /etc/init.d/. The names of the links follow the format `S##<service-name>` or `K##<service-name>`, where the ## token is the two-digit number used to determine the order in which the script should run. Order is alphabetical, and Kill scripts execute before Start scripts. The last thing to happen is to run the /etc/rc.local script, which is where you can add custom system commands that you want to execute at startup.

A system that uses the SysV scheme usually comes with the service program used to manage the services while the system is running. You can check on the status of a service, or all services, and start or stop a service, respectively, using the `service` utility:

- `$ service <service> status`
- `$ service status -all`
- `# service <service> start|stop`

To manage the assignment of services to a particular runlevel, you can use a tool called `sysv-rc-conf`, which manages the setup of all links in the respective rc directories. You also can switch the runlevel of the system at any time when you use the command `telinit` as a privileged user. For example, `telinit 6` will reboot a SysV system.

The SysV scheme still is in use today in Debian 7 (Wheezy) systems. However, the Debian developers will be changing the init system in version 8 to systemd. I cover systemd in more

The SysV scheme has been great, but it started to show its age around the time when Linux on the desktop gained a little more momentum.

detail below, but first, let's look at why we need a new init system.

The Problem with System V Init

The SysV scheme has been great, but it started to show its age around the time when Linux on the desktop gained a little more momentum. When the SysV scheme originally was designed, computers where nothing like they are today. SysV was not designed to handle certain things well:

- USB devices.
- External storage volumes.
- Bluetooth devices.
- The cloud.

The SysV scheme was designed for a world that was static and slow moving. This init scheme originally was responsible only for bringing the system into a normal running state after power on or gracefully shutting down services prior to shutdown. As a result, the design was strictly synchronous, blocking future tasks until the current one had completed.

This left the system unable to handle various events that were not related to the startup or shutdown of the system. Things that we take for granted today were really cumbersome to handle elegantly during the heyday of SysV init:

- There was no real process supervision—for example, dæmons were not automatically restarted when they crashed.
- There was no real dependency checking. The order of script naming determined the order in which they were loaded.
- The addition or removal of USB drives and other portable storage/network devices while the machine was running was cumbersome and oftentimes required a reboot.

- There were no facilities to discover and scan for new storage devices without locking the system, especially when a disk might not even power on until it was scanned.

- There were no facilities to load firmware for a device, which may have needed to occur after it was detected but before it was usable.

Inevitably, around the 2005/2006 time frame, several alternative efforts tried to fix all the issues with the SysV scheme. But the effort that looked most promising during that time was the Upstart init project sponsored by Canonical.

Upstart

To be sure, Upstart init doesn't share any code with the SysV init scheme, but it's rather a superset of it, providing a good degree of backward-compatibility. The main departure from the traditional SysV way of doing things is that Upstart implements an event-driven model that allows it to respond to milestones asynchronously as they are reached. Upstart also implements the concept of jobs, which are described by the files under /etc/init/*.conf, and whose purpose is to execute a script section that spawns a process. As such, system initialization can be expressed as a consecutive set of "spawn process X when event Y occurs" rules.

Just like in the SysV scheme, the Linux kernel gives control to Upstart when it executes the Upstart implementation of /sbin/init. At this point, things may work a little differently depending on your distribution of Linux. For Red Hat Enterprise Linux (RHEL) 6 users, you'll still find a file at /etc/inittab, but the sole function of this file is to set the default runlevel for the system. If your distribution is one of the Ubuntu derivatives, /etc/inittab doesn't even exist anymore, and the default runlevel is set in a file called /etc/init/rc-sysinit.conf instead.

The Upstart version of /sbin/init will emit a single event called startup, which triggers the rest of the system initialization. There are a few jobs that specify the startup event as their start condition, the most notable of which is `mountall`, which mounts all filesystems. The `mountall` job then triggers various other events related to disk and filesystem initialization. These events, in turn, trigger the udev kernel device manager to start, and it emits the event that starts the networking subsystem.

This is when one of the most critical

jobs is triggered by Upstart. This job is called `rc-sysinit`, which has a start dependency on the filesystem and network-up events. The role of this job is to bring the system to its default runlevel. It executes the command `telinit <runlevel>` to achieve this. The `telinit` command then emits the runlevel event, which causes many other jobs to start. This includes the /etc/init/rc.conf job, which implements a compatibility layer for the SysV init scheme. It executes `/etc/init.d/rc <runlevel>` and determines if a /etc/rc#.d/ directory exists for the current runlevel, executing all scripts in it.

In Upstart-based systems, such as Ubuntu and RHEL 6, you can use the tools `sysv-rc-conf` or `chkconfig`, respectively, to manage the runlevel of different services. You also can manage jobs via the `initctl` utility. You can list all jobs and their respective start and stop events with the command `initctl show-config`. You also can check on job status, list available jobs and start/stop jobs with the following commands, respectively:

- `$ initctl status <job>`
- `$ initctl list`
- `# initctl start|stop <job>`

The Upstart scheme has been used in popular distributions of Linux, such as Fedora from versions 9 up to 14, the RHEL 6 series and Ubuntu since version 6.10 to present. But for all the flexibility that Upstart init brings to Linux, it still falls short in a few fundamental ways:

- It ignores the system state between events. For instance, a system has a power cord plugged in, then the system runs on AC power for a while, and then the user unplugs the power cord. Upstart focuses on each event above as a single discrete and unrelated unit, instead of tracking the chain of events as a whole.

- The event-driven nature of the system turns the dependency chain on its head. Instead of doing the absolute minimum amount of work needed to get the system to a working state, when an event is triggered, it executes all jobs that could possibly follow it. For example, just because networking has started, it doesn't mean that NFS also should start. As a matter of fact, the opposite is the correct order of things: when a user requests access to an NFS share, the system should validate that networking is also up and running.

The main design goals of this init scheme are, according to Lennart Poettering, lead developer of systemd, "to start less, and to start more in parallel".

- The dependency chain is still present. Although many more things happen in parallel in Upstart, the user has to port the original script sequence from SysV init to a set of event trigger action rules in the *.conf files in /etc/init. Furthermore, because of the spanning tree structure of the event system, it is a real nightmare to figure out why something happened and what event triggered it.

There is another init scheme whose purpose is to address the issues listed above.

systemd

systemd is the latest milestone on the road to init system nirvana. The main design goals of this init scheme are, according to Lennart Poettering, lead developer of systemd, "to start less, and to start more in parallel". What that means is that you execute only that which is absolutely necessary to get the system to a running state, and you run as much as possible at the same time.

To accomplish these goals, systemd aims to act against two major trouble spots of previous init schemes: the shell and parallelism. The main executable for systemd, /lib/systemd/systemd, performs all calls that originally were present in scripts, thus eliminating the need to spawn a shell environment. What about the call to /sbin/init that's hard-coded in the Linux kernel? It's still there in the form of a symbolic link to /lib/systemd/systemd.

To address parallelism, you need to remove the dependency chain between the various services or at least make it a secondary concern. If you look at the problem at its most fundamental level, the dependency between the various services boils down to one thing: having a socket available for the processes to communicate among themselves. systemd creates all sockets first and then spawns all processes in parallel. For example, services that need to write to the system log need to wait for the

/dev/log socket to become available, but as soon as it is available, these services can start. Therefore, if systemd creates the socket /dev/log first, then that's one less dependency that blocks other services. Even if there is nothing to receive messages at the other end of the socket, this strategy still works. The kernel itself will manage a buffer for the socket, and as soon as the receiving service starts, it will flush the buffer and handle all the messages. The ideas above are not new or revolutionary. They have been tried before in projects like the xinetd superserver and the launchd init scheme used in OS X.

systemd does introduce the new concepts of units and targets. A target is analogous to a runlevel in previous schemes and is composed of several units. systemd will execute units to reach a target. The instructions for each unit reside in the /lib/systemd/system/ directory. These files use a declarative format that looks like a Windows INI file. The most common type of these units is the service unit, which is used to start a service. The sshd.service file from Arch Linux looks like this:

```
[Unit]
Description=OpenSSH Daemon
Wants=sshdgenkeys.service
After=sshdgenkeys.service
After=network.target

[Service]
ExecStart=/usr/bin/sshd -D
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=always

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

This format is really simple and really portable across several different distributions. There are other types of unit files that describe a system, and they are socket, device, mount, automount, swap, target, path, timer, snapshot, slice and scope. Going into all of them in detail is beyond the scope of this article; however, I want to mention one thing: target is a special type of unit file that glues the other types together into a cohesive whole. For example, here are the contents of basic.target from Arch Linux:

```
[Unit]
Description=Basic System
Documentation=man:systemd.special(7)
Requires=sysinit.target
Wants=sockets.target timers.target paths.target
 ➥slices.target
After=sysinit.target sockets.target timers.target
 ➥paths.target slices.target
JobTimeoutSec=15min
JobTimeoutAction=poweroff-force
```

You can follow the chain of dependencies if you look at what basic.target requires and wants. Those are actual unit files in the same /lib/systemd/system/ directory. The `Requires` and `Wants` directives above are how systemd defines the dependency chain among the units. The `Requires` directive denotes a hard requirement, and `Wants` denotes an optional requirement. Also keep in mind that `Requires` and `Wants` don't imply order. If the `After` directive isn't specified, systemd will start the units in parallel.

Timer units are also really interesting. They are unit files that contain a `[Timer]` section and define how the `TimeDateD` subsystem of systemd will activate a future event. In these timer units, you can create two types of timers: one that will activate after a time period based on a variable starting point, such as the systems boot, and another that activates at fixed intervals like a cron job. As a matter of fact, timer units are an alternative to cron jobs.

One last thing to mention about systemd unit files is that they provide the means to describe easily what to do when a service crashes. You can do that by using the directives `Restart` or `RestartSec` in your unit files. This feature allows systemd to take the role of process supervisor as well.

systemd refers to the init dæmon executable itself, namely /lib/systemd/systemd, but it also refers to the set of utilities and programs used to manage the system and services. Chief among these utilities is the `systemctl` program that's used to manage services. You can use it to enable, start and disable services, find the status of a given service and also list all loaded units. For example:

- `# systemctl enable sshd`
- `# systemctl start sshd`
- `# systemctl stop sshd`
- `# systemctl status sshd`
- `# systemctl list-units`

Some Linux distributions, like RHEL 7 and CentOS 7, provide a compatibility layer that translates SysV and Upstart commands into systemd commands. If you issue the command `service sshd status` in CentOS 7, you will get the following output:

```
Redirecting to /bin/systemctl status  sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Mon 2014-12-08 02:01:53 PST;
➥12h ago
  Process: 915 ExecStartPre=/usr/sbin/sshd-keygen (code=exited,
➥status=0/SUCCESS)
     Main PID: 937 (sshd)
       CGroup: /system.slice/sshd.service
               ...937 /usr/sbin/sshd -D
```

Notice that first line of console output above and how it indicates that the SysV-style command was redirected to the systemd-style of command. This allows the user to ease into the systemd way of doing things while still allowing the user to leverage the previous skill set.

Another really important program in the systemd toolbox is the `journalctl` utility. It allows you to view and manage the systemd logging subsystem called journald. systemd's logfile is a binary file and using `journalctl` really simplifies the user experience. Here are some interesting examples:

- Display full log: `# journalctl --all`
- Tail the log: `# journalctl -f`
- Filter log by executable:
  `# journalctl /lib/systemd/systemd`
- Display log since last boot:
  `# journalctl -b`
- Display errors from last boot:
  `# journalctl -b -p err`

I urge you to look at the documentation of the different schemes presented here to learn more.

Controversies

From my vantage point, the future is not 100% certain when it comes to init schemes for Linux. The clear leader, as I write this in late 2014, is systemd. A lot of distributions are adopting it; the latest ones are RHEL 7 and Debian 8. However, the adoption of systemd has been controversial, and these distributions have received a lot of strong feedback from their respective communities. Of note is the Debian technical committee debate that occurred in the Debian mailing list and a complaint by Linus Torvalds himself in the Linux kernel mailing list.

systemd is not just an init scheme. It unifies everything that is related to starting and managing system services into a centralized and monolithic whole: user login, cron jobs, network services, virtual TTY management and so on. The use of shell scripts to control system startup has the benefit of providing flexibility, and a lot of

members of the community want to be able to choose their favorite init scheme. This has spawned some forks of systemd and even a faction of the Linux community that is for completely boycotting systemd. Check out the site http://boycottsystemd.org.

Conclusion

The userspace initialization and management of Linux systems has had a rich and diverse history. I hope that this article has given you a new perspective for how we got to where we are today with systemd becoming the new standard. I have covered all the pros and cons of the different schemes and how those factors have influenced user choice in this space over time. I hope this article will foster further discussion, and your feedback is highly encouraged.■

Jonas Gorauskas is technically a software developer by trade but also a generalist with background in operations. In the past he has been one or more of the following: programmer, technical support analyst, technical writer, systems designer, database administrator, amateur cook and professional curmudgeon. Jonas is currently working at Intuit in Reno, Nevada, as part of the Application Operations Engineering team helping them with operations, deployment, DevOps or anything else they can think of.

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.

Resources

The source code of various Linux distributions, including:

- Debian 7 and 8
- CentOS 6.5 and 7
- Slackware 14
- Fedora 20
- Ubuntu 12.4 and 14.4
- Arch Linux

The Web site of Lennart Poettering: **http://0pointer.net/blog**

The systemd Documentation: **http://freedesktop.org/wiki/Software/systemd**

Upstart Documentation: **http://upstart.ubuntu.com/cookbook**