Customizing and monitoring Linux system startup

Compare different Linux startup mechanisms

William von Hagen February 19, 2014

Minimizing the amount of time required to boot a computer system is important regardless of whether you are turning on your home computer or restarting a server that provides services to thousands of users. This article discusses the various system startup and shutdown mechanisms that are used on different Linux® distributions. It explains how to integrate new services, customize existing startup configurations, and examine the behavior and performance of system startup configurations.

Overview of the Linux system startup process

You can think of the Linux system startup process, from powering on to having a fully running system, as two conceptual phases:

1. Booting from a device and loading and initializing the Linux kernel
2. Starting user-space applications (including server processes), mounting additional file systems, performing additional kernel configuration and customization, and providing access to additional devices

The basic steps in the first phase were described in an earlier developerWorks article entitled "Inside the Linux Boot Process" (see Related topics). Those steps haven't changed much for years, aside from using GRand Unified Bootloader 2 (GRUB 2), differences in the format and use of an initial RAM disk, and changes to the first user-space process that is started (traditionally, the `init` process). Performance improvements in this phase are primarily due to hardware improvements, such as faster boot devices, faster device access mechanisms, and faster and more powerful processors. However, performance improvements in the second phase are almost completely software-side, where work continues using a number of different approaches.

An easy way of reducing phase 2 system startup time is by improving the performance of the mechanism used to do the second phase of the Linux startup process. However, far bigger improvements are possible by changing the startup sequence itself. Key areas for reducing phase 2 system startup time are the following:

- **Minimization**, where only the smallest set of required services are started

- **Linearity**, which means understanding the requirements of one service on one or more others (also known as *dependency analysis*) and taking those requirements into account during the startup process
- **Parallelism**, which means maximizing the extent to which independent services (or chains of related services) can be started at the same time

Throughout its history, Linux has used a variety of startup and shutdown mechanisms, working to balance linearity and parallelism without completely disrupting backwards compatibility. The next few sections discuss the most common of these and how to monitor their behavior and performance on your systems to help identify optimization opportunities.

Understanding and using SysVinit

The traditional system startup and shutdown mechanism used on Linux systems is known as SysVinit. As the name suggests, the SysVinit mechanism has its conceptual roots in the Sys V version of UNIX®, more properly UNIX System V, Release 4 (SVR4), which was released in 1989. The SysVinit init program reads the /etc/inittab file to identify the collection of services that should be available when the system reaches its default state (known as the system's default *runlevel*) and the command that it should execute to reach that state. Systems that use SysVinit define this information using entries in the /etc/inittab file, as shown in Listing 1.

Listing 1. Traditional startup-related commands in /etc/inittab

```
si::sysinit:/etc/init.d/rcS
id:5:initdefault:
l5:5:wait:/etc/init.d/rc 5
```

The first line identifies the first script that the system should execute when the system is being initialized. The second line identifies the system's default runlevel after initialization. In this example, the default runlevel is 5, which typically means a system with full networking and graphical capabilities. The third line identifies the command that the system should use to reach runlevel 5.

When using SysVinit, the /etc/init.d directory contains shell scripts that start and stop all system-level processes. Each runlevel has its own directory that contains symbolic links to the selected shell scripts in the /etc/init.d directory that should be started or stopped when entering or exiting the associated runlevel. The third line in Listing 1 tells the SysVinit mechanism to execute the scripts in the directory that is associated with runlevel 5, which is typically /etc/rc5.d, /etc/init.d/rc5.d, or /etc/rc.d/rc5.d, depending on the Linux distribution that you are running.

The symbolic links in a runlevel directory begin with either an S or a K, indicating whether they should be executed to start (S) the associated system process when a system is entering the specified runlevel or terminate it (K) when a system is leaving that runlevel. The integer value that follows the S or K defines the order in which these scripts should be executed when entering or leaving that runlevel.

When adding new scripts to the /etc/init.d directory, specially formatted comments at the beginning of each script identify any other scripts that the script depends on and also identify each runlevel

that the script should be associated with. The commands and other information that should be present in these scripts are defined as part of the Linux Standard Base (LSB) specification, which is a standard that was developed by multiple Linux distributions to ensure compatibility of SysVinit scripts across different Linux distributions. Listing 2, taken from the LSB discussion of comments in init scripts, shows an example of this section.

Listing 2. Comment block in a traditional SysVinit script

```
### BEGIN INIT INFO
# Provides: lsb-ourdb
# Required-Start: $local_fs $network $remote_fs
# Required-Stop: $local_fs $network $remote_fs
# Default-Start:  2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: start and stop OurDB
# Description: OurDB is a very fast and reliable database
# engine used for illustrating init scripts
### END INIT INFO
```

Table 1 explains the information that each of these entries provides.

Table 1. Comments in the INIT INFO section of a SysVinit script

Keyword	Meaning
`Provides`	Logical name for the service that this initialization script provides.
`Required-Start`	Logical names of any other services that must be running to successfully start the service that is defined in this initialization script.
`Required-Stop`	Logical names of any other services that must be running to successfully stop the service that is defined in this initialization script.
`Default-Start`	Runlevels in which this initialization script should be executed to start the service that it defines.
`Default-Stop`	Runlevels in which this initialization script should be executed to stop the service that it defines.
`Short-Description` and `Description`	Provide short and more verbose descriptions of the service that is associated with the commands in an initialization script. These are used by various system utilities that query and summarize SysVinit initialization scripts.

You can start, stop, and list SysVinit scripts using the `/sbin/service` command. You can list or modify the runlevels that they are associated with using the `/sbin/chkconfig` command.

Using shell scripts for system startup makes it easy to add new commands to a system's startup process or to modify the actions that occur when starting or stopping a service. You can easily add new services at a specific point in the boot sequence simply by creating symbolic links to them with an appropriate number in the name of the symbolic link.

From a performance perspective, SysVinit executes multiple shell scripts in a specified order to reach a given runlevel. Executing shell scripts is relatively slow and provides an inherently sequential startup mechanism that can extend boot times because there is no way to take advantage of potential parallelism. Each shell script that is associated with the target runlevel must be executed in the order specified by the number in the name of the symbolic link that points to it,

and startup of another service cannot begin until the current one completes. Therefore, systems that use SysVinit can benefit the most from integrating the startup monitoring and profiling tools discussed in the last section of this article, "Monitoring system startup and script execution," to determine exactly how long each startup script is taking.

Understanding and using event-driven startup mechanisms

While SysVinit is easy to work with and modify because of its use of shell scripts to start and stop system services, using sequential shell scripts also makes SysVinit inherently slow and provides no opportunities for starting unrelated services in parallel. Another problem is that the scripts that start and stop services are only executed at system startup or shutdown—in other words, the only event that they respond to is when system runlevels change. With few exceptions (such as services that are started by other services), this means that all of the services that may be required on a system must always be running and waiting for requests that may never come.

Running unused processes is inefficient both in terms of memory and processor resource consumption and in terms of the time that it takes to start them in the first place. Today's increasingly flexible systems need to be able to function correctly and seamlessly in a large number of different situations, such as after changes in network connectivity from wired to wireless, migration from one network to another, or hardware changes such as adding and removing storage devices and other peripherals.

Enter event-driven startup mechanisms, which work exactly as their name suggests, executing specific commands and starting associated services when certain events occur on a system. The `inetd` and `xinetd` daemons for various network services provide good analogies to event-driven startup mechanisms because they simply wait for certain events to occur (network connection requests for the services that they manage) and then start the appropriate service as required. Event-driven startup mechanisms maximize parallelism during system startup by enabling multiple commands to execute at the same time in response to an event and extend that same dynamic responsiveness to the system's runtime environment. Events are essentially string messages that a process can send in response to a change in the state of something that the process is monitoring. Event messages are sent once by a process.

The best known event-driven startup mechanism is Upstart, which is the default startup mechanism used on Linux distributions such as Ubuntu 9.10 and later, RHEL6 and related distributions such as CentOS, Oracle Linux, and Scientific Linux, Fedora 9-14, and many other Linux distributions. Upstart provides backwards compatibility with the SysVinit runlevel model.

Upstart uses files known as job configuration (or conf) files to identify its response to events such as startup, shutdown, runlevel (runlevel transitions), and so on. These files are usually located in the /etc/init directory, though some are located in the /etc/event.d directory for legacy reasons. Any new system-wide conf files that you create should be located in the /etc/init directory. Conf files typically have a .conf extension. They are text files that must contain at least the following:

- One or more events in response to which that conf file should perform some action. For example, a `start on startup` entry states that a job file should be executed when a `startup`

event is received, and a `stop on runlevel` entry states that a job file should stop whenever a `runlevel` event is received, such as when a system's runlevel changes.
- A `task` or `respawn` section that contains at least an `exec` entry or `script` section that identifies the commands to be executed in response to the events that this job file is triggered by. An `exec` entry executes a specific command, usually a binary, with a specific set of command-line arguments. A `script` section, known as a *stanza*, provides commands that are executed by the shell, like any shell script, and must end with an `end script` statement. Upstart manages two conceptual classes of jobs via the `task` and `respawn` keywords:
  - Tasks must complete, meaning they must transition from the stopped to the started state and then return to the stopped state after completion.
  - Services must always be running and, therefore, simply transition from stopped to started. The `respawn` section identifies both how to start and restart a service.

You can also use other keywords in the `task` section of Upstart conf files to identify output devices, scripts to run before the primary `exec` or `script` section is executed, scripts to run after the primary `exec` or `script` sections complete, and so on. A `pre-start script` section provides commands that are used to initialize the environment required for `script` or `exec` commands, while a `post-stop script` section provides commands that are used to clean up or perform post-processing after an `exec` command or `script` stanza completes. Many other Upstart commands are also available in job files (see Related topics).

For example, Listing 3 is the /etc/init/rcS.conf Upstart job file, which emulates the SysVinit mechanism. I removed comments to improve readability.

Listing 3. The /etc/init/rcS.conf Upstart job file

```
start on runlevel [0123456]

stop on runlevel [!$RUNLEVEL]

task

export RUNLEVEL
console output
exec /etc/rc.d/rc $RUNLEVEL
```

This conf file defines a task that runs any existing SysVinit scripts from the directory that is associated with the current runlevel.

You can start, stop, and list Upstart jobs using the `/sbin/initctl` command. This command shows the Upstart jobs that are currently running and their current state. Adding a new job to a system's initialization sequence simply requires creating an appropriate conf file for that service and installing that file in the /etc/init directory. Upstart versions 1.3 and greater even support user-specific conf files that are located in the .init subdirectory of a user's home directory, but this option is not commonly enabled.

Understanding and using `systemd`

Just as Upstart seemed destined to purge SysVinit from Linux systems everywhere, another, even brighter, shinier, and more efficient startup mechanism appeared: `systemd` (system daemon),

initially written by Leonard Poettering. The `systemd` system startup mechanism provides significant improvements in parallelizing system startup by understanding the underlying resources that various services require and use. `systemd` also makes it easier to track and manage resources for related processes by using the control groups (cgroups) mechanism that has been supported in the Linux kernel since later versions of the 2.6 kernel.

Most modern Linux services and associated clients use UNIX sockets for inter-process communication, including the D-Bus message bus that is used for general hardware-related and local inter-application messages. When the required sockets exist or the D-Bus has been activated, any services that use them can be started, so `systemd` first creates all of the sockets that are associated with the services that you want to start on a given system. The services that use these resources typically block until they need to deliver or receive a message, so they can either be started in parallel or started on demand based on incoming requests.

Creating logical resources to be able to start services that may use them in parallel is not limited to just clients and servers. Even traditionally slow portions of the system startup process, such as file system consistency checking and mounting, can use this model for file systems other than the root file system (which everybody always needs) when combined with on-demand file system access mechanisms such as `autofs`. After a file system is mounted, you can use file or file system change events to trigger other actions using the kernel's `fanotify` and `fsnotify` mechanisms.

The `systemd` startup mechanism refers to anything that it manages as a *unit*. To satisfy different types of initialization and startup requirements, `systemd` supports different types of units, the most common of which are explained in Table 2.

Table 2. Common unit types for `systemd`

Unit type	Explanation
`automount`	Identifies a file system mount point used by an on-demand file system access mechanism such as `autofs`. Each `automount` unit has a matching `mount` unit.
`device`	Represents a physical device for which a `udev` rule exists.
`mount`	Identifies a standard file system mount point.
`service`	Defines a daemon that can be started, stopped, restarted, reloaded, and so on. Because these are the traditional types of things that are done by the SysVinit startup mechanism, `systemd` can automatically parse the LSB comments in SysVinit startup scripts (discussed in Understanding and using SysVinit) and use that information appropriately.
`socket`	Represents a file system or network socket of standard types such as `AF_INET` or `AF_UNIX` so that incoming connections to those sockets can trigger starting an associated service
`target`	Defines a logical unit that is used to group other units that are conceptually related. As an example, `systemd` uses the `graphical.target` unit to collect all of the applications that should be started on a system with a graphics console when the graphics device becomes available.

The service configuration files that are associated with the unit types that are supported by `systemd` are typically located in subdirectories of /etc/systemd. Listing 4 is a sample `systemd` service configuration file.

Listing 4. A sample `systemd` service configuration file

```
[Unit]
Description=System Logging Service

[Service]
EnvironmentFile=-/etc/sysconfig/rsyslog
ExecStart=/sbin/rsyslogd -n $SYSLOGD_OPTIONS
Sockets=syslog.socket
StandardOutput=null

[Install]
WantedBy=multi-user.target
Alias=syslog.service
```

This file is the unit definition for the syslog from a Fedora 17 system `rsyslog.service` and is located in the /etc/systemd/system/multi-user.target.wants directory. As you can see, the different sections of this file provide a description of the unit with which it is associated, identify how to start the service and what resources it uses, and identify the circumstances under which the unit is invoked—in this case, when the system's startup target is the `multi-user.target`.

The `systemd` startup mechanism uses the `systemctl` command for starting, stopping, and checking the status of units of various types from the command line. The `systemadm` command provides a graphical interface for the same capabilities but is not installed by default. To add it, you must install the `systemd-gtk` package.

The `systemd` startup mechanism can provide substantial performance improvements but can also be trickier to work with if you need to add commands at a specific, deterministic point in the startup sequence. While `systemd` is well worth testing to see if it improves startup performance on specific systems, not everyone is a fan because `systemd` is significantly different from previous Linux startup mechanisms. See Related topics for a link to a contrarian perspective on `systemd`. New approaches are inherently interesting but can also tarnish quickly.

Monitoring system startup and script execution

If you are trying to minimize the time required for a system to become available, simply measuring the amount of time that it takes for a system to become available after powering on doesn't provide much useful information. The real data that you're interested in are things such as the following:

- Which initialization commands or scripts were executed during system startup?
- In what order were commands or scripts executed?
- How long did each command or script take?

If you are just adding a single script to a system's startup process, your initial concerns are that it is actually being executed and at what point it is being executed. An easy way to determine that information is to invoke the `/usr/bin/logger` command from within your script. The `logger`

command writes a message to the system log with a specified priority. For example, the following command writes the message `New script executed` with the numeric priority that corresponds to `alert`, which should always be logged:

```
/usr/bin/logger -p 1 "New script executed"
```

Verifying that your new command executed is a good thing to do, but it is less useful than being able to examine the relative performance of all of your startup scripts, the time that each takes, and so on. An extremely handy utility that provides this sort of information in a graphical format is Bootchart.

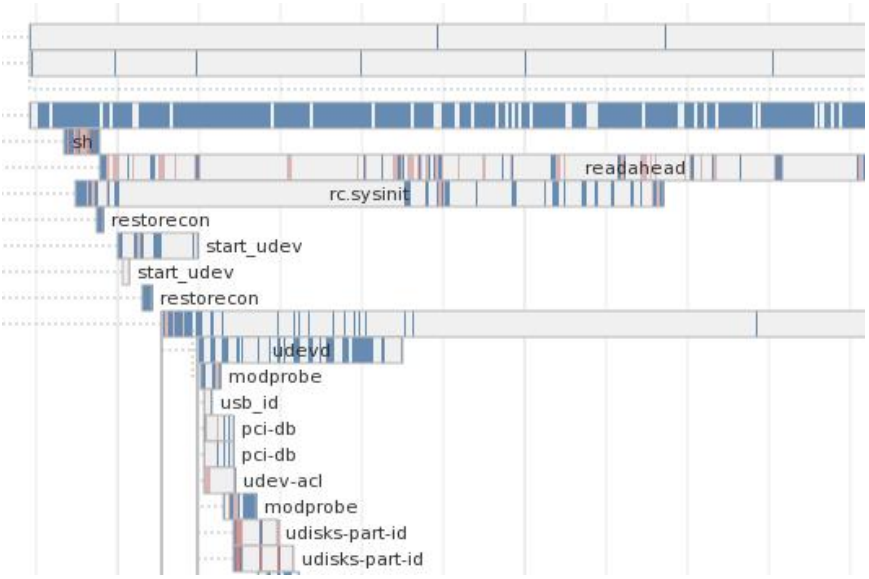
Bootchart was originally written as a shell script, but the latest version, Bootchart 2, was rewritten in the C programming language to provide higher performance and lessen the impact of the application itself on the data that it is collecting. You can use the latest version of Bootchart 2 by retrieving the source code for the current version, building and installing the application, and then integrating it into the bootloader commands that you use to boot your system. To build and install Bootchart 2, the following must be installed on your system: `git` source code control system, `make` application compilation tool, and `gcc` C compiler. You can then do the following:

1. Use the `git` command to clone the source code repository for Bootchart 2:
   ```
   git clone https://github.com/mmeeks/bootchart
   ```
2. Change directory to the bootchart directory that was created in the previous step and execute the `make` command to compile the various portions of Bootchart 2.
3. As the root user or via the `sudo` command, execute the `make install` command to install different Bootchart 2 applications and their documentation in their default locations.
4. Add the following fragment to the end of the kernel entry in the bootloader configuration file that you are using:
   ```
   initcall_debug printk.time=y quiet init=/sbin/bootchartd
   ```

   If you are using the GRUB bootloader, you will probably add the preceding text fragment to a boot stanza in the /boot/grub/menu.lst file. If you are using the GRUB2 bootloader, you will probably add the preceding text fragment to a boot stanza in the /boot/grub/grub.conf file. If the GRUB configuration file on your system is autogenerated using commands such as `grub-mkconfig` or `grub2-mkconfig`, you should add the preceding text fragment to the default kernel boot options that are specified in the /etc/default/grub file, and then regenerate your actual bootloader configuration file.

The next time that you reboot your system, Bootchart 2 will collect data about each stage in the boot process and generate a graphical summary of that data in Portable Network Graphics (PNG) format. The summary data is saved in the /var/log/bootchart.tgz file. The associated graphical representation of that data is saved in the /var/log/bootchart.png file. Figure 1 is an excerpt of a graphical representation of the boot process.

Figure 1. Excerpt of Bootchart's graphical boot representation



The top portion of Bootchart 2's graphical view of the boot processes summarizes information about the system on which the boot information was collected, including the total time for the startup process. The middle section provides a graphical representation of all startup processes (from which Figure 1 is an excerpt). The bottom two sections show cumulative processor usage by process and cumulative I/O usage by process.

Each time you reboot your system, any previous files with these names are overwritten. If you are experimenting with ways of modifying the order in which your system executes various scripts during startup, trying to optimize their content, or both, you probably want to keep the summary Bootchart 2 data from multiple system boots to make it easier to compare those results.

Bootchart 2 provides the `CUSTOM_POST_CMD` variable in its configuration file (/etc/bootchartd.conf). This variable enables you to specify the full path to a script or other application that executes after Bootchart 2 has created its data and associated graphics files. Listing 5 is an example script that you can use to rename your output files with names of the form YYYY-MM-DD-HH-MM-HOSTNAME, saving them in the /var/log/bootchart directory.

Listing 5. Sample script to uniquely rename Bootchart output files

```
#!/bin/bash

source /etc/bootchartd.conf

HOST=$(hostname -s)

if [ ! -d $(dirname $BOOTLOG_DEST)"/bootchart" ] ; then
    mkdir $(dirname $BOOTLOG_DEST)"/bootchart"
fi

DATE=$(date +%Y-%m-%d-%H-%M)

filebase="$DATE-$HOST"

mv $BOOTLOG_DEST $(dirname $BOOTLOG_DEST)"/bootchart/"${filebase}".tgz"

if [ "x$AUTO_RENDER" = "xyes" ] ; then
    mv ${AUTO_RENDER_DIR}/bootchart.${AUTO_RENDER_FORMAT} \
       $(dirname $BOOTLOG_DEST)"/bootchart/"${filebase}"."${AUTO_RENDER_FORMAT}
fi
```

Bootchart 2 works with all of the Linux system startup mechanisms described in this article and provides excellent insights into how much time, processor, and I/O each step in the system startup sequence takes. This helps identify portions of the startup sequence that you should try to optimize to minimize system startup time.

Summary

Linux offers many startup mechanisms, some that are easy to modify and others that are extensible but primarily designed for speed. This article summarized the three most popular Linux startup mechanisms, highlighting differences in their goals and implementations. Different Linux distributions use different startup mechanisms out of the box, but it is easy to install and experiment with different ones until you find the combination of performance, flexibility, and usability that best suits your requirements.

Related topics

- "Inside the Linux boot process" (developerWorks, May 2006) provides a great introduction to the basic stages of booting a Linux system.
- The LSB specification is a standard that was developed by multiple Linux distributions to ensure compatibility of SysVinit scripts across different Linux distributions.
- Init Script Actions defines the standard commands that should be available in LSB-compliant SysVinit scripts for LSB version 4.1, the current version when this article was written.
- Comment Conventions for Init Scripts defines the comments that should be present in LSB-compliant SysVinit scripts for LSB version 4.1 to identify dependencies and the runlevels with which a compliant initialization script should be associated.
- Upstart Intro, Cookbook and Best Practises is Ubuntu's official documentation on why and how best to use Upstart and create Upstart conf files.
- "Rethinking Pid 1" (Apr, 2010) is a great introduction to the goals and implementation of `systemd`, written by its original developer, Lennart Poettering.
- "Why systemd?" (Apr, 2010) compares the capabilities of SysVinit, Upstart, and `systemd`. Readers should note that this comparison was written by the original developer of `systemd`, Lennart Poettering, and its points of comparison may show some selection bias toward the capabilities of `systemd`.
- "systemd for Administrators, Part 1" (Aug, 2010) is the first in a series of `systemd` tutorials by Lennart Poettering for system administrators.
- The Bootchart 2 Project Summary on Ohloh provides a summary of the goals and changes in the latest version of Bootchart.
- Get the latest version of Bootchart 2 from the project's page on GitHub.