Initialization sequence in GNU/Linux

The process of booting your PC, from power to prompt

Steven Goodwin

The login prompt is a nice place to be. Poised, fingers on keyboards, ready to send mail, surf the web, or do a little programming. However, from power-on to login prompt there is a long road for our GNU-powered friend to travel.

The boot up

The first step on this journey is the BIOS power on, system test. Or POST for short. BIOS stands for Basic Input Output System, and is the custom chip on the motherboard that holds a small program to kickstart the whole boot-up. Its job is to check the low-level hardware components, such as memory and hard disks. Configuration of the BIOS can be a complex task, and is certainly an article in itself. If you're curious, hold down the delete key during boot-up to see the BIOS configuration screen. Control is passed from here to the boot loader.

The boot loader is a very small program that lives on the hard drive. It is not stored as a file, since that requires a file system—and one doesn't exist yet since we haven't loaded an operating system. So instead, it is stored in the very first 512 bytes of the disk (also known as the master boot record, or MBR). The BIOS only needs to know the physical layout of the hard drive (which it obtains from the POST) to load this program into memory, and execute it.

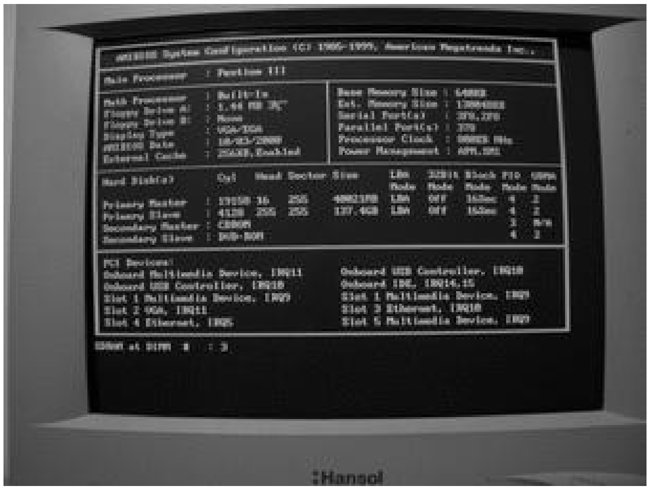
The whole boot process follows this pattern of small incremental steps; each doing a specific job, and passing control to the next program. The term "boot" stems from the phrase "bootstrapping", meaning to lift yourself up by your bootstraps.

The beginning of the POST

The term "boot" stems from the phrase "bootstrapping", meaning to lift yourself up by your bootstraps

The boot loader program itself is operating system-independent. Indeed, one boot loader can provide you with access to several OSes, often through a basic menu screen, facilitating a dual-boot machine. Most GNU/Linux distri-

Typical configuration data discovered by the BIOS



butions ship with a choice of two boot loaders, Lilo and GRUB. Lilo, for example, uses `/etc/lilo.conf` to determine which kernel, or kernels, can be loaded at boot time. This file can also be used to pass options from Lilo into the kernel. In order to create or change the Lilo boot loader you need to edit `/etc/lilo.conf` and run the `/sbin/lilo` program. This takes the loader and places it directly into the MBR. The most common change of this kind occurs after compiling a new kernel, as the boot loader needs to know where the new kernel lives. Since 512 bytes doesn't provide much room for code, most boot loaders are (transparently) loaded in two parts. The next evident activity comes from the kernel, in this case, Linux.

Linux, regardless of version, will always issue a flood of messages to the screen as it runs. These describe the computer system, as the kernel sees it, to aid troubleshooting. The messages themselves are also written into a log file at `/var/log/kern.log`, or can be retrieved using the `dmesg` command.

The kernel doesn't have any programs to manage yet, so it does little more than initialize itself. This includes preparation of the all-important file system. Most file systems will be built into the kernel

directly, but for some drivers (such as SCSI) these will be loaded into the RAM disk.

The RAM disk (or `initdisk` or `initrd`) is a memory-based file system that is created by the boot loader before Linux is run. A more exotic system configuration can be created using the `/linuxrc` file to load other drivers, or mount additional components into the file system. Linuxrc may be either a program, or a script. However, the latter would require a shell interpreter in order to run, and is thus rarer.

The Linux kernel, as its final act during boot-up, runs `init`. This is a program normally stored as `/sbin/init`.

Alternate locations for `init`

By default, `init` will live in `/sbin/init`, and is the first place the kernel will look, followed by `/etc/init` and `/bin/init`, in that order. Should all of these fail, a shell (`/bin/sh`) will be executed instead, enabling you to recover the system. You can also specify a different location for `init` by passing it as an argument to the kernel. Lilo users, for example, can include an `append` line in `lilo.conf` to use a completely different program. This line can also specify a runlevel that will take precedence over the one in `inittab`.

```
append="init=/sbin/myinit 5"
```

It is `init`, and not the kernel, which then configures the system, starts services (a more correct term for applications such as Apache) and opens virtual terminals. And this is where the real fun begins!

Once the kernel has started

`init` prepares the machine to work in a particular way: what software should be running, what terminals should be open, and which network services are to be allowed. It does this using *runlevels*. A runlevel describes the basic running state of the system, so naturally you can only be in one runlevel at a time.

A runlevel describes the basic running state of the system

There are a number of runlevels that are, by convention, used to describe a working Linux system, with 2 or 3 being usual for a multi-user configuration. The complete list is given below (according the Linux Standards Base - LSB).

- Runlevel 0 (halt) - Shuts down everything and brings the system to halt.
- Runlevel 1 (single user) - Useful for maintenance work.
- Runlevel 2 (multi-user) - multi-user, but with no network services exported.
- Runlevel 3 (multi-user) - normal/full multi-user mode.
- Runlevel 4 (multi-user) - reserved for local use. Usually the same as 3.
- Runlevel 5 (multi-user) - multiuser, but boots up into X Window, using `xdm`, or similar.
- Runlevel 6 (Reboot) - As 0, but reboots after closing everything down.

Most systems default to 2, 3 or 5.

Although the kernel starts `init` at the default runlevel, this can be changed at any time, provided you are the superuser, without rebooting your computer.

```
# init 2
```

This will switch the system into runlevel 2, starting all services that should be running at this runlevel, and killing those that shouldn't. Using runlevels as a profile in this manner lets you remove services during system maintenance, such as the network, very simply.

Looking at the runlevel table again you will notice runlevel zero is called "halt". This is not a misprint! Since a runlevel describes what services should be running, switching to a runlevel that closes all of its services and runs a program to halt the processor would be a considered shutdown procedure. Therefore:

```
# init 0
```

is equivalent to the more descriptive:

```
# shutdown -h now
```

Although longer, the latter is preferable because it is more extensible; letting you shutdown in half an hour, say.

Table for one

As with most programs under Linux, `init` has its own configuration file stored in `/etc`. It is called `inittab` and consists of comments (beginning with the over-familiar # symbol) and configuration data that indicates what, where and when particular services are to be run. The `inittab` file itself is well commented, and worth reading.

The first part of the file tells us:

```
# The default runlevel.
id:2:initdefault:
```

This format is typical, as each line in `inittab` has four fields separated by a colon. The first portion is an *identifier* for the action, and can be anything provided it is unique within the file (with the exception of the virtual terminals, which will be covered later). The second indicates the runlevel(s) to which this rule applies. In the above case it means runlevel 2 only, while the case below would work in all "multi-user" levels.

```
message:2345:wait:echo "In multi-user mode"
```

Parameter three is called the *action*. It indicates how the command (given in parameter four) is to be run. There are numerous actions available, and most are used in the `inittab` provided by most default distributions of GNU/Linux. The common ones are shown in the list below.

- `wait` – Execute the command, and wait until it completes before moving onto the next one. Used mostly for running software in sequence. If a program can not be run, `init` will work through the rest of the file. `init` does not stop on errors, but will report them to the console.
- `respawn` – Execute the command, but respawn it when the process completes. Used for virtual terminals that need to re-run the login prompt (through `getty`) whenever the user logs out.
- `ctrlaltdel` – Whenever the "three fingered salute" is given, run this program. It is usually configured to reboot the machine.
- `off` – Disables the entry, without deleting it from the file.
- `initdefault` – The default runlevel used if `init` is called without an argument.
- `sysinit` – The command is run first before anything else, when the system boots (runlevels are ignored).

Finally, the executable in parameter four may be a command, script or daemon, and can include arguments if required. But both the command and its arguments may be omitted, as we saw with `initdefault`.

The switch

`init` is a simple beast. It reads through each line in `inittab`, executing every command (relevant to the current runlevel) in the order in which it's presented in the file. The first command invariably performs system initialization, by specifying the sysinit action.

```
si::sysinit:/etc/init.d/rcS
```

This is the Linux equivalent to the `autoexec.bat`, or `confis.sys`, files from Windows and DOS. Its purpose is to configure any system-wide parameters (such as the system clock, or the serial port), regardless of runlevel. Once `init` has handled the system initialization it switches to the default runlevel and continues reading the rest of the file.

The sysinit start-up code is handled by the `/etc/init.d/rcS` script, which starts each process that is catalogued in the `/etc/rcS.d` directory. Since the boot-up sequence doesn't have a runlevel, a pseudo-runlevel called 'N' is used.

When switching between runlevels, the set of running services must also change. While this is possible to do from inside `inittab` and the `/etc` directory, it is cumbersome. To ease the pain, Linux uses a set of directories, one per runlevel (called `/etc/rc0.d`, `/etc/rc1.d`, `/etc/rc2.d` etc), that describe the services that must (and must not) be active in that specific runlevel. A script called `/etc/init.d/rc` is then responsible for starting and stopping them. The directory structure and absolute location of `rc0.d` and `init.d` can be inferred from the `inittab` file. If yours differs then refer to the SysVInit textbox in this article.

`init` is clever enough to start services in the correct order

Taking runlevel 2 as an example, this has a directory called `/etc/rc2.d` which contains a number of files. Those beginning with the letter 'S' are services that will start when we change into this runlevel, and those beginning with 'K' are services that will be killed.

`init` is clever enough to not stop, and then restart, any service that appears in the both the new runlevel, and the previous one. It is also clever enough to start them in the correct order—that order being one that you (the user) has numerically set up.

```
S10sysklogd
S12kerneld
S14ppp
S19bind
S19nfs-common
S20anacron
```

This fragment of the `/etc/rc2.d` directory from a GNU/Debian box shows that `sysklogd` will be started first, followed by `kerneld`, then `ppp`, and so on. The `/etc/init.d/rc` script will execute each of these files in order, passing it either the argument "start" or "stop", depending on whether the filename begins with an 'S' or 'K'. This gives the `/etc/rc2.d` directory similar functionality to the Windows Startup folder. But while Windows hides the order and method by which its startup program runs,

SysVInit

SysV Init has been incorporated by the Linux Standards Base as the method for system initialization. Depending on the distribution and version you use, the locations of certain files may be different from what we've given here. Firstly, the `init` program might live in `/sbin` and not `/etc` (although most distributions moved it there some time ago). And secondly, all of the runlevel configuration directories exist not in `/etc`, but in `/etc/rc.d`. The latter directory also holds the `rc` script, `init.d` directory and all other data mentioned above. Its working method, however, is no different. The `inittab` file will detail where your scripts live. Two scripts that also appear are `rc.local` and `rc.sysinit` (the latter will in turn usually run `rc.serial`). The execution order here is that `init` will run `rc.sysinit` first (configuring the network, checking the file system, and so on), followed by the runlevel scripts, and finally the `rc.local` file. If they don't exist, no error is produced, and Linux continues booting as normal.

Linux makes them available with ease, executing them directly from the `rc` script.

The `sysinit` directory `/etc/rcS.d` works in exactly the same way, but as it contains only system configuration information, no 'K' files are required. In contrast, booting up into single-user mode (runlevel 1) has almost nothing except kill files, to stop all the old services.

The missing link

The files in the `/etc/rc2.d` directory, however, are not actually files. They are links to scripts that do the real work! The script for `S10sysklogd`, for instance, would reside as `/etc/init.d/sysklogd` (the `S10` having been stripped off) and would start or stop the service based on its argument.

All the startup and shutdown scripts are in this directory (`/etc/init.d`), so controlling services on-the-fly is very easy. You can stop, start, or restart them with one simple command. Namely,

```
# /etc/init.d/apache start
```

Controlling services on-the-fly is very easy. You can stop, start, or restart them with one simple command

Some scripts will also support the restart directive. This removes the obligation to kill each process, and restart them manually, whenever a change to the configuration file is made. It also removes the need to reboot after installing new software, since the start command can be called directly.

You *can* add these links yourself with the `ln` command:

```
# ln -s /etc/init.d/apache /etc/rc2.d/S20apache
```

However, adding the same link to several runlevels (and its equivalent kill version to the halt, shutdown and single-user runlevels) can be a little monotonous, and therefore prone to user error. So there is a script that helps. It's called `update-rc.d`, and has the usage:

```
usage: update-rc.d [-n] [-f] <basename> remove
 update-rc.d [-n] [-f] <basename> \
   defaults [NN | sNN kNN]
 update-rc.d [-n] [-f] <basename> \
   start|stop NN runlvl runlvl .
  -n: not really
  -f: force
```

The basename is the name of the script, and would be `apache` in our example.

```
# update-rc.d apache remove
```

This would remove all the Apache symlinks in the `/etc/rc?.d` directories (where "?" can be any character). You must make sure the `/etc/init.d/apache` script is also removed, either by manually deleting it, or by use of the `-f` flag.

The `defaults` option will start the service in all multi-user runlevels, and stop it in the single user, halt and reboot runlevels.

If you want to add services to specific runlevels, the "start|stop" option must be used. This requires the order parameter. The runlevels are given as separate parameters, and terminated with an all-important full stop.

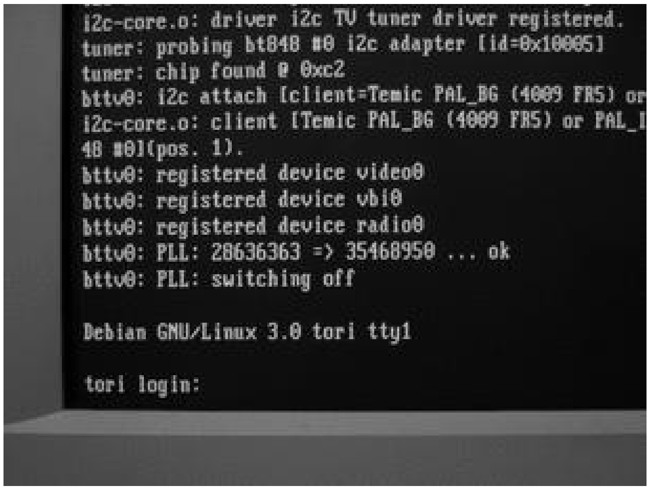
```
# update-rc.d apache start 20 3 4 5 .
 Adding system startup for /etc/init.d/apache \ldots{}
  /etc/rc3.d/S20apache -> ../init.d/apache
  /etc/rc4.d/S20apache -> ../init.d/apache
  /etc/rc5.d/S20apache -> ../init.d/apache
```

The NN symbol has been replaced with a number indicating the placement of the service in the start-up sequence. This represents the positional number we saw earlier on each of the symlinks, and can be omitted when using defaults. The default value is 20—a sensible choice since the network components are ready by this time.

The end of innocence

Once the various services have started, the `inittab` script arrives at the actions to configure the three-finger salute (ctrl-alt-delete) and prepare the virtual terminals. These work simply by running the `/sbin/getty` program (or equivalent) on each specified terminal. The respawn action is required to repopulate the virtual terminals after a user has logged out. If the terminal is connected via modem, `inittab` can handle that too by using `mgetty` instead of `getty`. At this point we have a login prompt, and our journey is over.

Ready at the command prompt



Conclusion

The apparently simple act of getting a prompt on-screen is very involved. However, it is not as complex as it first appears since it is comprised of several small steps, each building on the previous one. Each step increases our understanding and helps us streamline our system.

Copyright information

About the author

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders! When Steven Goodwin goes down the pub he doesn't talk about football. Or builders. He talks about computers. Constantly. . .