Linux Boot Scripts

Richard Gooch

21-NOV-2002

Introduction

This document describes the new Linux boot scripts that I'm developing. These boot scripts are a result of frustration with existing boot scripts and intensive dicussions with Patrick Jordan. We (Patrick and I) have also had email discussions with Larry McVoy and David Parsons, and their input is appreciated (even if they don't totally agree with us:-).

The new boot scripts require a modified version of simpleinit(8), which is distributed with the util-linux package. See http://www.kernel.org/ for a mirror site near you. I'm the new maintainer of simpleinit(8), and the changes I've made are available in the latest util-linux package.

OK, with all that out of the way, let me tell you why we think a new boot script scheme is needed. The two main existing schemes are the so-called "BSD" and "SysV" styles. Each have their respective disadvantages:

BSD-style problems

In this scheme, booting is controlled by one of a very few number of boot scripts. Often, there is a master boot script (typically `/etc/rc`) which orchestrates the whole boot procedure. This scheme is fairly easy to understand, as it has only a small number of scripts to read and the order in which things are started up is quite clear from the master boot script. It is fast, simple and efficient.

Where this scheme fails is in its scalability. If a 3rd-party package needs to have an initialisation script run during the boot procedure, it needs to **edit** one of the existing boot scripts. Such editing is dangerous, as boot scripts are fragile at the best of times. A simple mistake by the installer can lead to an unbootable system.

SysV-style problems

These boot scripts, in the tradition of SysV, can do anything. They are flexible. They are scalable. They can run industrial-strength systems. Unfortunately, they're bloated and ugly. OK, maybe they're not ugly to everyone (not that I've ever heard anyone admit they're elegant, only "it works"), but they sure are bloated. They are complex and hard to navigate. To see this, read about how they work:

This scheme places a number of mini-scripts in a master directory (typically `/etc/rc.d/init.d/`) which collectively can boot most of the system. Each of these mini-scripts starts and stops one "service". This is quite neat and modular.
A master boot script is used to orchestrate the boot process, which does some "special"

setup (i.e. anything which was too hard to put into a mini-script), and then proceeds to run each of the mini-scripts **in another directory**. The order is based on shell wildcard expansion rules.

The "other directory" is populated with symbolic links back into `/etc/rc.d/init.d/` (where the scripts are kept). Each script usually has two links to it. One starts with 'S' and the other with 'K'. The "S*" scripts are called when booting up the system, the "K*" scripts are called when shutting down the system. The desired ordering is achieved by using numbers after the 'S' and 'K' in the link names.

So a link with name "S10" will run before "S15", which in turn runs before "S20". It is the responsibility of the system integrator to name these links such that services are started and stopped in the correct order. A 3rd-party software installer can "simply" place their startup script in `/etc/rc.d/init.d/` and then create a symbolic link to the script in the "other directory". The installer has to pick a name that is not already taken, and has to determine the number to use (which depends on how far into the boot procedure the script must be run).
The author of the system boot scripts must therefore allocate numbers with sufficient gaps between them to allow for later insertions. Typically, the numbers 10, 20, 30, 40, 50, 60, 70 and so on are chosen. This reminds me of when I was a youngster programming BASIC on my Apple][. Every line had to be given a number, and you had to be careful to leave "space" for later insertions. The SysV numbering isn't quite so restrictive, as it is possible to append an arbitrary string to the number, which effectively increases the number space. Typically, the name of the script is appended, such as `10inetd` and `10named`. Thus, it is possible to "group" scripts so that the order between groups is well-defined, while ordering within a group is unknown (or knowable but not important).

The SysV booting scheme also supports the concept of "runlevels". What this means is that the system may be booted "all the way" (by convention, this is runlevel 5) by default, but may also be booted only part of the way. The most common purpose is to allow the system to be booted "single-user" (i.e. maintenance/repair mode), where only a handful of services are started. The runlevel scheme is supported by splitting the symlinks in the "other directory" into a number of directories, each directory corresponding to a runlevel. These directories are typically named: `/etc/rc.d/rc0.d/` `/etc/rc.d/rc1.d/` `/etc/rc.d/rc2.d/` `/etc/rc.d/rc3.d/` `/etc/rc.d/rc4.d/` `/etc/rc.d/rc5.d/` `/etc/rc.d/rc6.d/`.

The master boot script will start all scripts in the runlevel directory corresponding to the desired runlevel. Thus, the system can be booted to runlevel 1 by running the scripts in `/etc/rc.d/rc1.d/` (this is often "single-user" mode). Then, perhaps after some maintenance work the system can be booted all the way by switching to runlevel 5 by stopping services for runlevel 1 and starting the scripts in `/etc/rc.d/rc5.d/`. Similarly, the system can be taken from a higher runlevel to a lower one by stopping services.

The problem with all this, if it isn't obvious already, is that it's complex. The directories of symlinks make it difficult to see what is being run and how all the pieces fit together. If that doesn't convince you, consider the number of words required to describe this scheme. Note how the BSD-style scheme is much easier to describe (and by extension, understand).

All that is going for the SysV-style scheme is that it works. We believe that it is not elegant, and is confusing to experienced system administrators (when first exposed to the scheme), let alone novice administrators. And there remains a problem for 3rd-party boot scripts: which symlink name should be chosen? Usually the script is started in runlevel 6, because by that time "most" services are available. The simplest solution is to pick a random high number, which "should" work. Also, the use of numerical runlevels is far from intuitive. While old-guard SysV administrators may feel the runlevel definitions are simple to learn, the reality is the numbers convey no meaning. Certainly novice system administrators (the bulk of the Linux population now) will just scratch their collective heads and say "ah well, I guess that's just Unix".

The New Scheme

OK, so you've gotten past the ranting section. Thanks for your patience.

The solution we came up with is simple yet powerful. There is **no** master script which orchestrates everything. It's all done by init(8) and need(8) which provide the mechanism. The master script (which my old boot scripts have, and which SysV also has, despite their attempts to modularise), is broken into a bunch of smaller mini scripts.

The mini scripts are kept in `/sbin/init.d` and init(8) runs **all** of them, in random order. Ordering of the mini scripts is controlled by the scripts themselves. Each script runs any other scripts it depends on, using the need(8) programme which ensures that a script is only run once. It doesn't matter which order init(8) starts running the scripts, it all magically sorts itself out.

3rd-party scripts need only use need(8) to ensure services they require are running. That's all there is to the scheme. It's worth noting that Mastodon Linux (by David Parsons) has a similar dependency-based scheme (although it doesn't go quite as far). Thanks to Larry McVoy for pointing this out. NetBSD subsequently implemented a similar system although less flexible scheme, described here.

Implementation details

By default, simpleinit(8) will run `/etc/rc` as its startup script. The modified version allows you (either at the boot prompt or in `/etc/inittab`) to specify an alternative script to run. If the script specified is in fact a directory, all the scripts in that directory are run, in random order.

In the new scheme, init(8) is configured to run all mini startup scripts in `/sbin/init.d/`. Each script starts/stops one service (i.e. printing, filesystem checks, NFS mounting and so on). Ordering requirements are solved by each script pulling in other scripts it depends on using the need(8) programme. This basically means "run with dependency check". So the NFS export script would be:

```
#! /bin/sh
# /sbin/init.d/nfs-export

case "$1" in
  start)
    need portmap || exit 1
    rpc.mountd
```

```
    rpd.nfsd
    ;;
  stop)
    # Stop it
    ;;
esac
# End
```

or something like that. The need(8) programme is used to run a script with. If the programme has not been run before, need(8) will run it. If it has already run, need(8) does nothing.

Single-user and runlevels

For single-user mode, init(8) can be configured to run a specific script (or directory). This script can provide a featureful or stripped-down single-user mode, at your choice. Different runlevels are supported in the same way. Whatever argument is passed to init(8) at the command line (boot prompt), it is appended to a configurable prefix and together specify the script (or directory) to run. Thus, you can pass in "single", "3", "6" or "multi" and all that is required is the appropiately named script/directory.

There are two ways in which traditional runlevels can be supported. One is that an appropriate directory is created with symlinks back into `/sbin/init.d/`. A more elegant solution is to have a script for each runlevel, which would look something like this:

```
#! /bin/sh
# /sbin/init.d/runlevel.3

case "$1" in
  start)
    need runlevel.2
    need portmap
    mount -vat nfs
    ;;
  stop)
    umount -vat nfs
    ;;
esac
# End
```

Optimisations

One possible optimisation of our scheme is that init(8) reads the /sbin/init.d/ directory in FS order and reads in each file into a dummy buffer, so as to populate the page cache. This will reduce the number of head movements, which I suspect is one of the causes of the slow bootup of RedHat. My handful of boot scripts start up much faster.

Runlevels and rollback

OK, the idea is that init(8)/need(8) keeps track of what boot scripts have been run, forever (well, until we turn the lights off on this kernel and boot again). At any time, you can ask need(8) to run another boot script, with full dependency checking. And the new script that was run is also recorded in the table.

An orderly shutdown is as simple as rolling back the entire table. The algorithm is trivial: get the last entry in the table and run the appropriate stop script (which removes itself from the table). Repeat the process until the table is empty. All services have been stopped in the reverse order in which they were started.

Increasing runlevel is easy: just run the desired runlevel script. So going from runlevel 2 to 3 involves running runlevel.3 under the dependency management scheme. Simple.

Going from runlevel 3 to 2 is slightly more complicated, but not much. Just roll back, stopping each script/service in reverse order. As each is stopped, remove it's entry from the dependency table. Stop at "runlevel.2" (i.e. don't stop "runlevel.2", only the scripts listed after it).

This scheme works because "runlevel.3" is added to the dependency table **after** it can pull in new dependencies (because it's added to the list once it completes). So once you've rolled back to "runlevel.2", you know you've stopped all the services "runlevel.3" has started, plus all the services it depended on, **but not the services runlevel 2 depended on, or runlevel 2 itself**. Magic.

For switching between runlevels to work, the burden is placed on the runlevel scripts, not init(8), which is important IMO. Earlier I showed a sample implementation for "runlevel.3". It's important because it provides maximum flexibility in the construction of boot scripts and keeps init(8) simple.

Multiple providers and provide(8)

Sometimes, you have multiple service providers for the same generic service. For example, you might have sendmail(8) and qmail(8) installed on your system, and each has a boot script associated with it. Each one provides the "mta" service.

In this case, you only want one of these script to be started. It might not matter which one is started, or perhaps each script may check some system-specific configuration to determine whether or not it should start the service. Either way, all scripts providing the generic service should be run, but only one should start the service.

The solution to this is the provide(8) programme. It tells init(8) that the calling programme/script is able to provide the generic service. Init(8) then makes sure that only one provider will actually provide this service. An example script follows:

```
#! /bin/sh
# /sbin/init.d/sendmail

case "$1" in
  start)
    if [ ! -f /etc/mail/sendmail.cf ]; then exit 2; fi
    provide mta || exit 2
    need portmap
    /usr/sbin/sendmail -bd -q15m
    ;;
  stop)
    killall sendmail
    ;;
esac
# End
```

The need(8) implementation

Originally, I had intended to put most of the intelligence into need(8) and have init(8) only maintain the database of scripts. When the time came to start cutting code, I realised that this was not the most effective approach, since there may be no IPC services available, apart from named FIFOs and signals. Supporting parallel full-duplex communications with one or two fixed FIFOs and signals is quite difficult.

Script starting and stopping, as well as database management, is performed by init(8), and need(8) is a trivial programme which simply writes service requests to the FIFO and waits for a success/failure signal.

Because the dependency table for init(8)-started processes is kept in init(8), it makes partial rollbacks (switching between runlevels) easier to implement. One nice thing that we can rely on is that init(8) never dies, and doesn't crash (if it does, we're rooted anyway). So keeping the table in VM is quite safe. And it's small anyway, so we don't have to worry about memory limitations.

Note that need(8) and provide(8) are actually symbolic links to the initctl(8) programme.

Wrapup

I think that covers it. We've addressed the issues of ordering and modularity with a scheme that is simple but should give the full flexibility required.

There are other issues we still have to deal with (such as personalities (a scheme I came up with in my old boot scripts for configuration control)), but that's orthogonal to the above issues.

History

Some people have queried how this project came to be, so I've jotted down this rambling account. If this interests you, read on, otherwise pass onto the next section.

For years, Patrick and I have been bitching and moaning about the way Unix systems boot. "One day" we were going to figure out a better way. That day came on Saturday, 29th January 2000. After an afternoon checking out the new VW Beetle, we settled back and started plotting and arguing.

One significant issue where we disagreed was in how dependency data were to be gathered. Patrick suggested parsing the scripts, which I didn't like. While it would have made some things easier, this approach would either be too limiting (it would not support conditional dependencies), or it would require writing a full parser. Writing a full parser would yield something almost as complex as bash. In addition, it would not support scripts in other shell languages, or binaries. In the end, I convinced Patrick a parser was not practical.

That night, I sent off an email to Larry (with whom I'd sparred before, and he had previously expressed discontent with the way booting worked). Soon, he pointed out David had done something similar for his distribution (Mastodon Linux). David had done

something using a support script, which required a writable root FS. I considered this a significant limitation. Also, he hadn't gone far enough in my opinion, as he still had some large script orchestrating the boot process. Patrick and I both agreed that there should be no "special" or master script.

After (too much) email discussion, I decided that Patrick and I were still on the right track, and went forth and coded. This resulted in the basic need(8) implementation and a set of sample boot scripts using this scheme. Parallel booting even worked.

In March, Wichert Akkerman (Debian project leader) was in town for the Linux conference and Expo, which was our chance to sell the idea to the Debian project. Wichert liked the idea (in fact he'd once tried something similar, but never got it fully working), but wanted the addition of the provide(8) feature. This required more work, and thus the project stalled, since on my return from Sydney I had a large backlog of work to deal with.

Finally, in October, sitting on a plane headed for ALS, I had the time to get back to boot scripts, and started writing the provide(8) implementation. Shortly after my return from this trip I had finished this work, and finally had emptied out the ToDo list. Joy of joys!

As a side note, I find it interesting that other people have considered a similar approach (I've talked to people at conferences and had resonses like "oh, yeah, I've thought about doing that"). It seems this idea was ripe for the plucking. Hopefully this will translate into community acceptance.

Status

All the described changes to simpleinit(8) have been made, and we're already using it for our init(8). I have finished the need(8) and provide(8) implementation and the basic rollback mechanism. shutdown(8) makes use of rollback support to give ordered shutdown.

All these changes have been incorporated into util-linux-2.10q.

Things to do:

- I've considered keeping a full dependency history inside simpleinit(8) (right now it only keeps track of the currently depended-on service for each script). This would allow any service to be stopped and all services which depend on it to be stopped (dependent services would be stopped first, of course). This would be more flexible than either runlevels or rollback. In addition, a stopped service could still be recorded in the database and thus restarted with all the services that depended on it also being restarted. I have not yet determined whether these features would yield sufficient benefit to justify the implementation effort.

A set of sample boot scripts are available as a gzipped tarfile. I consider these "production quality". They have been in use on workstations and servers for years, and perform very well. They are fast and reliable. If there are missing features (such as configurations or daemons that are not supported), please let me know and I'll update them.