

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Advanced Boot Scripts

Richard Gooch

EMC Corporation

rgooch@atnf.csiro.au

<http://www.atnf.csiro.au/~rgooch/linux/boot-scripts/>

Abstract

This paper describes the design and implementation of a dependency-based scheme for system boot scripts. This scheme preserves the modularity of SysV-style boot scripts but does not suffer from its limitations (such as a complicated directory tree populated with symlinks, and the need for global dependency knowledge).

The dependency-based scheme simplifies the creation and integration of boot scripts by requiring only knowledge of direct dependencies (i.e. local rather than global knowledge). Dependency management is performed by **simpleinit(8)**, which may execute boot scripts in parallel, when those scripts have no cross dependencies.

This paper seeks to expose this new scheme to a wide audience, including distribution maintainers, with the hope that more widespread adoption follows.

1 Introduction

I propose a new mechanism for booting user-space on Unix-like systems. This scheme is a significant departure from existing boot mechanisms, and is a response to their respective limitations. The two main existing schemes are the so-called “BSD” and “SysV” styles. Each have their disadvantages, discussed below.

1.1 BSD-style

1.1.1 Mechanism

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.

1.1.2 Limitations

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.

1.2 SysV-style

1.2.1 Mechanism

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 considered too difficult 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 wild-card 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 II. 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.

These boot scripts, in the tradition of SysV, can

do anything. They are flexible and scalable and are designed to run large systems.

1.2.2 Limitations

A significant disadvantage of this scheme is its complexity. A simple measure of its complexity is the quantity of text describing it compared to that required for describing the BSD-style scripts. Due to this complexity, it is often difficult to see how the various scripts fit together and determine the execution sequence.

This intricate web of scripts, directories and symbolic links is difficult to construct and difficult to administer. Even an experienced system administrator can be confused by this scheme when first exposed to it. Novice administrators may be expected to be quite perplexed. With the growing popularity of Linux, the vast majority of Linux users are not experienced system administrators, but must still administer their systems. The SysV scheme does not cater to their needs.

While the SysV scheme is more scalable than the BSD scheme, 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.

Finally, 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.”

2 An Alternative

As indicated, each existing scheme has advantages and disadvantages. The use of mini-scripts provides scalability, and thus this aspect of the SysV scheme should be preserved. What is required is a mechanism that starts mini-scripts in an ordered fashion yet is easy to understand and does not suffer from name-space or number-space limitations.

The proposed solution is simple yet powerful. There is *no* master script which orchestrates everything. Instead, all scripts are executed in parallel. Ordered sequencing is obtained by allowing each script to declare which services it needs available (i.e. what it depends on) in order to successfully complete. Even the master script found in SysV-style booting schemes is eliminated.

Whenever a script declares that it needs another, it is suspended (blocked) until the required service is available or is determined to be unavailable. This simple mechanism enforces strict sequencing with precisely the level of granularity desired.

Placing dependency information inside each script has the following advantages:

- it is immediately clear what other services a script depends on
- the information is localised, requiring no global orchestration by the system integrator
- 3rd-party scripts can fine-tune their dependencies
- 3rd-party script installers need not be aware of the global sequencing details.

A beneficial side-effect of executing scripts in parallel is that some services will be started in

parallel, once the common services they depend on are available. This can reduce the time taken to boot the system.¹

3 Implementation

The **init(8)** programme is responsible for executing the boot scripts and orchestrating the correct sequencing. To accomplish this, I modified the **simpleinit(8)** programme from the **util-linux** package to support dependency-based boot sequencing. I wrote the **initctl(8)** programme which is used by scripts to declare their dependencies, and a set of boot scripts using this new mechanism. These boot scripts may be used as a guide for writing another set, or may be used directly in a production system.

The mini scripts are kept in a directory 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 (an alias of **initctl(8)**) which ensures that a script is only run once. It doesn't matter which order **init(8)** starts running the scripts, as **need(8)** ensures scripts wait as required.

3rd-party scripts need only use **need(8)** to ensure services they require are running. This eliminates the problem of deciding where to place the script in the sequence.

The changes to **simpleinit(8)** and the new **initctl(8)** have been incorporated into the **util-linux** package.

3.1 Implementation details

By default, **simpleinit(8)** will run `/etc/rc` as its startup script. The modified version allows the system administrator (either at the

¹consideration must be given to the effect this may have on disc head seek times, which could eliminate gains due to parallelism

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, file-system checks, NFS mounting and so on). Take the example of the NFS export script, which starts the daemons **rpc.mountd(8)** and **rpc.nfsd(8)**, but must wait until the RPC portmapper is running before starting the NFS daemons. The sample script below demonstrates this:

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

case "$1" in
  start)
    need portmap || exit 1
    rpc.mountd
    rpd.nfsd
    ;;
  stop)
    killall rpc.nfsd && \
    killall rpc.mountd
    ;;
esac
# End
```

The **need(8)** programme is used to run a script, and wait for its completion. If the programme has not been run before, **need(8)** will run it. If it has already run, **need(8)** does nothing. The exit code indicates whether the service (the portmapper in this case) started successfully or not. Since the portmapper is required, the script tests the exit code from **need(8)** and fails if it is unavailable for any reason.

3.1.1 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 simple or arbitrarily complex single-user mode, at the discretion of the designer of a set of boot scripts.

Different runlevels are supported in a similar fashion. Whatever argument is passed to **init(8)** at the command line (boot prompt), it is appended to a configurable prefix and together they specify the script (or directory) to run. Thus, you can pass in "single", "3", "6" or "multi" and all that is required is the appropriately named script or 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/`. This approach may be used when a rapid implementation of runlevels is desired. A more elegant solution is to have a script for each runlevel. An example script is shown below:

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

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

In this example, the distinction between runlevels 2 and 3 is that runlevel 3 will additionally mount remote file-systems. Thus, runlevel 2 is required as is the portmapper.

3.1.2 The **initctl(8)** implementation

Originally, I had intended to put most of the intelligence into **initctl(8)** and have **init(8)** only maintain the database of scripts. This approach was quickly discarded, since it would require reliable, full-duplex inter-process communication (IPC) services. Under Linux, these may be available as loadable modules, and thus may not be available at the time **init(8)** starts. The only IPC facilities that may be relied on are named pipes (FIFOs) and Unix signals. These are not suited to parallel, full-duplex communications.

The approach I adopted was to place the responsibility for script starting and stopping, as well as database management, with the **init(8)** programme, and have a simple control interface. By limiting the amount of information that is sent from **init(8)** to **initctl(8)** to a simple available/not-available/failed status, the need for a second FIFO (for each instance of **initctl(8)**) is avoided, and Unix signals may be used instead.

The **initctl(8)** control interface is a trivial programme which simply writes service requests to the control 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 and complete rollbacks (switching between runlevels and orderly shutdowns) easier to implement. Since **init(8)** never dies, and doesn't crash (if it does, the system will hang), it is quite safe to maintain the database inside the virtual memory space of **init(8)**. Also, since the database is quite small, there is no significant resource consumption.

3.2 Optimisations

A simple optimisation which can reduce booting time is the pre-fetching of all the script files, which can reduce the number of disc head seeks. Without this optimisation, the disc head may have to travel back and forth between the script files, the daemons they start and their configuration files. Assuming the script files are close to each other on the disc media, a small number of seeks will suffice for pre-fetching the script files. This optimisation has been implemented, by reading the scripts in file-system order into a dummy buffer.

3.3 Runlevels and rollback

Because **init(8)** maintains a table of which boot scripts have been run and which have failed (if any), and since **init(8)** runs for the lifetime of the booted system, it is ideally suited to managing orderly shutdown of the system. Further, since at any time **need(8)** may be used to run another boot script, with full dependency checking, then **init(8)** may also be used to switch between runlevels.

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

Increasing runlevel is also quite simple: just run the desired runlevel script. Thus going from runlevel 2 to 3 involves running `runlevel.3` under the dependency management scheme.

Going from runlevel 3 to 2 is slightly more complicated, but not much. Again, the system needs to be rolled back, stop-

ping each script/service in reverse order. As each is stopped, its entry is removed from the dependency table. The process is stopped at `runlevel.2` (without stopping `runlevel.2` itself).

This scheme works because `runlevel.3` is added to the dependency table *after* it registers new dependencies (because it's added to the list once it completes). So once the system has rolled back to `runlevel.2`, we can be sure that all the services `runlevel.3` has started have been stopped, plus all the services it depended on, *but not the services runlevel 2 depended on, or runlevel 2 itself.*

For this switching between runlevels to work, the burden is placed on the runlevel scripts, not **init(8)**, which is an important point, because it provides maximum flexibility in the construction of boot scripts and keeps **init(8)** simple. The same rollback mechanism required for orderly shutdown may be used to switch runlevels. No extra logic is required.

3.4 Multiple providers and provide(8)

Sometimes, there may be 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` (Mail Transport Agent) service.

In this case, only one of these scripts should 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. In either case, 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 pro-

gramme/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
```

Here, the script first checks to see if its configuration file `/etc/mail/sendmail.cf` is available. If not, the script exits with a “not available” status code. Then, the script registers its intention to provide the `mta` service. If given permission, it proceeds to start the service, otherwise it exits.

4 Future Work

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 roll-back. 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. It is not clear whether these features would yield sufficient benefit to justify the implementation effort.

5 Acknowledgements

This work is the result of an evening discussion session between Patrick Jordan² and myself. The basic concept of a dependency-based booting scheme, and the semantics of the **need(8)** programme, were established during that session. I thank Patrick for his enthusiasm for this project and his willingness to try the new, experimental boot scripts.

The implementation of the multiple providers feature (discussed in section 3.4) was added as a result of discussions with Wichert Akkerman (then Debian Project leader, email: **wichert@cistron.nl**), where the needs of Debian were raised.

A similar (although less complete) dependency-based booting scheme has been independently developed by David Parsons for his Mastodon Linux³. Thanks to Larry McVoy for pointing this out.

Except where otherwise noted, all work is my own.

²<http://www.ariel.com.au/~patrick/>

³<http://www.pell.portland.or.us/~orc/Mastodon/>