debian

3rd Debian Conference

System Init Scripts and the Debian O.S.

Henrique de Moraes Holschuh*

June 2002†

Abstract

This paper describes the System V, OpenBSD / FreeBSD, NetBSD and Debian 3.0 init script systems. Some new proposals made by third-parties for init script systems are also described, with emphasis on RICHARD GOOCH's simpleinit system. The system init script abstraction and policy layer currently being deployed in Debian for the release after 3.0 is described in detail. Changes to that abstraction layer are proposed to allow for dependency-based init script systems in Debian.

1 Linux System Bootstrap

System bootstrapping procedures vary quite a bit, depending on which kernel is being used. With the imminent addition of the BSD kernels to Debian, we can expect the bootstrap procedures to vary even more in the near future[1].

For Linux kernels, the bootstrapping sequence depends on the existence of initrd images or compressed RAM file system images, and even remote boot images. Debian uses at least three variants of Linux bootstrapping: initrd for our very big includes-even-the-kitchen-sink-as-modules 2.4 kernel images; direct bootstrap for smaller 2.4 and 2.2 kernels, such as those normally created by users of the kernel-package package; and compressed RAM file system for boot-floppies and CD/DVDROMs.

Regardless of the kernel and bootstrapping method used, at the very end of a normal Debian bootstrapping process the kernel will have mounted the real root file system in read-only mode for us, and then it will execute */sbin/init*, which will run with pid 1.

/sbin/init will then proceed to call the system init scripts, which will do the dirty job of mounting file systems in read-write mode, loading modules, starting service daemons, and running a login shell for the user.

In Debian, these init scripts live in */etc/init.d*, and take one or more command-line parameters. The first parameter tells the init script which action it should take: start the service, stop the service, reload service configuration and so on. All other parameters are optional, and init-script-dependent. The script will return exit status zero if no errors occur, and non-zero otherwise.

This init script command line convention is part of the System V init script architecture, which is Debian's default. Init scripts following this command line convention and stored in */etc/init.d* will be called SysV-like init scripts[2] in this paper.

2 Init Script Systems

2.1 System V

The System V init script system is quite powerful, and very adaptable due to its modular design, true to System V style. This is a very big advantage over the original BSD-like init script

*hmh@debian.org †This is revision 1.16 of this document, commited in 2nd July 2002, 12:01:34.

[1] The Debian BSD effort will also introduce a new standard C library (libc), so system bootstrapping is the least of our worries.

[2] SysV is the short form for System V.

systems[3]. Given the displeasure many have for some parts of the System V init script system, this is a Very Good Thing, indeed: the modular design makes it possible to fix what is bad, and keep what works fine.

The System V init script system is well understood by most seasoned system administrators, and well defined. It works mostly well with the software packages concept, as long as they come from the same vendor. However, it is also messy and ungraceful, it is unintuitive, and it takes quite a bit of maintenance to keep the init script execution sequence just right.

The init script system is composed of three modules: */sbin/init* (Sysvinit), */etc/init.d/rc* (*/etc/rc?.d/** symbolic link farms) and the SysV-like system init scripts themselves (*/etc/init.d/**).

It has a concept of different system profiles, which are called runlevels. The runlevels are 0 (system halt); 1 (single-user mode); 2, 3, 4 and 5 (normal modes)[4]; 6 (reboot system); 7, 8 and 9 (normal but not normally used); and "S" (system startup). Runlevel "S" is only used during system startup and allow single-user mode login. What the other runlevels are used for varies from vendor to vendor, and sometimes even with the operating system's version.

2.1.1 Sysvinit

Debian's default (and currently only) */sbin/init* is the Linux System V init. It is packaged and maintained both for Debian and upstream by Miquel van Smoorenburg[5]. We shall call it Sysvinit.

Sysvinit has a large memory footprint[6]. On an ia32 system, it takes about 1280 kibibytes[7] of virtual space, and 480 kibibytes of RSS. That amounts to "an itty-tiny bit" of memory in these KDE/GNOME days, but it is hardly what you would use for memory-starved embedded systems.

Sysvinit is a fully-featured *init* as far as these things go. Sysvinit can:

- use a script to start every process it needs to (*/etc/initscript*);
- deal with several system profiles (runlevels), and switch between them upon user command;
- execute a configurable script to run the system startup setup (*/etc/init.d/rcS*);
- babysit processes (restart, kill, control fork rate), with different sets of processes for every runlevel;
- execute scripts on SIGPWR, SIGINT and kbrequest;
- re-execute itself without bothering the rest of the system;
- execute in system maintenance mode. These are single-user mode (runs only the system startup scripts, and then run sulogin), and emergency mode (only run sulogin);
- change the runlevel init script subsystem (one can tell Sysvinit to run something other than */etc/init.d/rc* to execute runlevel changes);
- deal with Linux serial consoles.

Sysvinit currently cannot:

- deal directly with kernel ACLs[8] (but patches for Sysvinit to add such support, and patches to the Linux kernel to work around Sysvinit's limitations do exist).

  The default ACL inheritable set[9] is "empty" in the mainline Linux Kernel (and also in Debian's). Sysvinit needs to change that, or ACLs are effectively disabled without a kernel patch. It should also be able to modify the capability bound set[10], so as to emulate the *securelevel* feature of the BSDs.

[3] At least from the point of view of a distribution using a package-based system, anyway. [4] The runlevels are normally used in additive mode. For example: runlevel 3 is runlevel 2 plus networking functions; runlevel 4 is runlevel 3 plus NFS and other remote services enabled; runlevel 5 is runlevel 4 plus a X11 display server. [5] miquels@cistron.nl [6] This is glibc's fault. Sysvinit itself takes only 25 kibibytes. [7] A kibibyte is 1024 bytes

[8] ACL is the short form for Access Control Lists. It is a fine-grained permission control system, capable of limiting even the superuser's access to the kernel. [9] Set of ACL permissions available to children of the current process. [10] The capability bound set selects either superuser or ACL-based access control for every possible ACL.

- directly control processes that change their process group (as most daemons do);

Usually only the first limitation is of any importance, as daemons are controlled through init scripts instead of by Sysvinit. Debian will need to address ACLs in a consistent way very soon: we already have the standard Linux kernel ACLs (that cannot be used by default due to Sysvinit's lack of ACL knowledge), and SE Linux[11] is coming. We also have at least one ACL-aware file system (xfs).[12]. The BSD kernels also have ACL-enabled extensions (Trusted BSD), which might enter Debian one day.

2.1.2 Init scripts symbolic link farm

The System V init script system decides which services are to be started or killed in a given runlevel (or during a runlevel change) using the */etc/init.d/rc* script. This information is stored by symbolic link farms in directories, one directory per runlevel. To figure out if a service is supposed to be started or stopped in a given runlevel, one must inspect the correct link farm.

The S runlevel is special, for Sysvinit is configured to run */etc/init.d/rcS* at system startup. One should not try to switch to runlevel S directly, as Sysvinit does not execute any init scripts for runlevel S.

The symbolic link farms are stored in */etc/rc?.d*, where "?" stands for the runlevel. The symbolic links are relative, and point to the system init scripts in *../init.d/* (i.e. to */etc/init.d/*).

The symbolic links are named with a starting *S* or *K* letter, followed by two digits and by a variable, non-zero number of letters[13]. Services that should be started have symbolic links starting with *S*. Services that should be stopped have symbolic links starting with *K*. One can have both an *S* and *K* symbolic link for the same service if that service is to be restarted during the runlevel change. The name after the digits is the name of the init script in */etc/init.d* that the symbolic link points to[14].

The symbolic links are processed in shell collation order (which will be the one for the C locale if one is lucky). First all *K* scripts are run with the *stop* argument, then all *S* scripts are run with the *start* argument. Some */etc/init.d/rc* scripts are known to try to optimize this a bit, and will not call a init script to start a service that should have been started in the previous runlevel and was not stopped in the current runlevel. Debian's */etc/init.d/rc* does just that.

One must make sure the scripts are numbered in the correct order. Scripts with the same number are not guaranteed to always run in the same deterministic order (due to locale collation rules, for example).

Right now, Debian does not take all the care that such a fragile ordering enforcement system requires. Maintainers often select the wrong ordering numbers for their init scripts, and that will only get fixed when someone notices it in her system, and file a bug. Also, the default order number of 20 has too many services.

2.1.3 System init scripts

The system init scripts are stored in */etc/init.d*. They are named after the service they control (or after the package that contains the script, but that's likely a Debian extension).

All scripts take at least one command line parameter. This parameter selects which action the script should perform. The common actions are: *start*, *stop*, *restart*, *reload*, *force-reload*, *status*. Other actions are possible, and often used. However, only the *start* and *stop* actions must exist (because they are the only ones the System V init script system actually uses).

Debian and the Linux Standard Base (also known as LSB) have further requirements for the system init scripts. There is ongoing work to make sure Debian's requirements are compatible with the LSB ones, both to allow better interoperation, and to be less confusing to end-users. This work is currently stalled, waiting for DEBIAN 3.0 "WOODY" to be released.

[11] Security Enhanced Linux: a Linux kernel with mandatory ACLs. [12] ACL-awareness is available for the ext2 and ext3 file systems as well, as kernel patches. [13] Extended *regexp* (S|K)[0–9][0–9][a–zA–Z]+; Most implementations will accept just about anything after the two digits, however.

[14] Most implementations will just follow the link to find the correct init script (such as Debian's).

The LSB defines a few exit codes, and requires that *start*, *stop*, *restart*, *force-reload*, and *status* be supported by all scripts, as well as that all error messages be printed to *stderr* and all status messages to *stdout*. It also requires all init scripts to be shell scripts[12]. LSB init scripts will probably be easy to convert to whatever system Debian is using.

Debian policy has many more requirements regarding the init scripts, which can be found at policy chapter 10.3.1. They are basically[14]:

- scripts ending in ".sh" are to be sourced during system startup, instead of executed in a subshell;
- actions *start*, *stop*, *restart* (defined to be stop if running, then start), and *force-reload* should be implemented. Implementing *reload* is optional;
- init scripts must behave sensibly in a variety of border conditions (which are then defined and the desired behavior described in the policy document);
- configuration variables used by the init scripts should go in special shell-snippet files in */etc/default/**.

2.1.4 Alternate System V-like init script system: file-rc

Most of the System V init script system is actually quite good. Complaints are almost always directed to the symbolic link farm in */etc/rc?.d*, which is hard to maintain and somewhat difficult to understand at first glance.

File-rc[9] takes advantage of the modularity of the System V init script system to do away with the symbolic link farm. The list of services that should be started or stopped in each runlevel is kept in a single */etc/debian/runlevel.conf* file, in an easy-to-parse, tabular format. One can have the full picture of the services in every runlevel with a simple glance at the table.

It still keeps the annoying ordering problem of System V's init script system, though.

File-rc is fully packaged, supported and integrated in Debian. The Debian maintainers Tom Lees[15], Martin Schulze[16] and Roland Rosenfeld[17] built upon the initial work by Winfried Trümper[18], and have improved file-rc many times since.

2.2 FreeBSD and OpenBSD init script systems

FreeBSD and OpenBSD have very similar, and simple init script systems inherited from 4.4BSD. Something good can be said about applying the KISS principle[19] to system initialization, but it makes for a less interesting specimen to study.

/sbin/init runs */etc/rc*, if told to go in multi-user mode. Otherwise, it starts a single-user mode shell. There is no concept of runlevels or system profiles, as far as */sbin/init* is concerned. It can, like Sysvinit, babysit processes. It can also run an */etc/rc.shutdown* script on system shutdown.

FreeBSD and OpenBSD */sbin/init* can also set the kernel security-level.

The system initscripts are all grouped up into one or more shell scripts, often by functionality (e.g. rc.network; rc.shutdown). There is no concept of packaging modularity. These scripts are sourced by the main */etc/rc* script. OpenBSD does not even attempt to have many rc.* scripts, remaining true to the KISS principle. FreeBSD throws "KISS" to the wind, and has many rc.* scripts.

2.3 NetBSD init script system

Unlike the two other BSDs, NetBSD has a very advanced init script system. It is arguably more advanced than System V's in some parts, and it shows interesting concepts that have been long standing requests from more advanced Debian users. The rationale and implementation details of the system are well described in [5].

/sbin/init is mostly the same as in the other BSDs. All the intelligence in NetBSD's init script system is in the */sbin/rcorder* script, called by */etc/rc* to define the order the scripts should be called.

[15] tom@lpsg.demon.co.uk [16] joey@debian.org [17] roland@debian.org [18] winni@xpilot.org [19] Short form for "keep it simple, stupid!". Complex systems are harder to maintain, and thus easier to break. Therefore, one should avoid complexity where possible.

```
# This is the configuration file for /etc/debian/runlevel.conf
#
#Format:
#<sort> <off>   <on>      <script>
05        -     0         /etc/init.d/halt
05        -     1         /etc/init.d/single
05        -     6         /etc/init.d/reboot
10      0,1,6   2,3,4,5   /etc/init.d/sysklogd
12      0,1,6   2,3,4,5   /etc/init.d/kerneld
[...]
89      0,1,6   2,3,4,5   /etc/init.d/cron
99      -       2,3,4,5   /etc/init.d/rmnologin
99      0,1,6   2,3,4,5   /etc/init.d/xdm
```

Figure 1: a file-rc runlevel configuration file

rcorder uses special keywords in the system init script files (inserted inside shell comments) to build a graph of the interdependencies of the scripts. Then, it outputs the topologically ordered graph[20] to *stdout*. */sbin/rc* simply goes through this list and executes every script in it.

It works very much like Debian's packaging system package dependencies without any version information. The keywords are: *Require*, *Provide*, *Before* and *Keyword*. They declare facilities for a given script, and the dependency information on other facility (and therefore, on scripts that declare that facility).

Circular dependencies are not allowed. The boot process starts with the scripts that have no facility requirements. Should any dependency fail to be resolved, *rcorder* will abort with an non-zero exit status, which will cause the boot process to switch to single-user mode.

Common functions accessed by the init scripts are placed in a */etc/rc.subr* shell file that all init scripts source. NetBSD also introduced a new, optional *rcvar* action that outputs all the configuration variables used by the init script.

The rcorder approach to ordering the init scripts makes it rather hard to do any sort of parallel execution of the system init scripts. NetBSD doesn't even attempt to do that. The information needed for parallel execution is there, but it would require a far different implementation approach to be used.

[20] i.e. ordered such as to make sure all declared dependencies are satisfied.

2.4 runit

Gerrit Pape's[21] runit[1] is written to work side-to-side with Daniel J. Bernstein's[22] daemontools[23]. It could be thought of as "daemontools init script system". It is a simple, straightforward init script system with three states (system init, multi-user, system halt) that depends on daemontools to do all the daemon servicing work.

runit's */sbin/init* is even simpler than the BSD's. It has a very small memory footprint. It can be linked statically against dietlibc[10] for a total virtual memory usage of 24 kibibytes and RSS usage of 20 kibibytes in Linux ia32. However, should one take into account the memory footprint of daemontools' *svscanboot* and *svscan*, there is little memory economy when compared to Sysvinit without daemontools.

runit's system init scripts work very much like the ones in FreeBSD and OpenBSD, and are packaging-unfriendly. There is no runlevel concept at all.

2.5 Simpleinit

Richard Gooch[24] has written much about the subject of using dependency information, and parallel execution in system init scripts (his work is

[21] pape@smarden.org [22] aka DJB, of *qmail* fame.
[23] *Daemontools,* like most other *DJBware,* has an obnoxious license that is not compliant with the Debian Free Software Guidelines. Its existence is, therefore, irrelevant for Debian. [24] rgooch@atnf.csiro.au

detailed in [3]) to automatically start and stop services in the correct order, very much like the NetBSD rcorder concept. His work influenced the development of at least two init script systems (minit and jinit), in addition to GOOCH's own "simpleinit" system.

2.5.1 The need(8) concept

The strategy used by GOOCH to implement dependency information seems to have become known as "the need(8) concept". His approach to the ordering problem is very different from NetBSD's: instead of building a dependency graph, every init script tries to run the scripts it depends on, using the *need* command. *need* simply ensures that every init script is run only once. It is beautiful in its simplicity. Until you get to shutdown sequences, and runlevel changes, that is.

There is some functionality in NetBSD's rcorder system that cannot be easily implemented using the simplest form of the need(8) concept: the *before* keyword, and reverting the startup sequence to run the shutdown sequence. Such functionality needs dependency information to be known beforehand.

This issue can be solved for the shutdown sequence by storing part of the dependency information when services are started, so that one can perform rollbacks by calling the init scripts with a stop action. Partial rollbacks are possible, and a shutdown is just a total rollback, followed by a call to *shutdown*.

The System V runlevel functionality can be implemented using rollbacks, but it will be slightly different from the real thing, as moving between runlevels might cause services to be stopped and then started again needlessly.

There is also a *provide* command. It was added on request of WICHERT AKKERMAN, with Debian in mind, even. Two services are not allowed to provide the same service at the same time; the second one to call *provide* will receive a failure exit status, and must itself fail, then.

The need(8) concept allows for easy parallel execution of scripts, if one makes *need* a synchronization point (and deals with the *stdout* and *stderr* from the init scripts to avoid the resulting console mess). The need(8) concept also places no requirement of the init scripts being shell scripts, as anything that can exec *need* and *provide* will work.

2.5.2 Simpleinit's implementation of the need(8) concept

The simpleinit init script system is distributed in the util-linux upstream package, but it is currently absent from Debian's binary util-linux packages. Its */sbin/init* is similar to a runlevel-less Sysvinit, but it appears not to handle the powerfail signals, nor to be able to re-exec itself (which would be useful during libc and simpleinit upgrades).

The simpleinit init script system replaces */sbin/init*, */sbin/shutdown* and */sbin/initctl*. It also provides */sbin/need* and */sbin/provide*, which are symbolic links to *initctl*. Configuration goes in */etc/inittab*, which is close to the Sysvinit *inittab* file.

Simpleinit does handle something akin to the runlevel concept during system initialization. It supports different init script sets, by selecting a different init script (or directory of scripts) based on the first argument given in the command line.

Thus, you could have different directories, one for each runlevel, and symbolic links to the real scripts in some other directory... but you would just be resurrecting the dreaded System V symbolic link farm, then.

It is much better to implement runlevels as dependency chains, using a "runlevel.1" service for example, and tell simpleinit to execute that script on startup. This works very well for additive runlevels.

A special startup/shutdown script is also supported, if configured in */etc/inittab*. It is called with a *start* argument during initial system startup, after all terminal programs listed in */etc/inittab* are started, and with a *stop* argument at the start of the shutdown process.

Dependency tables are kept in memory by *init* itself. */sbin/initctl* talks to */sbin/init* through a named FIFO (*/dev/initctl*), and signals. To perform rollbacks, one calls */sbin/need* with the -r switch and the service up to which it should rollback.

/sbin/need simply requests the running */sbin/init* to start a service, or to rollback up

to a service. *Init* then forks, and executes the needed service(s) with a *start* or *stop* action.

2.6 minit

FELIX VON LEITNER's[25] minit[2] is a small */sbin/init*, linked against dietlibc[10]. It has a good set of features, and it includes its own daemontools-like set of service managers, removing any dependencies on non-free software[26]. It is service interdependency-based, just like NetBSD's rcorder system. However, it is much less powerful than rcorder, for it has only the concept of direct dependencies. It seems to have been developed loosely around the need(8) concept described in section 2.5.1.

According to minit's documentation:

- it can start services and take dependencies into account;
- it can restart services;
- it can start services in sync mode (i.e. wait until they terminate);
- there is a companion utility *msvc* that can be used much in the same way as the *svc* from daemontools. Communication works over two FIFOs: */etc/minit/in* and */etc/minit/out*;
- there is a companion utility *pidfilehack* that can be used to run daemons that start their own sessions and write the new pid to a file (OpenSSH, for example);
- it can pipe *stdout* to a dedicated log process.

minit uses directories under */etc/minit* to lay out information about the services it should control. Special files inside the directory set the service's characteristics:

depends: contains the names of services that must have been started before this service does. One name per line.

run: absolute symbolic link to the name of the program that should be executed for this service.

params: command line parameters (no shell expansion is done) to feed to the service. One parameter per line.

respawn: if this file exists, the service is respawned if it dies. This behavior is mutually exclusive to "sync".

sync: if this file exists, minit will wait until the service returns before proceeding with the next service. This behavior is mutually exclusive to "respawn".

log: if this directory exists, minit will treat it as a separate service, and its *stdin* is connected to the *stdout* of the parent service.

2.7 jinit

JOHN FREMLIN's[27] jinit[7] is an init script system based on the need(8) concept (see section 2.5.1), although it currently lacks *provide* functionality. Unlike all other init implementations I know of, it is written in C++[28].

According to jinit's documentation:

- it has complete service start and stop (rollback) support with dependency tracking. Stopping a service will cause all services depending on it to be stopped also;
- it can be told to respawn a command without editing *inittab* or *ttys*, through the *need* command;
- it communicates over System V message queues, so it does not need to touch a filesystem when it comes up;
- it will kill all processes and force umount filesystems automatically on shutdown;
- if it receives a fatal error signal, such as SIGSEGV, it will fork a child to dump core, and exec itself again to keep on working.

The following disadvantages are also listed:

- jinit is currently not very tested, and it is probably buggy;

[25] web@fefe.de [26] Software non-compliant with the Debian Free Software Guidelines.

[27] vii@penguinpowered.com [28] Which is *not* a good thing, as far as binary size goes. There is no dietlibc++, after all.

- it does not use a traditional (System V) *inittab* or (BSD) *ttys* configuration file;
- it cannot dump state to and exec a later version of itself, so state will be lost between upgrades;
- it cannot handle all events Sysvinit can (such as powerfail);
- jinit is bigger and more complex than other inits.

2.8 Busybox

Busybox[8] tries to be the Swiss Army Knife of embedded Linux, and as such it can also behave as a simple */sbin/init*.

Busybox has no concept of runlevels, but it will parse the standard Sysvinit */etc/inittab* file (and promptly ignore all runlevel information in there). It supports the following Sysvinit actions: *sysinit, respawn, askfirst, wait*, *once*, *restart*, *ctrlaltdel*, *and shutdown*, which is quite a lot, actually. It does not have respawn throttle control.

Busybox */sbin/init* can deal with serial consoles, and it is very suited for small boot-floppies and for bootstrapping installation systems.

2.9 twsinit

twsinit[4] is a very minimal init system. It concerns itself with doing the bare minimum required to get a system up and running, and it can take as little as 8192 bytes of virtual memory and RSS.

twsinit is capable of running a single startup script, and of respawning services. It has no concept of respawn throttle control, and one can assume that most other traditional */sbin/init* capabilities are also outside of its intended scope.

2.10 Serel

Leni Mayo's[29] Serel[11] aims at reducing the time the system spends booting. In order to achieve that objective, Serel implements[30] a dependency-based init script system capable of parallel execution. To deal with the dependencies, it implements the basic need(8) concept.

It is interesting to note it implements dependency both statically (it has to, in order to optimize the dependency tree for fastest boot) and dynamically. It is currently tailored to work out of the box with RedHat 7 systems.

Serel also has tools to analyze the time every init script takes to boot (which is logged[31] by Serel during bootstrap), and attempts to create an optimal execution path for the next boot.

Serel does not change */sbin/init*. It is executed by *init*, and spawns a daemon which controls the boot process. It then runs the init scripts in the order it was configured to by the optimization tools (if such information is available), to try to achieve the minimum bootstrapping time possible. Otherwise, it uses the dependency information to simply run as much stuff in parallel as possible.

The init scripts use an external command (*/sbin/serelc*) to communicate with the daemon, to tell it about their dependencies dynamically. Serel supports *provide*, and *need* dependencies. It does not support *before* dependencies.

Serel can also read dependency information stored in the LSB format (see section 3.1), and in its own XML/RDF[17]-based format. The RDF files can be visualized in a graphical view, as shown in figure 2.

3 Linux Standard Base Init Scripts

The Linux Standard Base also has something to say about system init scripts[12]. It defines the following actions:

start: start the service.

stop: stop the service.

restart: stop and restart the service if the service is already running, otherwise start the service.

reload: cause the configuration of the service to be reloaded without actually stopping and restarting the service.

[29] leni@moniker.net [30] Serel version 0.3.2 was the current release when this paper was written.

[31] Such logging is a feature I would like to see in other init script systems. As well as logging the output of the init script themselves, of course.

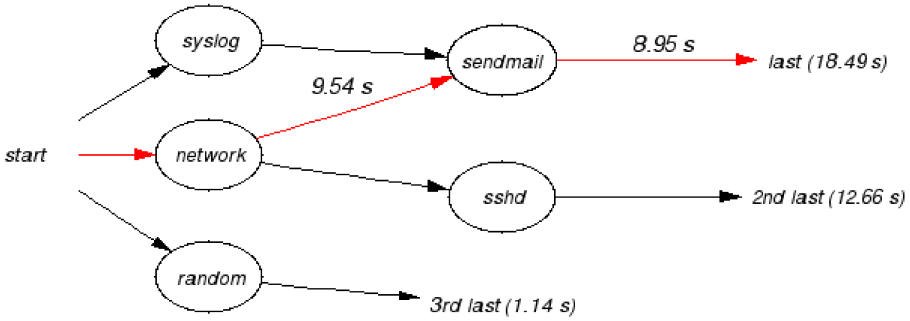

Figure 2: Serel dependency graph visualization

force-reload: cause the configuration to be reloaded if the service supports this and it is running. Otherwise, restart the service.

status: print the current status of the service.

These definitions are completely compatible with the definitions in Debian policy[14]. Actions *start*, *stop*, *restart* and *force-reload* are required to be supported by all init scripts, both in the LSB and by Debian policy.

The LSB requires *status*[32] to be supported by all init scripts, while Debian policy doesn't. It also provides a bunch of shell helper functions in */lib/lsb/init-functions* that init scripts are supposed to (but not forced to) use.

It has a few other requirements, which are also in Debian policy:

- init scripts must ensure that they will behave sensibly if invoked with action *start* when the service is already running, or with *stop* when it isn't;
- if a service reloads its configuration automatically (as in the case of *cron*, for example), the reload option of the init script must behave as if the configuration has been reloaded successfully.

And a few requirements that are not in Debian policy, but should be:

- since an init script may be run manually by a system administrator with non-standard environment variable values for *PATH*, *USER*, *LOGNAME*, etc. init scripts must not depend on the values of these environment variables. They should set them to some known/default values if they are needed;
- all error messages must be printed on *stderr*. All status messages must be printed on *stdout*.

[32] which isn't easy to implement correctly, and not very useful.

Status	Description
0	Action successful. Also, starting an already-running service, and stopping a service that is not running
1	Generic or unspecified error
2	Syntax error
3	Unimplemented feature
4	User has insufficient privilege
5	Service is not installed
6	Service is not configured
7	Service is not running
8–99	Reserved for future LSB use
100–149	Reserved for distribution use
150–199	Reserved for application use
200–254	Reserved

Table 1: LSB init script exit status codes

The LSB defines the exit status codes for init scripts, as listed on table 1. Exit status 5 ("service is not installed") is incompatible with Debian policy. Debian practice is to return exit status 0 without doing anything, when a package is removed but not purged from the system.

The *invoke-rc.d* interface was engineered to be compatible with the LSB specification.

Status	Description
0	Service is running, and ok
1	Service is dead, and */var/run pidfile* still exists
2	Service is dead, and */var/lock* file still exists
3	Service is stopped
4	Service status unknown
5–99	Reserved for future LSB use
100–149	Reserved for distribution use
150–199	Reserved for application use
200–254	Reserved

Table 2: LSB exit status codes for *status*

The LSB defines the exit status codes for the *status* action, summarized in table 2.

Debian policy doesn't say anything about the exit status for init scripts.

The LSB has also defines the runlevels*:* runlevels 0, 1 and 6 follow the System V runlevels (system halt, single-user and system reboot); runlevel 2 is "multiuser with no network services exported"; runlevel 3 is "normal, full multiuser"; runlevel 4 is "reserved for local use", and defaults to be the same as runlevel 3; and runlevel 5 is "multiuser with X11 display manager".

It also defines that the init script ids are in a single namespace (that includes the init scripts of the O.S. distribution), and that one should register init script ids, including the distribution ones, with LANANA[13].

Why this was done in such an intrusive way, instead of requesting a well-defined namespace for LSB init scripts (such as "vendor-*", or "lsb-*"), is beyond me. The LANANA pages have no mention of the init script namespace registry anywhere; so far, it looks like nobody is really paying attention to this part of the LSB[33].

3.1 LSB facilities for dependency-based init script systems

The LSB provides rudimentary support for dependency-based init script systems. It does not propose any such a system, though.

This support is composed of: comments in the init script themselves, that convey the dependency information; and a set of standard facilities that the init scripts may depend upon. The standard system facilities are listed in table 3. Init scripts providing other facilities should use their id for the facility name, without the leading "$".

Facility	Description
$local_fs	all local filesystems are mounted. Note that /usr and other filesystems might be remote
$network	low level networking is operational
$named	daemons which may provide hostname resolution (if present) are running
$portmap	daemons providing Sun-RPC/ONCRPC portmapping service (if present) are running
$remote_fs	all remote filesystems are mounted
$syslog	system logger is operational
$time	the system time has been set

Table 3: LSB boot time facilities

The comments are supposed to convey all extra information an init script might need to provide to other automated tools of the init script system. They have the serious inconvenience of forcing all init scripts to be shell scripts.

Their format is the shell comment character "`#`", a blank space, and the keyword, followed by colons, a space, and the value.

The comments must be delimited in the script by the lines "`### BEGIN INIT INFO`" and "`### END INIT INFO`". Distributions may define additional keywords prefixed by "`X-`" followed by the distribution name, such as "`# X-Debian-xyzzy`".

The keywords defined in LSB for dependency tracking are:

Provides: space delimited list of facilities provided by this service;

Required-Start: space delimited list of facilities required by this service to be active before it can be started;

Required-Stop: space delimited list of facilities required by this service to be active for it to successfully shutdown itself;

[33] which is a good thing, as far as Debian is concerned.

Should-Start: space delimited list of facilities that should be started if at all possible, before this service is started (suggested by LSB, not normative);

Should-Stop: space delimited list of facilities that should still be active when this service is shutdown (suggested by LSB, not normative).

Unfortunately, this approach is useless for dynamic dependency schemes such as the need(8) concept. We could build on them, though.

3.2 LSB Init Script functions

The LSB includes a library of POSIX.2 shell script functions, accessed by sourcing */lib/lsb/init-functions*. The functions can be provided as external commands, made available to the init script after sourcing the *init-functions* script. All LSB init scripts are supposed to source that file.

The full list of functions is summarized in table 4.

4 The Debian Init Script Subsystem

We have had an intermediate abstraction layer in Debian to access part of the system init script systems for a very long time.

Manipulation of the symbolic link farm of the System V init script system is tedious to code, and prone to error. Any Debian package installing a system init script would have to duplicate this code in its maintainer scripts[34]. Thus *update-rc.d* was born, long before it was needed for its abstraction capabilities.

/usr/sbin/update-rc.d is used to register a Sysvinit-like system init script with the underlying init script system. In a normal System V init script-based Debian system, that means adding to or removing symbolic links from the correct runlevel symbolic link farm.

[34] There are five maintainer scripts in deb packages: *config*, *preinst*, *prerm*, *postinst* and *postrm*. The *post** scripts run after package installation and removal. The *pre** scripts are run before installation and removal. The *config* script is run by debconf sometime before the *postinst* script (it may be run even before the *preinst* script), if it exists.

start_daemon [-f] [-n *nicelevel*] *pathname* [*args*]

This runs the specified program as a daemon. *start_daemon* will check to see if there is a program with the same name as the daemon already running. If so, it will not start another copy of the daemon unless the –f option is given. The –n option specifies a nice level. See nice(1).

killproc *basename* [*signal*]

This stops the specified program. The program is found using the algorithm given by *pidofproc*. If a signal is specified, the program is sent that signal. Otherwise, a SIGTERM followed by a SIGKILL after some number of seconds is sent.

pidofproc *basename*

This function returns one or more pid(s) for a particular daemon. If an entry is found in */var/run/basename.pid*, then that value is returned. For security reasons, LSB-compliant applications who wish to use the *pidofproc* function in their init scripts must store the pid in */var/run/basename.pid*.

log_success_msg *"message"*

This requests the distribution to print a success message. The message should be relatively short; no more than 60 characters is highly desirable.

log_failure_msg *"message"*

This requests the distribution to print a failure message. The message should be relatively short; no more than 60 characters is highly desirable.

log_warning_msg *"message"*

This requests the distribution to print a warning message. The message should be relatively short; no more than 60 characters is highly desirable.

Table 4: LSB shell function library for init scripts

When the file-rc SysV-like init script system entered Debian, *update-rc.d* was already there, and all maintainer scripts used it to register their system init scripts. So, it was a simple matter of adding a file-rc version of *update-rc.d* and everything kept working.

The init script system abstraction layer was incomplete, though. Maintainer scripts still had to directly call the system init scripts to start or stop the services during package install, removal and upgrade. At the time *update-rc.d* was added to the distribution, running the init scripts directly did not appear to have any major drawbacks. Or at least not enough of them to justify the added work of creating and deploying a complete abstraction layer: this happened way back in 1996, and Debian was quite small in those days.

But there are two major drawbacks to maintainer scripts running the system init scripts directly, in fact: services would be started automatically in package install and upgrades, even if the service should never have been started in that runlevel. And Debian could not move away from SysV-like init scripts stored in */etc/init.d* without fixing a big number of packages.

A fix for those problems is being deployed right now[15]. Two new scripts were added to DEBIAN 3.0 WOODY: *invoke-rc.d* and *policy-rc.d*. Their usage will become mandatory[35] after WOODY is officially out, so that the new abstraction layer is fully deployed when the next Debian stable release after WOODY is done.

4.1 update-rc.d, invoke-rc.d and policy-rc.d

There are two big sets of init script users in Debian: the system administrator herself, and maintainer scripts run by *dpkg*. The system administrator is supposed to know what she's doing; maintainer scripts, however, have a knack to start or stop services when they shouldn't.

This is particularly bad during system upgrades, or when installing packages inside a chroot jail. The maintainer scripts can mess around with the services no matter what the system administrator configured the init script system to do.

Requests for a way to keep unconfigured (as in not configured by the local system administrator yet) services from starting during first-time package installs are common in the Debian mailinglists, too. Many people seem to consider such behavior an unacceptable security risk, apparently.

To improve the init script abstraction layer, and empower the local system administrator to fully control how services are manipulated by the packages' maintainer scripts, the new *invoke-rc.d* script was introduced. Maintainer scripts are expected to use *invoke-rc.d* to start, stop, and otherwise manipulate system services.

invoke-rc.d can refuse to execute an action requested by a maintainer script. It will, for example, refuse to start or restart a service if that service should not be running in the current runlevel when the System V init script system is being used. It will also try to execute */usr/sbin/policy-rc.d* if it exists, and carry out the request as modified by *policy-rc.d*.

policy-rc.d is an optional program that enforces whatever system initscript policy the local system administrator might want. It can refuse action requests from *invoke-rc.d*, and it can also modify any such request to another one (e.g, you could change all *restart* action requests to *stop*) or to a group of requests which will be tried one by one until a request succeeds.

4.1.1 update-rc.d interface

update-rc.d registers an init script with the init script subsystem. It has to be called previous to any *invoke-rc.d* and *policy-rc.d* calls. While this is not currently done (or at least not enforced or checked for), the init script itself should not be executed before it is registered by *update-rc.d*.

The complete interface is described in the update-rc.d(8) manpage.

update-rc.d provides the following functionality:

[35] This will be done in the usual hacker-friendly way: A *should* clause in Debian policy, along with many bug reports, with apply-and-upload patches attached to add the required functionality in all packages affected by the change. When 90% of the packages have been converted to the new system, the Debian policy clause will be changed to a *must*.

1. **update-rc.d** [–n] [–f] *id* remove

   Unregisters the system init script *id.*

2. **update-rc.d** [–n] *id* defaults [*n1* [*n2*]]

   Register, or update the registry information for, the system init script in the default runlevel, with ordering number *n1*. If *n2* is specified, *n1* is used for the start ordering number, and *n2* for the stop ordering number. The default ordering number is 20.

3. **update-rc.d** [–n] *id* start|stop *n1 runlevel runlevel*... . start|stop *n1 runlevel runlevel*... . ...

   Register, or update the registry information for, the system init script in the specified runlevels, with ordering number *n1*, and specified action (*start* or *stop*). The supported runlevels are: 0–9 and *S* (system startup).

update-rc.d is supposed to behave in a administrator-friendly way. It will try to detect when local changes were made to that init script's runlevel information, and will not override them. For the System V link farm, this means that as long as any link exists for that service, in any runlevel, and the links do not match the last registered state, they will not be updated.

update-rc.d(8) lists the following as a bug: "There should be a way for the system administrator to specify at least the default *start* and *stop* runlevels to be used by *defaults* and possibly to override other things as well."

update-rc.d returns exit status zero if the operation is carried out, and non-zero if an error happens.

4.1.2 invoke-rc.d interface

invoke-rc.d executes a system init script, subject to whatever init script policy is active. It should be used for all interactions of a maintainer script with a system init script[36]. *invoke-rc.d* enforces only very basic policies by itself: it blocks attempts to start a service if the system is currently in a runlevel where the service would not be active. It relies on *policy-rc.d* to implement more complex policies.

The complete interface for *invoke-rc.d* is described in the invoke-rc.d(8) manpage, and also with a bit more detail in [16]. *invoke-rc.d* is provided by the Sysvinit and file-rc packages.

invoke-rc.d provides the following functionality:

1. **invoke-rc.d** [–force] [–disclose-deny] [–nofallback] *id action* [*init script parameters...*]

   Executes the system initscript *id*, requesting the given action. Parameters after *action* are sent to the init script. If execution is denied by the init script policy layer, *invoke-rc.d* will return a zero exit status unless –disclose-deny is specified.

2. **invoke-rc.d** –query [–disclose-deny] *id action* [*init script parameters...*]

   Returns exit status codes describing what would happen, but does not execute the init script. This allows one to know, for example, if the init script is registered; or if an init script would be started in the current runlevel.

Should an init script be executed, *invoke-rc.d* always returns the status code returned by the init script.

Init scripts should not return status codes in the 100+ range (which is reserved in Debian and by the LSB).

The status codes returned by *invoke-rc.d* proper are summarized in table 5. Exit status 104, 105 and 106 are only returned when in –query mode.

4.1.3 policy-rc.d interface

policy-rc.d is responsible for init script policy decisions. It arbitrates when *invoke-rc.d* should execute an action using a service init script, and how. This script is optional, and it will likely not be present on most Debian systems.

The complete interface for *policy-rc.d* is described in the policy-rc.d(8) manpage[37], and in [16].

[36] Note that the init script system itself does not use *invoke-rc.d* ever. The user shouldn't, either.

[37] This manpage has not been written yet.

Status	Description
0	Success: Either the init script was run and returned exit status 0, or it was not run because of *runlevel*/local policy constrains and –disclose-deny is not in effect.
1–99	Reserved for the init script, usually indicates a failure
100	Init script *id* unknown
101	Action not allowed: The requested action was denied
102	Subsystem error: init script subsystem malfunction
103	Syntax error
104	Action allowed: init script would be run
105	Behaviour uncertain
106	Fallback action requested

Table 5: *invoke-rc.d* exit status codes

policy-rc.d has more capabilities than just approving or denying a certain action. It is possible to execute one or more actions, instead of the action initially requested to *invoke-rc.d*. This is known in the *invoke-rc.d* and *policy-rc.d* documentation as a fallback action.

Should *policy-rc.d* reply to *invoke-rc.d* that a list of actions is to be used as a fallback action, instead of a single action, they will be executed in FIFO sequence, until one of them succeeds.

policy-rc.d provides the following functionality:

1. **policy-rc.d** [–quiet] –list *id* [*runlevel ...*]

   List, in a format friendly for parsing by humans, all policies defined for init script *id*, for the specified runlevels. If no runlevels are specified, list policies for all runlevels. All known actions and their fallback actions for the init script *id*.

2. **policy-rc.d** [–quiet] *id list-of-actions* [*runlevel*]

   Verify policy for the space-separated list of actions *list-of-actions* (the list must be sent as a single parameter to *policy-rc.d*), and the specified runlevel. Should the runlevel not be specified, it will be considered unknown.

Status	Description
0	Action allowed
1	unknown action, therefore undefined policy
100	Init script *id* unknown
101	Action forbidden
102	Subsystem error: init script subsystem malfunction
103	Syntax error
104	*reserved, do not use*
105	Behaviour uncertain, undefined policy
106	Action forbidden. Use the returned *fallback* actions instead

Table 6: *policy-rc.d* exit status codes

The exit status codes *policy-rc.d* should return are summarized in table 6.

stdin: not to be used by *policy-rc.d*.

stdout: used to return –list output, or to output a single line containing a space-separated list of fallback actions.

stderr: used to output error messages.

5 What remains to be done in Debian

Debian does not have an */sbin/init* abstraction layer. However, proper implementation of */sbin/telinit* would work just as fine as an extra wrapper would, and very few packages try to talk to *init* anyway (glibc tries to restart *init* when the libc is upgraded, but that's about it).

Configuration packages that touch */etc/inittab* might cause grief, too. But there is little that can be done about that, other than not using */etc/inittab* for the *init* config file if the syntax changes too much.

We also have no init script logging, and the init script output is loosely standardized in Debian policy[38]. The former is something we should address soon, maybe even in our implementation of

[38] It is better to have it loosely standardized than to have a strict standard nobody cares about, but we don't enforce even *stdout/stderr* consistency, which is bad.

the System V init script system, and certainly in any new systems we deploy. The later is something that should wait until proper logging is implemented, when it will make a bigger difference[39] to have consistent output.

5.1 Debian Init Script Registry

There is one thing Debian should do as soon as possible: a registry of init scripts. The main reason for such a registry would be to make sure that the System V ordering is always sane, but it would also have some extra benefits: we would know the packages that have init scripts at a glance; and we could register the init script names with LANANA should that become necessary.

Such registry could be implemented as a simple, machine-readable text file, shipped in a very small package with little else. The Debian bug-tracking system could be used to request changes in the database. An aggressive 2-day NMU[40] policy (or a big set of uploader maintainers, which amounts to the same) for this package would be enough to keep it current.

Debian policy would be changed to require all packages providing system init scripts to register them with the init script database, after all such packages currently in Debian are added to the database as an initial set.

The database probably needs to track the state of a package in both the stable and unstable distributions, to help quality assurance, and bug fixing.

Lintian[41] and Linda[42] warnings should be added also, to remind maintainers that their init scripts are not in the database yet (and to help track down typos).

5.2 Minimum command line interface specification: */sbin/init*, */sbin/telinit*

We need to define a minimum subset of Sysvinit */sbin/telinit*'s command line interface that any new */sbin/init* must implement.

Either that, or we will need an abstraction layer to deal with */sbin/init*. It is probably better to just define a compatibility set for */sbin/init* and */sbin/telinit*, much in the same way as it is done for */usr/sbin/sendmail*:

1. **/sbin/init** *runlevel*

   /sbin/telinit *runlevel*

   Switches to the specified runlevel for runlevel-based init script systems. Starts the runlevel-named facility for dependency-based init script systems.

   During system bootstrap, */sbin/init* may be executed by the kernel with the *auto* keyword appended to the command line, with or without a runlevel. It must be able to deal with that situation gracefully.

2. **/sbin/telinit** [–t *seconds*] [u|U] [q|Q]

   The –t option sets the time that */sbin/init* should wait between sending processes the SIGTERM and SIGKILL signals. It is ignored if */sbin/init* has no such capability.

   The fake runlevel "u" (or "U") tells */sbin/init* to re-execute itself. This functionality is **required**, and it must not cause *init* to lose state.

   The fake runlevel "q" (or "Q") tells */sbin/init* to reload its configuration. This functionality is **optional**, but if it is not implemented, *telinit* must ignore it instead of returning an error.

5.3 Logging, and uniform output

Init scripts are either managed directly by */sbin/init*, or by an special script. There is little reason for why we could not log their standard output (*stdout*) and standard error (*stderr*) streams to a ring-buffer during early system bootstrap, and dump that to a log file later[43].

Any new, non-minimal init script system should attempt to do proper logging of the init scripts' output. And this output should be made concise, clear in its intent, and as uniform as possible.

All normal init script output should be sent to *stdout*. All error output should be sent to *stderr*. The service must not pollute[44] the console. All

[39] Which is to say: "when we will have more chances to get maintainers to actually fix the damn things".
[40] Non-Maintainer Upload.
[41] Lintian is an automated tool used to verify common mistakes in Debian packages.
[42] An ongoing rewrite of Lintian in *Python*.
[43] Other than the fact that nobody has taken the time to write the code to do so, thus far.
[44] Pollute as in "write to, or read directly from".

output must be to the *stdout* or *stderr* pipes. Once the init script finishes its run, these pipes will be closed. The service (which might be a daemon) must be able to cope with that gracefully[45].

5.4 Extending the abstraction layer for dependency-based systems

Adding a new, dependency-capable, parallel-execution-capable init script system to Debian is an interesting challenge. Deploying such a system will require a few updates in the current init script system abstraction layer, however.

Such changes are needed so that we can add support for generic, dependency-based systems. Once they are in place, Debian will be able to host different dependency-based systems as well as our current capability of hosting different numerical-order-based init script systems.

The following improvements to Debian need to be deployed to support dependency-based systems correctly:

1. Some runlevels will have to be removed from the abstraction layer, or to be made optional.

   In dependency-based init script systems, runlevels are not needed and often not desired. It is much easier to simply use dependencies to create as many different named sets of init scripts as one wants.

   As far as current Debian policy is concerned, runlevels are already reduced to: system startup (S), single-user (1), multi-user (2–5), and shutdown (6). It is up to the local system administrator to customize runlevels 2–5 to his preferences.

   It is useful to retain these four sets of init scripts, but we should make sure developers are not mislead into thinking they are actually allowed to have Debian packages trying to use runlevels with more granularity than the four sets above. One easy way to do that is to change *update-rc.d* so that only the four-runlevel set is supported for Debian packages.

   It is important to note that the limitation in the number and type of runlevels allowed for manipulation in Debian maintainer scripts must not impact the local system administrator. If an init script system allows for the System V runlevels, those must be fully available for customization by the local system administrator. If an init script system supports unlimited named runlevels, those too must be fully available for customization by the local system administrator.

   Whether we want the full runlevel set to be available through *update-rc.d* to local administrators, or not, is the important question. Missuse of *update-rc.d* by users is a known problem[46].

2. The system startup and shutdown runlevels should be unified.

   The current split system (runlevels S, 0 and 6) we have is fine, and it will work well as long as the new init script systems special-case system shutdowns and system startup.

   However, the split system is not as clean as, nor as simple to understand as an unified system initialization, where the *start* action sent to the system initialization scripts during system startup, and the *stop* action is sent during system shutdown. We would drop runlevels 0 and 6 from the abstraction layer, then.

   Given the already confusing state of affairs regarding the start and stop actions in runlevels 0 and 6[47], it is arguable that the change is a price worth paying. We simply will not have any *S* scripts receiving *stop* actions in the shutdown runlevels anymore, so it won't be terribly confusing to people used to Sysvinit, anyway.

[45] A properly coded daemon must do the following: Close file descriptors 0–2 (*stdin, stdout, stderr*), start another session group to detach from the controlling terminal, and fork. Broken ones must be fixed, or worked around using *start-stop-daemon*.

[46] The local administrator should deal directly with the init script system, and not go through Debian's abstraction layer. Otherwise, his changes may be lost.

[47] Scripts that stop services must be registered as *K* (*stop*) scripts, while functionality exclusive to runlevels 0 and 6 must be registered as *S* (*start*) scripts, in System V init script speech. The *S* scripts on runlevels 0 and 6 will be executed with the *stop* action, which is misleading at best.

The system init scripts would use an environment variable to differentiate system reboot from system shutdown.

3. There needs to be a registry of virtual facilities, just like the one we have for virtual packages.

   This registry would be yet another small text file inside the package containing the init script registry.

4. A new init script action defined to be "restart only if the service is currently active" is needed[48].

   Without such functionality, it is very difficult, if not impossible, to implement dynamic dependencies where restarting a service causes a few others to be restarted. Such functionality is desired for the syslog service, for example.

5. We will have to extend the *update-rc.d* interface (or create a new one) to support static dependencies in init script systems.

   We need some way to store the new dependency information for init script systems with static dependencies. There are a number of alternatives, such as:

   a) Store it in file fragments, inside an */etc/init.d*-like directory;
   b) Store it inside the init script itself, as comments (see sections 2.3 and 3 for systems that use this);
   c) Store it inside the init script itself, as a call to an external command (be it a new *update-rc.d* interface, or something else);
   d) Store it at package installation time, just like it is done for *update-rc.d* in Sysvinit.

   My personal preference goes to alternative 5c. It keeps the information inside the init script itself, an easy place for the user to look it up and modify it, while at the same time not limiting the comment syntax for the init script (i.e. one can still have init scripts in any language allowed by Debian policy).

6. The trigger for executing S and K init scripts could change.

   In Sysvinit, these init scripts are executed when a runlevel is entered. It is likely that for dependency-based systems, S init scripts would be executed when a runlevel is entered, and K init scripts when exiting the runlevel.

   This is not a huge problem, but care needs to be taken to properly document such changes in behavior since they are likely to break custom-made runlevel configurations done by the local system administrator.

Any init script system to be hosted in Debian would need to implement the following prerequisites:

1. SysV init-like init script interface, accessible through *invoke-rc.d*;
2. Sysvinit-like subset of the *telinit* and *init* interface, as described in section 5.2;
3. Full implementation of the entire Debian init script abstraction layer.

It is possible to keep the System V full runlevel set, but we would probably benefit from simpler systems without them.

5.4.1 Standard interface

To avoid namespace pollution, it would be better to avoid general use names such as "need", and "provide" for the init script system functions and executables. Also, we should define a superset of the interfaces needed by the various possible dependency-based and non-dependency-based SysV init-like systems.

The */usr/sbin/update-rc.d* interface needs to have its behavior defined for non-order-based systems as well. Dependency-based systems should use the order information provided by *update-rc.d* to enforce strict ordering on legacy init scripts, or it must use package conflicts to avoid problems with legacy init script packages.

The standard interface to communicate dependency information to the init script system needs

[48] A possible name for this action is *restart-if-running*.

to be done through external commands instead of shell functions, so that we support non-shell init scripts in systems that support dynamic configuration of dependencies.

A proposal for such an interface is described in table 7. An init script may assume a dependency-based system is available if */sbin/init-after* exists, and it is executable. Otherwise, the init script must either assume that an order-based system (such as the standard System V init script system) is being used (and all dependencies are already satisfied at runtime), or it must refuse to work, reporting exit status 102 ("init script subsystem malfunction").

Init script conflicts and dependency failures must not be allowed to happen, and the packaging system dependency system is to be used to guarantee that. Init scripts must be written in a defensive coding style, and do all the proper error checking.

Dependency-based init script systems should refuse to start any init scripts that it cannot guarantee the dependencies requirements for. It should halt system initialization and rollback to either single-user mode or to an emergency mode (if single-user mode cannot be reached for some reason), using */sbin/sulogin* to allow the local system administrator to fix any problems.

The init script system must not deadlock. Loop and deadlock detection and breaking safeties must be implemented, if those situations are possible (and they usually are in dependency-based systems).

The proposed interface has one big shortcoming: it relies on *init* (or another part of the init script subsystem) to be able to figure out the pid of the process calling the */sbin/init-** commands. This is a non-trivial assumption, which may prove to be impossible to implement because:

- the */sbin/init-** commands might be run by a child process of the init script, and that is a desirable ability we should not forbid;
- the init script may be run by the local administrator directly, outside of a *invoke-rc.d* or runlevel change context. It is desirable that the dependency system is not disabled for such cases;

1. **/sbin/init-provide** *keyword* [*keyword...*]

   Declares that the init script also provides the facilities identified by the given keywords, in addition to its own id.

   This command will not block, and returns immediately. It may not be called after *init-before* or *init-after* were used.

2. **/sbin/init-before** *keyword* [*keyword...*]

   Declares a dependency where the current init script must be run sometime before the facilities identified by the given keywords are available. Strict ordering must be enforced.

   Not all dependency-based init scripts will implement this functionality, and in that case, init-before should always fail. This command may block in dynamic systems. It may not be called after *init-after* have been used.

3. **/sbin/init-after** *keyword* [*keyword...*]

   Declares a dependency where the current init script must be run sometime after the facilities identified by the given keywords are available.

   In dynamic systems, this command will block and return with exit status 0 after the dependencies are satisfied, or with non-zero exit status if an error happens (in which case the dependencies are **not** guaranteed to be satisfied, and the init script should fail).

4. **/sbin/init-test** *keyword* [*keyword...*]

   Returns exit status 0 if the facilities named by the keywords are all available, exit status 1 if they are not available yet, and exit status 2 if any of them will never be available (this is an optional feature).

Table 7: init script interface for dependency-aware systems

An easy solution to these problems is to add a required argument to *init-provide*, *init-before* and *init-after* that specifies the pid of the process that should be watched. These issues, and the best way to address them, are open for discussion.

References

[1] Pape, Gerrit; *runit - minimal replacement for Sysvinit*; `http://smarden.org/runit/`

[2] von Leitner, Felix; *minit - a small yet feature-complete init*; `http://www.fefe.de/minit/`

[3] Gooch, Richard; *Linux Boot Scripts*; 2002-05-31; `http://www.atnf.csiro.au/people/rgooch/linux/boot-scripts/`

[4] proton@energymech.net; *Bizarre Sources*; `http://www.energymech.net/users/proton/`

[5] Mewburn, Luke; *The Design and Implementation of the NetBSD rc.d system*; Wasabi Systems, Usenix Annual Technical Conference, June 2001; `http://www.mewburn.net/luke/papers/rc.d.pdf`

[6] van Smoorenburg, Miquel; manpage *INIT(8): init, telinit - process control initialization*; Debian Sysvinit 2.84-3 package

[7] Fremlin, John; *jinit* ; `http://homepage.ntlworld.com/john.fremlin/programs/linux/jinit/index.html`

[8] Andersen, Erik & others; *Busybox*; `http://www.busybox.net/`

[9] Trümper, Winfried; Lees, Tom; Schulze, Martin & Rosenfeld, Roland; *file-rc: Alternative boot mechanism using a single configuration file*; `http://packages.debian.org/file-rc`

[10] von Leitner, Felix; *diet libc - a libc optimized for small size*; `http://www.fefe.de/dietlibc/`

[11] Mayo, Leni; *Serel - fast boot software*; `http://www.fastboot.org`

[12] Free Standards Group; *Linux Standard Base, common specification, Chapter 22 - System Initialization*; `http://www.linuxbase.org/`

[13] *LANANA - The Linux Assigned Names And Numbers Authority*; `http://www.lanana.org/`

[14] Jackson, Ian; Schwarz, Christian; Debian Policy ML; *Debian Policy*; Release 3.5.6.1, 2002-03-14

[15] Holschuh, Henrique de Moraes; *Debian Bug report logs - #76868 [PENDING AMENDMENT 2001-02-27] invoke-rc.d interface to invoke initscripts*; Debian bug tracking system; `http://bugs.debian.org/76868`

[16] Holschuh, Henrique de Moraes; *Invoke-rc.d/policy-rc.d interfaces*; Draft; `http://people.debian.org/~hmh/invokerc.d-policyrc.d-specification.txt`

[17] W3C; *Resource Description Framework (RDF)*; `http://www.w3.org/RDF/`

6 Revision Log

Revision 1.16

Added fake runlevel u/U to telinit interface; fixed semantics for fake runlevel q/Q; added footnote with sysvinit's sans-glibc size; fixed mistake regarding the internal working of runlevel S in Sysvinit (it does not execute any scripts, ever. It is not silently converted into a switch to runlevel 1); clarified the issue of runlevels 0 and 6 (system shutdown); and added paragraph 6 to section 5.4.

Revision 1.15 (2002-06-30 22:15:20)

Fixes for proper PDF output using PDFLaTeX.