systemd for Administrators, Part 1

As many of you know, systemd is the new Fedora init system, starting with F14, and it is also on its way to being adopted in a number of other distributions as well (for example,OpenSUSE). For administrators systemd provides a variety of new features and changes and enhances the administrative process substantially. This blog story is the first part of a series of articles I plan to post roughly every week for the next months. In every post I will try to explain one new feature of systemd. Many of these features are small and simple, so these stories should be interesting to a broader audience. However, from time to time we'll dive a little bit deeper into the great new features systemd provides you with.

Verifying Bootup

Traditionally, when booting up a Linux system, you see a lot of little messages passing by on your screen. As we work on speeding up and parallelizing the boot process these messages are becoming visible for a shorter and shorter time only and be less and less readable -- if they are shown at all, given we use graphical boot splash technology like Plymouth these days. Nonetheless the information of the boot screens was and still is very relevant, because it shows you for each service that is being started as part of bootup, wether it managed to start up successfully or failed (with those green or red [OK] or [FAILED] indicators). To improve the situation for machines that boot up fast and parallelized and to make this information more nicely available during runtime, we added a feature to systemd that tracks and remembers for each service whether it started up successfully, whether it exited with a non-zero exit code, whether it timed out, or whether it terminated abnormally (by segfaulting or similar), both during start-up and runtime. By simply typing systemctl in your shell you can query the state of all services, both systemd native and SysV/LSB services:

```
[root@lambda] ~# systemctl
UNIT                                LOAD   ACTIVE       SUB         JOB           DESCRIPTION
dev-hugepages.automount                  loaded active       running                 Huge Pages File System Automount Point
dev-mqueue.automount                     loaded active       running                 POSIX Message Queue File System
Automount Point
proc-sys-fs-binfmt_misc.automount        loaded active       waiting                 Arbitrary Executable File Formats File
System Automount Point
sys-kernel-debug.automount               loaded active       waiting                 Debug File System Automount Point
sys-kernel-security.automount            loaded active       waiting                 Security File System Automount Point
sys-devices-pc...0000:02:00.0-net-eth0.device loaded active      plugged                 82573L Gigabit Ethernet Controller
[...]
sys-devices-virtual-tty-tty9.device      loaded active       plugged                 /sys/devices/virtual/tty/tty9
-.mount                             loaded active       mounted                 /
boot.mount                          loaded active       mounted                 /boot
dev-hugepages.mount                      loaded active       mounted                 Huge Pages File System
dev-mqueue.mount                         loaded active       mounted                 POSIX Message Queue File System
home.mount                          loaded active       mounted                 /home
proc-sys-fs-binfmt_misc.mount            loaded active       mounted                 Arbitrary Executable File Formats File
System
abrtd.service                       loaded active       running                 ABRT Automated Bug Reporting Tool
accounts-daemon.service                  loaded active       running                 Accounts Service
acpid.service                       loaded active       running                 ACPI Event Daemon
atd.service                         loaded active       running                 Execution Queue Daemon
auditd.service                      loaded active       running                 Security Auditing Service
avahi-daemon.service                     loaded active       running                 Avahi mDNS/DNS-SD Stack
bluetooth.service                   loaded active       running                 Bluetooth Manager
console-kit-daemon.service               loaded active       running                 Console Manager
cpuspeed.service                    loaded active       exited                  LSB: processor frequency scaling support
crond.service                       loaded active       running                 Command Scheduler
```

```
cups.service                                 loaded active       running                 CUPS Printing Service
dbus.service                                 loaded active       running                 D-Bus System Message Bus
getty@tty2.service                             loaded active       running                   Getty on tty2
getty@tty3.service                             loaded active       running                   Getty on tty3
getty@tty4.service                             loaded active       running                   Getty on tty4
getty@tty5.service                             loaded active       running                   Getty on tty5
getty@tty6.service                             loaded active       running                   Getty on tty6
haldaemon.service                              loaded active       running                   Hardware Manager
hdapsd@sda.service                               loaded active       running                     sda shock protection daemon
irqbalance.service                           loaded active       running                   LSB: start and stop irqbalance daemon
iscsi.service                              loaded active       exited                  LSB: Starts and stops login and scanning of iSCSI
devices.
iscsid.service                               loaded active       exited                  LSB: Starts and stops login iSCSI daemon.
livesys-late.service                           loaded active       exited                    LSB: Late init script for live image.
livesys.service                              loaded active       exited                  LSB: Init script for live image.
lvm2-monitor.service                           loaded active       exited                    LSB: Monitoring of LVM2 mirrors, snapshots
etc. using dmeventd or progress polling
mdmonitor.service                              loaded active       running                   LSB: Start and stop the MD software RAID
monitor
modem-manager.service                          loaded active       running                   Modem Manager
netfs.service                              loaded active       exited                  LSB: Mount and unmount network filesystems.
NetworkManager.service                         loaded active       running                   Network Manager
ntpd.service                               loaded maintenance  maintenance               Network Time Service
polkitd.service                              loaded active       running                 Policy Manager
prefdm.service                               loaded active       running                 Display Manager
rc-local.service                             loaded active       exited                  /etc/rc.local Compatibility
rpcbind.service                              loaded active       running                 RPC Portmapper Service
rsyslog.service                              loaded active       running                 System Logging Service
rtkit-daemon.service                           loaded active       running                   RealtimeKit Scheduling Policy Service
sendmail.service                             loaded active       running                 LSB: start and stop sendmail
sshd@172.31.0.53:22-172.31.0.4:36368.service  loaded active       running                 SSH Per-Connection Server
sysinit.service                              loaded active       running                 System Initialization
systemd-logger.service                         loaded active       running                   systemd Logging Daemon
udev-post.service                            loaded active       exited                  LSB: Moves the generated persistent udev rules to
/etc/udev/rules.d
udisks.service                               loaded active       running                 Disk Manager
upowerd.service                              loaded active       running                 Power Manager
wpa_supplicant.service                         loaded active       running                   Wi-Fi Security Service
avahi-daemon.socket                            loaded active       listening                 Avahi mDNS/DNS-SD Stack Activation Socket
cups.socket                                  loaded active       listening               CUPS Printing Service Sockets
dbus.socket                                  loaded active       running                 dbus.socket
rpcbind.socket                               loaded active       listening               RPC Portmapper Socket
sshd.socket                                loaded active       listening               sshd.socket
systemd-initctl.socket                         loaded active       listening                 systemd /dev/initctl Compatibility Socket
systemd-logger.socket                          loaded active       running                   systemd Logging Socket
systemd-shutdownd.socket                       loaded active       listening                 systemd Delayed Shutdown Socket
dev-disk-by\x1...x1db22a\x1d870f1adf2732.swap loaded active       active                  /dev/disk/by-uuid/fd626ef7-34a4-
4958-b22a-870f1adf2732
basic.target                               loaded active       active                  Basic System
bluetooth.target                             loaded active       active                    Bluetooth
dbus.target                                loaded active       active                  D-Bus
getty.target                               loaded active       active                  Login Prompts
graphical.target                             loaded active       active                    Graphical Interface
local-fs.target                            loaded active       active                  Local File Systems
multi-user.target                            loaded active       active                    Multi-User
network.target                               loaded active       active                    Network
remote-fs.target                             loaded active       active                    Remote File Systems
sockets.target                             loaded active       active                  Sockets
```

```
swap.target                 loaded active      active                Swap
sysinit.target              loaded active      active                System Initialization

LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.
JOB    = Pending job for the unit.

221 units listed. Pass --all to see inactive units, too.
[root@lambda] ~#
```

(I have shortened the output above a little, and removed a few lines not relevant for this blog post.)

Look at the ACTIVE column, which shows you the high-level state of a service (or in fact of any kind of unit systemd maintains, which can be more than just services, but we'll have a look on this in a later blog posting), whether it is *active* (i.e. running), *inactive* (i.e. not running) or in any other state. If you look closely you'll see one item in the list that is marked *maintenance* and highlighted in red. This informs you about a service that failed to run or otherwise encountered a problem. In this case this is ntpd. Now, let's find out what actually happened to ntpd, with the systemctl status command:

```
[root@lambda] ~# systemctl status ntpd.service
ntpd.service - Network Time Service
                  Loaded: loaded (/etc/systemd/system/ntpd.service)
                  Active: maintenance
                     Main: 953 (code=exited, status=255)
                  CGroup: name=systemd:/systemd-1/ntpd.service
[root@lambda] ~#
```

This shows us that NTP terminated during runtime (when it ran as PID 953), and tells us exactly the error condition: the process exited with an exit status of 255.

In a later systemd version, we plan to hook this up to ABRT, as soon as this enhancement request is fixed. Then, if systemctl status shows you information about a service that crashed it will direct you right-away to the appropriate crash dump in ABRT.

Summary: use systemctl and systemctl status as modern, more complete replacements for the traditional boot-up status messages of SysV services. systemctl statusnot only captures in more detail the error condition but also shows runtime errors in addition to start-up errors.

systemd for Administrators, Part II

Which Service Owns Which Processes?

On most Linux systems the number of processes that are running by default is substantial. Knowing which process does what and where it belongs to becomes increasingly difficult. Some services even maintain a couple of worker processes which clutter the "ps" output with many additional processes that are often not easy to recognize. This is further complicated if daemons spawn arbitrary 3rd-party processes, as Apache does with CGI processes, or cron does with user jobs.

A slight remedy for this is often the process inheritance tree, as shown by "ps xaf". However this is usually not reliable, as processes whose parents die get reparented to PID 1, and hence all information about inheritance gets lost. If a process "double forks" it hence loses its relationships to the processes that started it. (This actually is supposed to be a feature and is relied on for the traditional Unix daemonizing logic.) Furthermore processes can freely change their names with PR_SETNAME or by patching argv[0], thus making it harder to recognize them. In fact they can play hide-and-seek with the administrator pretty nicely this way.

In systemd we place every process that is spawned in a *control group* named after its service. Control groups (or *cgroups*) at their most basic are simply groups of processes that can be arranged in a hierarchy and labelled individually. When processes spawn other processes these children are automatically made members of the parents cgroup. Leaving a cgroup is not possible for unprivileged processes. Thus, cgroups can be used as an effective way to label processes after the service they belong to and be sure that the service cannot escape from the label, regardless how often it forks or renames itself. Furthermore this can be used to safely kill a service and all processes it created, again with no chance of escaping.

In today's installment I want to introduce you to two commands you may use to relate systemd services and processes. The first one, is the well known ps command which has been updated to show cgroup information along the other process details. And this is how it looks:

```
$ ps xawf -eo pid,user,cgroup,args
    PID USER     CGROUP                  COMMAND
      2 root     -                       [kthreadd]
      3 root     -                        \_ [ksoftirqd/0]
[...]
   4281 root     -                       \_ [flush-8:0]
      1 root     name=systemd:/systemd-1            /sbin/init
    455 root     name=systemd:/systemd-1/sysinit.service /sbin/udevd -d
28188 root     name=systemd:/systemd-1/sysinit.service  \_ /sbin/udevd -d
28191 root     name=systemd:/systemd-1/sysinit.service  \_ /sbin/udevd -d
   1096 dbus     name=systemd:/systemd-1/dbus.service /bin/dbus-daemon --system --address=systemd: --nofork --systemd-
activation
   1131 root     name=systemd:/systemd-1/auditd.service auditd
   1133 root     name=systemd:/systemd-1/auditd.service  \_ /sbin/audispd
   1135 root     name=systemd:/systemd-1/auditd.service      \_ /usr/sbin/sedispatch
   1171 root     name=systemd:/systemd-1/NetworkManager.service /usr/sbin/NetworkManager --no-daemon
   4028 root     name=systemd:/systemd-1/NetworkManager.service  \_ /sbin/dhclient -d -4 -sf /usr/libexec/nm-dhcp-
client.action -pf /var/run/dhclient-wlan0.pid -lf /var/lib/dhclient/dhclient-7d32a784-ede9-4cf6-9ee3-60edc0bce5ff-
wlan0.lease -
   1175 avahi    name=systemd:/systemd-1/avahi-daemon.service avahi-daemon: running [epsilon.local]
   1194 avahi    name=systemd:/systemd-1/avahi-daemon.service  \_ avahi-daemon: chroot helper
   1193 root     name=systemd:/systemd-1/rsyslog.service /sbin/rsyslogd -c 4
   1195 root     name=systemd:/systemd-1/cups.service cupsd -C /etc/cups/cupsd.conf
```

```
  1207 root     name=systemd:/systemd-1/mdmonitor.service mdadm --monitor --scan -f --pid-
file=/var/run/mdadm/mdadm.pid
  1210 root     name=systemd:/systemd-1/irqbalance.service irqbalance
  1216 root     name=systemd:/systemd-1/dbus.service /usr/sbin/modem-manager
  1219 root     name=systemd:/systemd-1/dbus.service /usr/libexec/polkit-1/polkitd
  1242 root     name=systemd:/systemd-1/dbus.service /usr/sbin/wpa_supplicant -c /etc/wpa_supplicant/wpa_supplicant.conf
-B -u -f /var/log/wpa_supplicant.log -P /var/run/wpa_supplicant.pid
  1249 68       name=systemd:/systemd-1/haldaemon.service hald
  1250 root     name=systemd:/systemd-1/haldaemon.service  \_ hald-runner
  1273 root     name=systemd:/systemd-1/haldaemon.service      \_ hald-addon-input: Listening on /dev/input/event3
/dev/input/event9 /dev/input/event1 /dev/input/event7 /dev/input/event2 /dev/input/event0 /dev/input/event8
  1275 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-rfkill-killswitch
  1284 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-leds
  1285 root     name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-generic-backlight
  1287 68       name=systemd:/systemd-1/haldaemon.service      \_ /usr/libexec/hald-addon-acpi
  1317 root     name=systemd:/systemd-1/abrtd.service /usr/sbin/abrtd -d -s
  1332 root     name=systemd:/systemd-1/getty@.service/tty2 /sbin/mingetty tty2
  1339 root     name=systemd:/systemd-1/getty@.service/tty3 /sbin/mingetty tty3
  1342 root     name=systemd:/systemd-1/getty@.service/tty5 /sbin/mingetty tty5
  1343 root     name=systemd:/systemd-1/getty@.service/tty4 /sbin/mingetty tty4
  1344 root     name=systemd:/systemd-1/crond.service crond
  1346 root     name=systemd:/systemd-1/getty@.service/tty6 /sbin/mingetty tty6
  1362 root     name=systemd:/systemd-1/sshd.service /usr/sbin/sshd
  1376 root     name=systemd:/systemd-1/prefdm.service /usr/sbin/gdm-binary -nodaemon
  1391 root     name=systemd:/systemd-1/prefdm.service  \_ /usr/libexec/gdm-simple-slave --display-id
/org/gnome/DisplayManager/Display1 --force-active-vt
  1394 root     name=systemd:/systemd-1/prefdm.service      \_ /usr/bin/Xorg :0 -nr -verbose -auth /var/run/gdm/auth-for-
gdm-f2KUOh/database -nolisten tcp vt1
  1495 root     name=systemd:/user/lennart/1            \_ pam: gdm-password
  1521 lennart  name=systemd:/user/lennart/1                \_ gnome-session
  1621 lennart  name=systemd:/user/lennart/1                   \_ metacity
  1635 lennart  name=systemd:/user/lennart/1                   \_ gnome-panel
  1638 lennart  name=systemd:/user/lennart/1                   \_ nautilus
  1640 lennart  name=systemd:/user/lennart/1                   \_ /usr/libexec/polkit-gnome-authentication-agent-1
  1641 lennart  name=systemd:/user/lennart/1                   \_ /usr/bin/seapplet
  1644 lennart  name=systemd:/user/lennart/1                   \_ gnome-volume-control-applet
  1646 lennart  name=systemd:/user/lennart/1                   \_ /usr/sbin/restorecond -u
  1652 lennart  name=systemd:/user/lennart/1                   \_ /usr/bin/devilspie
  1662 lennart  name=systemd:/user/lennart/1                   \_ nm-applet --sm-disable
  1664 lennart  name=systemd:/user/lennart/1                   \_ gnome-power-manager
  1665 lennart  name=systemd:/user/lennart/1                   \_ /usr/libexec/gdu-notification-daemon
  1670 lennart  name=systemd:/user/lennart/1                   \_ /usr/libexec/evolution/2.32/evolution-alarm-notify
  1672 lennart  name=systemd:/user/lennart/1                   \_ /usr/bin/python /usr/share/system-config-printer/applet.py
  1674 lennart  name=systemd:/user/lennart/1                   \_ /usr/lib64/deja-dup/deja-dup-monitor
  1675 lennart  name=systemd:/user/lennart/1                   \_ abrt-applet
  1677 lennart  name=systemd:/user/lennart/1                   \_ bluetooth-applet
  1678 lennart  name=systemd:/user/lennart/1                   \_ gpk-update-icon
  1408 root     name=systemd:/systemd-1/console-kit-daemon.service /usr/sbin/console-kit-daemon --no-daemon
  1419 gdm      name=systemd:/systemd-1/prefdm.service /usr/bin/dbus-launch --exit-with-session
  1453 root     name=systemd:/systemd-1/dbus.service /usr/libexec/upowerd
  1473 rtkit    name=systemd:/systemd-1/rtkit-daemon.service /usr/libexec/rtkit-daemon
  1496 root     name=systemd:/systemd-1/accounts-daemon.service /usr/libexec/accounts-daemon
  1499 root     name=systemd:/systemd-1/systemd-logger.service /lib/systemd/systemd-logger
  1511 lennart  name=systemd:/systemd-1/prefdm.service /usr/bin/gnome-keyring-daemon --daemonize --login
  1534 lennart  name=systemd:/user/lennart/1       dbus-launch --sh-syntax --exit-with-session
  1535 lennart  name=systemd:/user/lennart/1       /bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
  1603 lennart  name=systemd:/user/lennart/1       /usr/libexec/gconfd-2
  1612 lennart  name=systemd:/user/lennart/1       /usr/libexec/gnome-settings-daemon
```

```
 1615 lennart  name=systemd:/user/lennart/1       /usr/libexec/gvfsd
 1626 lennart  name=systemd:/user/lennart/1       /usr/libexec//gvfs-fuse-daemon /home/lennart/.gvfs
 1634 lennart  name=systemd:/user/lennart/1       /usr/bin/pulseaudio --start --log-target=syslog
 1649 lennart  name=systemd:/user/lennart/1        \_ /usr/libexec/pulse/gconf-helper
 1645 lennart  name=systemd:/user/lennart/1       /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=24
 1668 lennart  name=systemd:/user/lennart/1       /usr/libexec/im-settings-daemon
 1701 lennart  name=systemd:/user/lennart/1       /usr/libexec/gvfs-gdu-volume-monitor
 1707 lennart  name=systemd:/user/lennart/1       /usr/bin/gnote --panel-applet --oaf-activate-
iid=OAFIID:GnoteApplet_Factory --oaf-ior-fd=22
 1725 lennart  name=systemd:/user/lennart/1       /usr/libexec/clock-applet
 1727 lennart  name=systemd:/user/lennart/1       /usr/libexec/wnck-applet
 1729 lennart  name=systemd:/user/lennart/1       /usr/libexec/notification-area-applet
 1733 root     name=systemd:/systemd-1/dbus.service /usr/libexec/udisks-daemon
 1747 root     name=systemd:/systemd-1/dbus.service  \_ udisks-daemon: polling /dev/sr0
 1759 lennart  name=systemd:/user/lennart/1       gnome-screensaver
 1780 lennart  name=systemd:/user/lennart/1       /usr/libexec/gvfsd-trash --spawner :1.9 /org/gtk/gvfs/exec_spaw/0
 1864 lennart  name=systemd:/user/lennart/1       /usr/libexec/gvfs-afc-volume-monitor
 1874 lennart  name=systemd:/user/lennart/1       /usr/libexec/gconf-im-settings-daemon
 1903 lennart  name=systemd:/user/lennart/1       /usr/libexec/gvfsd-burn --spawner :1.9 /org/gtk/gvfs/exec_spaw/1
 1909 lennart  name=systemd:/user/lennart/1       gnome-terminal
 1913 lennart  name=systemd:/user/lennart/1        \_ gnome-pty-helper
 1914 lennart  name=systemd:/user/lennart/1        \_ bash
29231 lennart  name=systemd:/user/lennart/1        |  \_ ssh tango
 2221 lennart  name=systemd:/user/lennart/1        \_ bash
 4193 lennart  name=systemd:/user/lennart/1        |  \_ ssh tango
 2461 lennart  name=systemd:/user/lennart/1        \_ bash
29219 lennart  name=systemd:/user/lennart/1        |  \_ emacs systemd-for-admins-1.txt
15113 lennart  name=systemd:/user/lennart/1        \_ bash
27251 lennart  name=systemd:/user/lennart/1           \_ empathy
29504 lennart  name=systemd:/user/lennart/1           \_ ps xawf -eo pid,user,cgroup,args
 1968 lennart  name=systemd:/user/lennart/1       ssh-agent
 1994 lennart  name=systemd:/user/lennart/1       gpg-agent --daemon --write-env-file
18679 lennart  name=systemd:/user/lennart/1       /bin/sh /usr/lib64/firefox-3.6/run-mozilla.sh /usr/lib64/firefox-3.6/firefox
18741 lennart  name=systemd:/user/lennart/1        \_ /usr/lib64/firefox-3.6/firefox
28900 lennart  name=systemd:/user/lennart/1           \_ /usr/lib64/nspluginwrapper/npviewer.bin --plugin
/usr/lib64/mozilla/plugins/libflashplayer.so --connection /org/wrapper/NSPlugins/libflashplayer.so/18741-6
 4016 root     name=systemd:/systemd-1/sysinit.service /usr/sbin/bluetoothd --udev
 4094 smmsp    name=systemd:/systemd-1/sendmail.service sendmail: Queue runner@01:00:00 for
/var/spool/clientmqueue
 4096 root     name=systemd:/systemd-1/sendmail.service sendmail: accepting connections
 4112 ntp      name=systemd:/systemd-1/ntpd.service /usr/sbin/ntpd -n -u ntp:ntp -g
27262 lennart  name=systemd:/user/lennart/1       /usr/libexec/mission-control-5
27265 lennart  name=systemd:/user/lennart/1       /usr/libexec/telepathy-haze
27268 lennart  name=systemd:/user/lennart/1       /usr/libexec/telepathy-logger
27270 lennart  name=systemd:/user/lennart/1       /usr/libexec/dconf-service
27280 lennart  name=systemd:/user/lennart/1       /usr/libexec/notification-daemon
27284 lennart  name=systemd:/user/lennart/1       /usr/libexec/telepathy-gabble
27285 lennart  name=systemd:/user/lennart/1       /usr/libexec/telepathy-salut
27297 lennart  name=systemd:/user/lennart/1       /usr/libexec/geoclue-yahoo
```

(Note that this output is shortened, I have removed most of the kernel threads here, since they are not relevant in the context of this blog story)

In the third column you see the cgroup systemd assigned to each process. You'll find that the udev processes are in the name=systemd:/systemd-1/sysinit.service cgroup, which is where systemd places all processes started by the sysinit.service service, which covers early boot.

My personal recommendation is to set the shell alias psc to the ps command line shown above:

```
alias psc='ps xawf -eo pid,user,cgroup,args'
```

With this service information of processes is just four keypresses away!

A different way to present the same information is the systemd-cgls tool we ship with systemd. It shows the cgroup hierarchy in a pretty tree. Its output looks like this:

```
$ systemd-cgls
+    2 [kthreadd]
[...]
+ 4281 [flush-8:0]
+ user
| \ lennart
|   \ 1
|     + 1495 pam: gdm-password
|     + 1521 gnome-session
|     + 1534 dbus-launch --sh-syntax --exit-with-session
|     + 1535 /bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
|     + 1603 /usr/libexec/gconfd-2
|     + 1612 /usr/libexec/gnome-settings-daemon
|     + 1615 /ushr/libexec/gvfsd
|     + 1621 metacity
|     + 1626 /usr/libexec//gvfs-fuse-daemon /home/lennart/.gvfs
|     + 1634 /usr/bin/pulseaudio --start --log-target=syslog
|     + 1635 gnome-panel
|     + 1638 nautilus
|     + 1640 /usr/libexec/polkit-gnome-authentication-agent-1
|     + 1641 /usr/bin/seapplet
|     + 1644 gnome-volume-control-applet
|     + 1645 /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=24
|     + 1646 /usr/sbin/restorecond -u
|     + 1649 /usr/libexec/pulse/gconf-helper
|     + 1652 /usr/bin/devilspie
|     + 1662 nm-applet --sm-disable
|     + 1664 gnome-power-manager
|     + 1665 /usr/libexec/gdu-notification-daemon
|     + 1668 /usr/libexec/im-settings-daemon
|     + 1670 /usr/libexec/evolution/2.32/evolution-alarm-notify
|     + 1672 /usr/bin/python /usr/share/system-config-printer/applet.py
|     + 1674 /usr/lib64/deja-dup/deja-dup-monitor
|     + 1675 abrt-applet
|     + 1677 bluetooth-applet
|     + 1678 gpk-update-icon
|     + 1701 /usr/libexec/gvfs-gdu-volume-monitor
|     + 1707 /usr/bin/gnote --panel-applet --oaf-activate-iid=OAFIID:GnoteApplet_Factory --oaf-ior-fd=22
|     + 1725 /usr/libexec/clock-applet
|     + 1727 /usr/libexec/wnck-applet
|     + 1729 /usr/libexec/notification-area-applet
|     + 1759 gnome-screensaver
|     + 1780 /usr/libexec/gvfsd-trash --spawner :1.9 /org/gtk/gvfs/exec_spaw/0
|     + 1864 /usr/libexec/gvfs-afc-volume-monitor
|     + 1874 /usr/libexec/gconf-im-settings-daemon
|     + 1882 /usr/libexec/gvfs-gphoto2-volume-monitor
|     + 1903 /usr/libexec/gvfsd-burn --spawner :1.9 /org/gtk/gvfs/exec_spaw/1
|     + 1909 gnome-terminal
|     + 1913 gnome-pty-helper
|     + 1914 bash
```

```
|    +  1968 ssh-agent
|    +  1994 gpg-agent --daemon --write-env-file
|    +  2221 bash
|    +  2461 bash
|    +  4193 ssh tango
|    + 15113 bash
|    + 18679 /bin/sh /usr/lib64/firefox-3.6/run-mozilla.sh /usr/lib64/firefox-3.6/firefox
|    + 18741 /usr/lib64/firefox-3.6/firefox
|    + 27251 empathy
|    + 27262 /usr/libexec/mission-control-5
|    + 27265 /usr/libexec/telepathy-haze
|    + 27268 /usr/libexec/telepathy-logger
|    + 27270 /usr/libexec/dconf-service
|    + 27280 /usr/libexec/notification-daemon
|    + 27284 /usr/libexec/telepathy-gabble
|    + 27285 /usr/libexec/telepathy-salut
|    + 27297 /usr/libexec/geoclue-yahoo
|    + 28900 /usr/lib64/nspluginwrapper/npviewer.bin --plugin /usr/lib64/mozilla/plugins/libflashplayer.so --connection
/org/wrapper/NSPlugins/libflashplayer.so/18741-6
|    + 29219 emacs systemd-for-admins-1.txt
|    + 29231 ssh tango
|    \ 29519 systemd-cgls
\ systemd-1
   + 1 /sbin/init
   + ntpd.service
   | \ 4112 /usr/sbin/ntpd -n -u ntp:ntp -g
   + systemd-logger.service
   | \ 1499 /lib/systemd/systemd-logger
   + accounts-daemon.service
   | \ 1496 /usr/libexec/accounts-daemon
   + rtkit-daemon.service
   | \ 1473 /usr/libexec/rtkit-daemon
   + console-kit-daemon.service
   | \ 1408 /usr/sbin/console-kit-daemon --no-daemon
   + prefdm.service
   | + 1376 /usr/sbin/gdm-binary -nodaemon
   | + 1391 /usr/libexec/gdm-simple-slave --display-id /org/gnome/DisplayManager/Display1 --force-active-vt
   | + 1394 /usr/bin/Xorg :0 -nr -verbose -auth /var/run/gdm/auth-for-gdm-f2KUOh/database -nolisten tcp vt1
   | + 1419 /usr/bin/dbus-launch --exit-with-session
   | \ 1511 /usr/bin/gnome-keyring-daemon --daemonize --login
   + getty@.service
   | + tty6
   | | \ 1346 /sbin/mingetty tty6
   | + tty4
   | | \ 1343 /sbin/mingetty tty4
   | + tty5
   | | \ 1342 /sbin/mingetty tty5
   | + tty3
   | | \ 1339 /sbin/mingetty tty3
   | \ tty2
   |   \ 1332 /sbin/mingetty tty2
   + abrtd.service
   | \ 1317 /usr/sbin/abrtd -d -s
   + crond.service
   | \ 1344 crond
   + sshd.service
   | \ 1362 /usr/sbin/sshd
   + sendmail.service
```

```
    | + 4094 sendmail: Queue runner@01:00:00 for /var/spool/clientmqueue
    | \ 4096 sendmail: accepting connections
    + haldaemon.service
    | + 1249 hald
    | + 1250 hald-runner
    | + 1273 hald-addon-input: Listening on /dev/input/event3 /dev/input/event9 /dev/input/event1 /dev/input/event7
/dev/input/event2 /dev/input/event0 /dev/input/event8
    | + 1275 /usr/libexec/hald-addon-rfkill-killswitch
    | + 1284 /usr/libexec/hald-addon-leds
    | + 1285 /usr/libexec/hald-addon-generic-backlight
    | \ 1287 /usr/libexec/hald-addon-acpi
    + irqbalance.service
    | \ 1210 irqbalance
    + avahi-daemon.service
    | + 1175 avahi-daemon: running [epsilon.local]
    + NetworkManager.service
    | + 1171 /usr/sbin/NetworkManager --no-daemon
    | \ 4028 /sbin/dhclient -d -4 -sf /usr/libexec/nm-dhcp-client.action -pf /var/run/dhclient-wlan0.pid -lf
/var/lib/dhclient/dhclient-7d32a784-ede9-4cf6-9ee3-60edc0bce5ff-wlan0.lease -cf /var/run/nm-dhclient-wlan0.conf wlan0
    + rsyslog.service
    | \ 1193 /sbin/rsyslogd -c 4
    + mdmonitor.service
    | \ 1207 mdadm --monitor --scan -f --pid-file=/var/run/mdadm/mdadm.pid
    + cups.service
    | \ 1195 cupsd -C /etc/cups/cupsd.conf
    + auditd.service
    | + 1131 auditd
    | + 1133 /sbin/audispd
    | \ 1135 /usr/sbin/sedispatch
    + dbus.service
    | +  1096 /bin/dbus-daemon --system --address=systemd: --nofork --systemd-activation
    | +  1216 /usr/sbin/modem-manager
    | +  1219 /usr/libexec/polkit-1/polkitd
    | +  1242 /usr/sbin/wpa_supplicant -c /etc/wpa_supplicant/wpa_supplicant.conf -B -u -f /var/log/wpa_supplicant.log -P
/var/run/wpa_supplicant.pid
    | +  1453 /usr/libexec/upowerd
    | +  1733 /usr/libexec/udisks-daemon
    | +  1747 udisks-daemon: polling /dev/sr0
    | \ 29509 /usr/libexec/packagekitd
    + dev-mqueue.mount
    + dev-hugepages.mount
    \ sysinit.service
        +   455 /sbin/udevd -d
        +  4016 /usr/sbin/bluetoothd --udev
        + 28188 /sbin/udevd -d
        \ 28191 /sbin/udevd -d
```

(This too is shortened, the same way)

As you can see, this command shows the processes by their cgroup and hence service, as systemd labels the cgroups after the services. For example, you can easily see that the auditing service auditd.service spawns three individual processes, auditd, audisp and sedispatch.

If you look closely you will notice that a number of processes have been assigned to the cgroup /user/1. At this point let's simply leave it at that systemd not only maintains services in cgroups, but user session processes as well. In a later installment we'll discuss in more detail what this about.

systemd for Administrators, Part III

How Do I Convert A SysV Init Script Into A systemd Service File?

Traditionally, Unix and Linux services (*daemons*) are started via SysV init scripts. These are Bourne Shell scripts, usually residing in a directory such as /etc/rc.d/init.d/which when called with one of a few standardized arguments (verbs) such as start, stop or restart controls, i.e. starts, stops or restarts the service in question. For starts this usually involves invoking the daemon binary, which then forks a background process (more precisely *daemonizes*). Shell scripts tend to be slow, needlessly hard to read, very verbose and fragile. Although they are immensly flexible (after all, they are just code) some things are very hard to do properly with shell scripts, such as ordering parallized execution, correctly supervising processes or just configuring execution contexts in all detail. systemd provides compatibility with these shell scripts, but due to the shortcomings pointed out it is recommended to install native systemd service files for all daemons installed. Also, in contrast to SysV init scripts which have to be adjusted to the distribution systemd service files are compatible with any kind of distribution running systemd (which become more and more these days...). What follows is a terse guide how to take a SysV init script and translate it into a native systemd service file. Ideally, upstream projects should ship and install systemd service files in their tarballs. If you have successfully converted a SysV script according to the guidelines it might hence be a good idea to submit the file as patch to upstream. How to prepare a patch like that will be discussed in a later installment, suffice to say at this point that the daemon(7) manual page shipping with systemd contains a lot of useful information regarding this.

So, let's jump right in. As an example we'll convert the init script of the ABRT daemon into a systemd service file. ABRT is a standard component of every Fedora install, and is an acronym for Automatic Bug Reporting Tool, which pretty much describes what it does, i.e. it is a service for collecting crash dumps. Its SysV script I have uploaded here.

The first step when converting such a script is to read it (surprise surprise!) and distill the useful information from the usually pretty long script. In almost all cases the script consists of mostly boilerplate code that is identical or at least very similar in all init scripts, and usually copied and pasted from one to the other. So, let's extract the interesting information from the script linked above:

- A description string for the service is "*Daemon to detect crashing apps*". As it turns out, the header comments include a redundant number of description strings, some of them describing less the actual service but the init script to start it. systemd services include a description too, and it should describe the service and not the service file.
- The LSB header[1] contains dependency information. systemd due to its design around socket-based activation usually needs no (or very little) manually configured dependencies. (For details regarding socket activation see the original announcement blog post.) In this case the dependency on $syslog (which encodes that abrtd requires a syslog daemon), is the only valuable information. While the header lists another dependency ($local_fs) this one is redundant with systemd as normal system services are always started with all local file systems available.
- The LSB header suggests that this service should be started in runlevels 3 (multi-user) and 5 (graphical).
- The daemon binary is /usr/sbin/abrtd

And that's already it. The entire remaining content of this 115-line shell script is simply boilerplate or otherwise redundant code: code that deals with synchronizing and serializing startup (i.e. the code

regarding lock files) or that outputs status messages (i.e. the code calling echo), or simply parsing of the verbs (i.e. the big case block).

From the information extracted above we can now write our systemd service file:

```
[Unit]
Description=Daemon to detect crashing apps
After=syslog.target

[Service]
ExecStart=/usr/sbin/abrtd
Type=forking

[Install]
WantedBy=multi-user.target
```

A little explanation of the contents of this file: The [Unit] section contains generic information about the service. systemd not only manages system services, but also devices, mount points, timer, and other components of the system. The generic term for all these objects in systemd is a *unit*, and the [Unit] section encodes information about it that might be applicable not only to services but also in to the other unit types systemd maintains. In this case we set the following unit settings: we set the description string and configure that the daemon shall be started after Syslog[2], similar to what is encoded in the LSB header of the original init script. For this Syslog dependency we create a dependency of type After=on a systemd unit syslog.target. The latter is a special target unit in systemd and is the standardized name to pull in a syslog implementation. For more information about these standardized names see the systemd.special(7). Note that a dependency of type After= only encodes the suggested ordering, but does not actually cause syslog to be started when abrtd is -- and this is exactly what we want, since abrtd actually works fine even without syslog being around. However, if both are started (and usually they are) then the order in which they are is controlled with this dependency.

The next section is [Service] which encodes information about the service itself. It contains all those settings that apply only to services, and not the other kinds of units systemd maintains (mount points, devices, timers, ...). Two settings are used here: ExecStart= takes the path to the binary to execute when the service shall be started up. And with Type=we configure how the service notifies the init system that it finished starting up. Since traditional Unix daemons do this by returning to the parent process after having forked off and initialized the background daemon we set the type to forking here. That tells systemd to wait until the start-up binary returns and then consider the processes still running afterwards the daemon processes.

The final section is [Install]. It encodes information about how the suggested installation should look like, i.e. under which circumstances and by which triggers the service shall be started. In this case we simply say that this service shall be started when the multi-user.target unit is activated. This is a special unit (see above) that basically takes the role of the classic SysV Runlevel 3[3]. The setting WantedBy= has little effect on the daemon during runtime. It is only read by the systemctl enable command, which is the recommended way to enable a service in systemd. This command will simply ensure that our little service gets automatically activated as soon as multi-user.target is requested, which it is on all normal boots[4].

And that's it. Now we already have a minimal working systemd service file. To test it we copy it to /etc/systemd/system/abrtd.service and invoke systemctl daemon-reload. This will make systemd take notice of it, and now we can start the service with it: systemctl start abrtd.service. We can verify the status via systemctl status abrtd.service. And we can stop it again via systemctl stop abrtd.service.

Finally, we can enable it, so that it is activated by default on future boots withsystemctl enable abrtd.service.

The service file above, while sufficient and basically a 1:1 translation (feature- and otherwise) of the SysV init script still has room for improvement. Here it is a little bit updated:

```
[Unit]
Description=ABRT Automated Bug Reporting Tool
After=syslog.target

[Service]
Type=dbus
BusName=com.redhat.abrt
ExecStart=/usr/sbin/abrtd -d -s

[Install]
WantedBy=multi-user.target
```

So, what did we change? Two things: we improved the description string a bit. More importantly however, we changed the type of the service to dbus and configured the D-Bus bus name of the service. Why did we do this? As mentioned classic SysV services *daemonize* after startup, which usually involves double forking and detaching from any terminal. While this is useful and necessary when daemons are invoked via a script, this is unnecessary (and slow) as well as counterproductive when a proper process babysitter such as systemd is used. The reason for that is that the forked off daemon process usually has little relation to the original process started by systemd (after all the daemonizing scheme's whole idea is to remove this relation), and hence it is difficult for systemd to figure out after the fork is finished which process belonging to the service is actually the main process and which processes might just be auxiliary. But that information is crucial to implement advanced babysitting, i.e. supervising the process, automatic respawning on abnormal termination, collectig crash and exit code information and suchlike. In order to make it easier for systemd to figure out the main process of the daemon we changed the service type to dbus. The semantics of this service type are appropriate for all services that take a name on the D-Bus system bus as last step of their initialization[5]. ABRT is one of those. With this setting systemd will spawn the ABRT process, which will no longer fork (this is configured via the -d -s switches to the daemon), and systemd will consider the service fully started up as soon as com.redhat.abrt appears on the bus. This way the process spawned by systemd is the main process of the daemon, systemd has a reliable way to figure out when the daemon is fully started up and systemd can easily supervise it.

And that's all there is to it. We have a simple systemd service file now that encodes in 10 lines more information than the original SysV init script encoded in 115. And even now there's a lot of room left for further improvement utilizing more features systemd offers. For example, we could set Restart=restart-always to tell systemd to automatically restart this service when it dies. Or, we could use OOMScoreAdjust=-500 to ask the kernel to please leave this process around when the OOM killer wreaks havoc. Or, we could use CPUSchedulingPolicy=idle to ensure that abrtd processes crash dumps in background only, always allowing the kernel to give preference to whatever else might be running and needing CPU time.

For more information about the configuration options mentioned here, see the respective man pages systemd.unit(5), systemd.service(5), systemd.exec(5). Or, browse all of systemd's man pages.

Of course, not all SysV scripts are as easy to convert as this one. But gladly, as it turns out the vast majority actually are.

That's it for today, come back soon for the next installment in our series.

Footnotes

[1] The LSB header of init scripts is a convention of including meta data about the service in comment blocks at the top of SysV init scripts and is defined by the Linux Standard Base. This was intended to standardize init scripts between distributions. While most distributions have adopted this scheme, the handling of the headers varies greatly between the distributions, and in fact still makes it necessary to adjust init scripts for every distribution. As such the LSB spec never kept the promise it made.

[2] Strictly speaking, this dependency does not even have to be encoded here, as it is redundant in a system where the Syslog daemon is socket activatable. Modern syslog systems (for example rsyslog v5) have been patched upstream to be socket-activatable. If such a init system is used configuration of the After=syslog.target dependency is redundant and implicit. However, to maintain compatibility with syslog services that have not been updated we include this dependency here.

[3] At least how it used to be defined on Fedora.

[4] Note that in systemd the graphical bootup (graphical.target, taking the role of SysV runlevel 5) is an implicit superset of the console-only bootup (multi-user.target, i.e. like runlevel 3). That means hooking a service into the latter will also hook it into the former.

[5] Actually the majority of services of the default Fedora install now take a name on the bus after startup.

systemd for Administrators, Part IV

Killing Services

Killing a system daemon is easy, right? Or is it?

Sure, as long as your daemon persists only of a single process this might actually be somewhat true. You type killall rsyslogd and the syslog daemon is gone. However it is a bit dirty to do it like that given that this will kill all processes which happen to be called like this, including those an unlucky user might have named that way by accident. A slightly more correct version would be to read the .pid file, i.e. kill `cat /var/run/syslogd.pid`. That already gets us much further, but still, is this really what we want?

More often than not it actually isn't. Consider a service like Apache, or crond, or atd, which as part of their usual operation spawn child processes. Arbitrary, user configurable child processes, such as cron or at jobs, or CGI scripts, even full application servers. If you kill the main apache/crond/atd process this might or might not pull down the child processes too, and it's up to those processes whether they want to stay around or go down as well. Basically that means that terminating Apache might very well cause its CGI scripts to stay around, reassigned to be children of init, and difficult to track down.

systemd to the rescue: With systemctl kill you can easily send a signal to all processes of a service. Example:

```
# systemctl kill crond.service
```

This will ensure that SIGTERM is delivered to all processes of the crond service, not just the main process. Of course, you can also send a different signal if you wish. For example, if you are bad-ass you might want to go for SIGKILL right-away:

```
# systemctl kill -s SIGKILL crond.service
```

And there you go, the service will be brutally slaughtered in its entirety, regardless how many times it forked, whether it tried to escape supervision by double forking or fork bombing.

Sometimes all you need is to send a specific signal to the main process of a service, maybe because you want to trigger a reload via SIGHUP. Instead of going via the PID file, here's an easier way to do this:

```
# systemctl kill -s HUP --kill-who=main crond.service
```

So again, what is so new and fancy about killing services in systemd? Well, for the first time on Linux we can actually properly do that. Previous solutions were always depending on the daemons to actually cooperate to bring down everything they spawned if they themselves terminate. However, usually if you want to use SIGTERM or SIGKILL you are doing that because they actually do not cooperate properly with you.

How does this relate to systemctl stop? kill goes directly and sends a signal to every process in the group, however stop goes through the official configured way to shut down a service, i.e. invokes the stop command configured with ExecStop= in the service file. Usually stop should be sufficient. kill is the tougher version, for cases where you either don't want the official shutdown command of a service to run, or when the service is hosed and hung in other ways.

(It's up to you BTW to specify signal names with or without the SIG prefix on the -s switch. Both works.)

It's a bit surprising that we have come so far on Linux without even being able to properly kill services. systemd for the first time enables you to do this properly.

systemd for Administrators, Part V

The Three Levels of "Off"

In systemd, there are three levels of turning off a service (or other unit). Let's have a look which those are:

1. You can **stop** a service. That simply terminates the running instance of the service and does little else. If due to some form of activation (such as manual activation, socket activation, bus activation, activation by system boot or activation by hardware plug) the service is requested again afterwards it will be started. Stopping a service is hence a very simple, temporary and superficial operation. Here's an example how to do this for the NTP service:

   ```
   $ systemctl stop ntpd.service
   ```

   This is roughly equivalent to the following traditional command which is available on most SysV inspired systems:

   ```
   $ service ntpd stop
   ```

   In fact, on Fedora 15, if you execute the latter command it will be transparently converted to the former.

2. You can **disable** a service. This unhooks a service from its activation triggers. That means, that depending on your service it will no longer be activated on boot, by socket or bus activation or by hardware plug (or any other trigger that applies to it). However, you can still start it manually if you wish. If there is already a started instance disabling a service will *not* have the effect of stopping it. Here's an example how to disable a service:

   ```
   $ systemctl disable ntpd.service
   ```

   On traditional Fedora systems, this is roughly equivalent to the following command:

   ```
   $ chkconfig ntpd off
   ```

   And here too, on Fedora 15, the latter command will be transparently converted to the former, if necessary.

   Often you want to combine stopping and disabling a service, to get rid of the current instance and make sure it is not started again (except when manually triggered):

   ```
   $ systemctl disable ntpd.service
   $ systemctl stop ntpd.service
   ```

   Commands like this are for example used during package deinstallation of systemd services on Fedora.

   Disabling a service is a permanent change; until you undo it it will be kept, even across reboots.

3. You can **mask** a service. This is like disabling a service, but on steroids. It not only makes sure that service is not started automatically anymore, but even ensures that a service cannot even be started manually anymore. This is a bit of a hidden feature in systemd, since it is not commonly useful and might be confusing the user. But here's how you do it:

   ```
   $ ln -s /dev/null /etc/systemd/system/ntpd.service
   ```

```
$ systemctl daemon-reload
```

> By symlinking a service file to /dev/null you tell systemd to never start the service in question and completely block its execution. Unit files stored in/etc/systemd/system override those from /lib/systemd/system that carry the same name. The former directory is administrator territory, the latter terroritory of your package manager. By installing your symlink in /etc/systemd/system/ntpd.service you hence make sure that systemd will never read the upstream shipped service file/lib/systemd/system/ntpd.service.
>
> systemd will recognize units symlinked to /dev/null and show them as *masked*. If you try to start such a service manually (via systemctl start for example) this will fail with an error.
>
> A similar trick on SysV systems does not (officially) exist. However, there are a few unofficial hacks, such as editing the init script and placing an exit 0 at the top, or removing its execution bit. However, these solutions have various drawbacks, for example they interfere with the package manager.
>
> Masking a service is a permanent change, much like disabling a service.

Now that we learned how to turn off services on three levels, there's only one question left: how do we turn them on again? Well, it's quite symmetric. use systemctl start to undo systemctl stop. Use systemctl enable to undo systemctl disable and use rm to undo ln.

systemd for Administrators, Part VI

Changing Roots

As administrator or developer sooner or later you'll ecounter chroot() environments.
The chroot() system call simply shifts what a process and all its children consider the root directory /, thus limiting what the process can see of the file hierarchy to a subtree of it.
Primarily chroot() environments have two uses:

1. For security purposes: In this use a specific isolated daemon is chroot()ed into a private subdirectory, so that when exploited the attacker can see only the subdirectory instead of the full OS hierarchy: he is trapped inside the chroot() jail.
2. To set up and control a debugging, testing, building, installation or recovery image of an OS: For this a whole guest operating system hierarchy is mounted or bootstraped into a subdirectory of the host OS, and then a shell (or some other application) is started inside it, with this subdirectory turned into its /. To the shell it appears as if it was running inside a system that can differ greatly from the host OS. For example, it might run a different distribution or even a different architecture (Example: host x86_64, guest i386). The full hierarchy of the host OS it cannot see.

On a classic System-V-based operating system it is relatively easy to use chroot() environments. For example, to start a specific daemon for test or other reasons inside a chroot()-based guest OS tree, mount /proc, /sys and a few other API file systems into the tree, and then use chroot(1) to enter the chroot, and finally run the SysV init script via/sbin/service from inside the chroot.

On a systemd-based OS things are not that easy anymore. One of the big advantages of systemd is that all daemons are guaranteed to be invoked in a completely clean and independent context which is in no way related to the context of the user asking for the service to be started. While in sysvinit-based systems a large part of the execution context (like resource limits, environment variables and suchlike) is inherited from the user shell invoking the init skript, in systemd the user just notifies the init daemon, and the init daemon will then fork off the daemon in a sane, well-defined and pristine execution context and no inheritance of the user context parameters takes place. While this is a formidable feature it actually breaks traditional approaches to invoke a service inside a chroot() environment: since the actual daemon is always spawned off PID 1 and thus inherits the chroot() settings from it, it is irrelevant whether the client which asked for the daemon to start is chroot()ed or not. On top of that, since systemd actually places its local communications sockets in/run/systemd a process in a chroot() environment will not even be able to talk to the init system (which however is probably a good thing, and the daring can work around this of course by making use of bind mounts.)

This of course opens the question how to use chroot()s properly in a systemd environment. And here's what we came up with for you, which hopefully answers this question thoroughly and comprehensively:

Let's cover the first usecase first: locking a daemon into a chroot() jail for security purposes. To begin with, chroot() as a security tool is actually quite dubious, since chroot() is not a one-way street. It is relatively easy to escape a chroot() environment, as even the man page points out. Only in combination with a few other techniques it can be made somewhat secure. Due to that it usually requires specific support in the applications to chroot() themselves in a tamper-proof way. On top of that it usually requires a deep understanding of the chroot()ed service to set up the chroot() environment properly, for example to know which directories to bind mount from the host tree, in order to make available all communication channels in the chroot() the service actually needs. Putting this together, chroot()ing

software for security purposes is almost always done best in the C code of the daemon itself. The developer knows best (or at least *should* know best) how to properly secure down the chroot(), and what the minimal set of files, file systems and directories is the daemon will need inside the chroot(). These days a number of daemons are capable of doing this, unfortunately however of those running by default on a normal Fedora installation only two are doing this: Avahi and RealtimeKit. Both apparently written by the same really smart dude. Chapeau! ;-) (Verify this easily by running ls -l /proc/*/root on your system.)

That all said, systemd of course does offer you a way to chroot() specific daemons and manage them like any other with the usual tools. This is supported via the RootDirectory=option in systemd service files. Here's an example:

```
[Unit]
Description=A chroot()ed Service

[Service]
RootDirectory=/srv/chroot/foobar
ExecStartPre=/usr/local/bin/setup-foobar-chroot.sh
ExecStart=/usr/bin/foobard
RootDirectoryStartOnly=yes
```

In this example, RootDirectory= configures where to chroot() to before invoking the daemon binary specified with ExecStart=. Note that the path specified in ExecStart=needs to refer to the binary inside the chroot(), it is not a path to the binary in the host tree (i.e. in this example the binary executed is seen as/srv/chroot/foobar/usr/bin/foobard from the host OS). Before the daemon is started a shell script setup-foobar-chroot.sh is invoked, whose purpose it is to set up the chroot environment as necessary, i.e. mount /proc and similar file systems into it, depending on what the service might need. With the RootDirectoryStartOnly= switch we ensure that only the daemon as specified in ExecStart= is chrooted, but not the ExecStartPre= script which needs to have access to the full OS hierarchy so that it can bind mount directories from there. (For more information on these switches see the respective man pages.) If you place a unit file like this in /etc/systemd/system/foobar.serviceyou can start your chroot()ed service by typing systemctl start foobar.service. You may then introspect it with systemctl status foobar.service. It is accessible to the administrator like any other service, the fact that it is chroot()ed does -- unlike on SysV -- not alter how your monitoring and control tools interact with it.

Newer Linux kernels support file system namespaces. These are similar to chroot() but a lot more powerful, and they do not suffer by the same security problems as chroot(). systemd exposes a subset of what you can do with file system namespaces right in the unit files themselves. Often these are a useful and simpler alternative to setting up full chroot() environment in a subdirectory. With the switches ReadOnlyDirectories= and InaccessibleDirectories= you may setup a file system namespace jail for your service. Initially, it will be identical to your host OS' file system namespace. By listing directories in these directives you may then mark certain directories or mount points of the host OS as read-only or even completely inaccessible to the daemon. Example:

```
[Unit]
Description=A Service With No Access to /home

[Service]
ExecStart=/usr/bin/foobard
InaccessibleDirectories=/home
```

This service will have access to the entire file system tree of the host OS with one exception: /home will not be visible to it, thus protecting the user's data from potential exploiters. (See the man page for details on these options.)

File system namespaces are in fact a better replacement for chroot()s in many many ways. Eventually Avahi and RealtimeKit should probably be updated to make use of namespaces replacing chroot()s.

So much about the security usecase. Now, let's look at the other use case: setting up and controlling OS images for debugging, testing, building, installing or recovering.

chroot() environments are relatively simple things: they only virtualize the file system hierarchy. By chroot()ing into a subdirectory a process still has complete access to all system calls, can kill all processes and shares about everything else with the host it is running on. To run an OS (or a small part of an OS) inside a chroot() is hence a dangerous affair: the isolation between host and guest is limited to the file system, everything else can be freely accessed from inside the chroot(). For example, if you upgrade a distribution inside a chroot(), and the package scripts send a SIGTERM to PID 1 to trigger a reexecution of the init system, this will actually take place in the host OS! On top of that, SysV shared memory, abstract namespace sockets and other IPC primitives are shared between host and guest. While a completely secure isolation for testing, debugging, building, installing or recovering an OS is probably not necessary, a basic isolation to avoid *accidental* modifications of the host OS from inside the chroot() environment is desirable: you never know what code package scripts execute which might interfere with the host OS.

To deal with chroot() setups for this use systemd offers you a couple of features:

First of all, systemctl detects when it is run in a chroot. If so, most of its operations will become NOPs, with the exception of systemctl enable and systemctl disable. If a package installation script hence calls these two commands, services will be enabled in the guest OS. However, should a package installation script include a command likesystemctl restart as part of the package upgrade process this will have no effect at all when run in a chroot() environment.

More importantly however systemd comes out-of-the-box with the systemd-nspawn tool which acts as chroot(1) on steroids: it makes use of file system and PID namespaces to boot a simple lightweight container on a file system tree. It can be used almost like chroot(1), except that the isolation from the host OS is much more complete, a lot more secure and even easier to use. In fact, systemd-nspawn is capable of booting a *complete* systemd or sysvinit OS in container with a single command. Since it virtualizes PIDs, the init system in the container can act as PID 1 and thus do its job as normal. In contrast to chroot(1) this tool will implicitly mount /proc, /sys for you.

Here's an example how in three commands you can boot a Debian OS on your Fedora machine inside an nspawn container:

```
# yum install debootstrap
# debootstrap --arch=amd64 unstable debian-tree/
# systemd-nspawn -D debian-tree/
```

This will bootstrap the OS directory tree and then simply invoke a shell in it. If you want to boot a full system in the container, use a command like this:

```
# systemd-nspawn -D debian-tree/ /sbin/init
```

And after a quick bootup you should have a shell prompt, inside a complete OS, booted in your container. The container will not be able to see any of the processes outside of it. It will share the network configuration, but not be able to modify it. (Expect a couple of EPERMs during boot for that,

which however should not be fatal). Directories like /sys and/proc/sys are available in the container, but mounted read-only in order to avoid that the container can modify kernel or hardware configuration. Note however that this protects the host OS only from *accidental* changes of its parameters. A process in the container can manually remount the file systems read-writeable and then change whatever it wants to change.

So, what's so great about systemd-nspawn again?

1. It's really easy to use. No need to manually mount /proc and /sys into your chroot() environment. The tool will do it for you and the kernel automatically cleans it up when the container terminates.
2. The isolation is much more complete, protecting the host OS from accidental changes from inside the container.
3. It's so good that you can actually boot a full OS in the container, not just a single lonesome shell.
4. It's actually tiny and installed everywhere where systemd is installed. No complicated installation or setup.

systemd itself has been modified to work very well in such a container. For example, when shutting down and detecting that it is run in a container, it just calls exit(), instead of reboot() as last step.

Note that systemd-nspawn is not a full container solution. If you need that LXC is the better choice for you. It uses the same underlying kernel technology but offers a lot more, including network virtualization. If you so will, systemd-nspawn is the GNOME 3 of container solutions: slick and trivially easy to use -- but with few configuration options. LXC OTOH is more like KDE: more configuration options than lines of code. I wrote systemd-nspawn specifically to cover testing, debugging, building, installing, recovering. That's what you should use it for and what it is really good at, and where it is a much much nicer alternative to chroot(1).

So, let's get this finished, this was already long enough. Here's what to take home from this little blog story:

1. Secure chroot()s are best done natively in the C sources of your program.
2. ReadOnlyDirectories=, InaccessibleDirectories= might be suitable alternatives to a full chroot() environment.
3. RootDirectory= is your friend if you want to chroot() a specific service.
4. systemd-nspawn is made of awesome.
5. chroot()s are lame, file system namespaces are totally l33t.

systemd for Administrators, Part VII

The Blame Game

Fedora 15[1] is the first Fedora release to sport systemd. Our primary goal for F15 was to get everything integrated and working well. One focus for Fedora 16 will be to further polish and speed up what we have in the distribution now. To prepare for this cycle we have implemented a few tools (which are already available in F15), which can help us pinpoint where exactly the biggest problems in our boot-up remain. With this blog story I hope to shed some light on how to figure out what to blame for your slow boot-up, and what to do about it. We want to allow you to put the blame where the blame belongs: on the system component responsible.

The first utility is a very simple one: systemd will automatically write a log message with the time it needed to syslog/kmsg when it finished booting up.

```
systemd[1]: Startup finished in 2s 65ms 924us (kernel) + 2s 828ms 195us (initrd) + 11s 900ms 471us (userspace) = 16s 794ms 590us.
```

And here's how you read this: 2s have been spent for kernel initialization, until the time where the initial RAM disk (initrd, i.e. dracut) was started. A bit less than 3s have then been spent in the initrd. Finally, a bit less than 12s have been spent after the actual system init daemon (systemd) has been invoked by the initrd to bring up userspace. Summing this up the time that passed since the boot loader jumped into the kernel code until systemd was finished doing everything it needed to do at boot was a bit less than 17s. This number is nice and simple to understand -- and also easy to misunderstand: it does not include the time that is spent initializing your GNOME session, as that is outside of the scope of the init system. Also, in many cases this is just where systemd finished doing everything it needed to do. Very likely some daemons are still busy doing whatever *they* need to do to finish startup when this time is elapsed. Hence: while the time logged here is a good indication on the general boot speed, it is not the time the user might *feel* the boot actually takes.

Also, it is a pretty superficial value: it gives no insight which system component systemd was waiting for all the time. To break this up, we introduced the tool systemd-analyze blame:

```
$ systemd-analyze blame
     6207ms udev-settle.service
     5228ms cryptsetup@luks\x2d9899b85d\x2df790\x2d4d2a\x2da650\x2d8b7d2fb92cc3.service
      735ms NetworkManager.service
      642ms avahi-daemon.service
      600ms abrtd.service
      517ms rtkit-daemon.service
      478ms fedora-storage-init.service
      396ms dbus.service
      390ms rpcidmapd.service
      346ms systemd-tmpfiles-setup.service
      322ms fedora-sysinit-unhack.service
      316ms cups.service
      310ms console-kit-log-system-start.service
      309ms libvirtd.service
      303ms rpcbind.service
      298ms ksmtuned.service
      288ms lvm2-monitor.service
      281ms rpcgssd.service
      277ms sshd.service
      276ms livesys.service
```

```
     267ms iscsid.service
     236ms mdmonitor.service
     234ms nfslock.service
     223ms ksm.service
     218ms mcelog.service
...
```

This tool lists which systemd unit needed how much time to finish initialization at boot, the worst offenders listed first. What we can see here is that on this boot two services required more than 1s of boot time: udev-settle.service and cryptsetup@luks\x2d9899b85d\x2df790\x2d4d2a\x2da650\x2d8b7d2fb92cc3.service. This tool's output is easily misunderstood as well, it does not shed any light on why the services in question actually need this much time, it just determines that they did. Also note that the times listed here might be spent "in parallel", i.e. two services might be initializing at the same time and thus the time spent to initialize them both is much less than the sum of both individual times combined.

Let's have a closer look at the worst offender on this boot: a service by the name of udev-settle.service. So why does it take that much time to initialize, and what can we do about it? This service actually does very little: it just waits for the device probing being done by udev to finish and then exits. Device probing can be slow. In this instance for example, the reason for the device probing to take more than 6s is the 3G modem built into the machine, which when not having an inserted SIM card takes this long to respond to software probe requests. The software probing is part of the logic that makes ModemManager work and enables NetworkManager to offer easy 3G setup. An obvious reflex might now be to blame ModemManager for having such a slow prober. But that's actually ill-directed: hardware probing quite frequently is this slow, and in the case of ModemManager it's a simple fact that the 3G hardware takes this long. It is an essential requirement for a proper hardware probing solution that individual probers can take this much time to finish probing. The actual culprit is something else: the fact that we actually wait for the probing, in other words: that udev-settle.service is part of our boot process.

So, why is udev-settle.service part of our boot process? Well, it actually doesn't need to be. It is pulled in by the storage setup logic of Fedora: to be precise, by the LVM, RAID and Multipath setup script. These storage services have not been implemented in the way hardware detection and probing work today: they expect to be initialized at a point in time where "all devices have been probed", so that they can simply iterate through the list of available disks and do their work on it. However, on modern machinery this is not how things actually work: hardware can come and hardware can go all the time, during boot and during runtime. For some technologies it is not even possible to know when the device enumeration is complete (example: USB, or iSCSI), thus waiting for all storage devices to show up and be probed must necessarily include a fixed delay when it is assumed that all devices that can show up have shown up, and got probed. In this case all this shows very negatively in the boot time: the storage scripts force us to delay bootup until all potential devices have shown up and all devices that did got probed -- and all that even though we don't actually need most devices for anything. In particular since this machine actually does not make use of LVM, RAID or Multipath![2]

Knowing what we know now we can go and disable udev-settle.service for the next boots: since neither LVM, RAID nor Multipath is used we can mask the services in question and thus speed up our boot a little:

```
# ln -s /dev/null /etc/systemd/system/udev-settle.service
# ln -s /dev/null /etc/systemd/system/fedora-wait-storage.service
# ln -s /dev/null /etc/systemd/system/fedora-storage-init.service
# systemctl daemon-reload
```

After restarting we can measure that the boot is now about 1s faster. Why just 1s? Well, the second worst offender is cryptsetup here: the machine in question has an encrypted/home directory. For testing purposes I have stored the passphrase in a file on disk, so that the boot-up is not delayed because I as the user am a slow typer. The cryptsetup tool unfortunately still takes more han 5s to set up the encrypted partition. Being lazy instead of trying to fix cryptsetup[3] we'll just tape over it here [4]: systemd will normally wait for all file systems not marked with the noauto option in /etc/fstab to show up, to be fscked and to be mounted before proceeding bootup and starting the usual system services. In the case of /home (unlike for example /var) we know that it is needed only very late (i.e. when the user actually logs in). An easy fix is hence to make the mount point available already during boot, but not actually wait until cryptsetup, fsck and mount finished running for it. You ask how we can make a mount point available before actually mounting the file system behind it? Well, systemd possesses magic powers, in form of the comment=systemd.automount mount option in /etc/fstab. If you specify it, systemd will create an automount point at /home and when at the time of the first access to the file system it still isn't backed by a proper file system systemd will wait for the device, fsck and mount it.

And here's the result with this change to /etc/fstab made:

```
systemd[1]: Startup finished in 2s 47ms 112us (kernel) + 2s 663ms 942us (initrd) + 5s 540ms 522us (userspace) = 10s 251ms 576us.
```

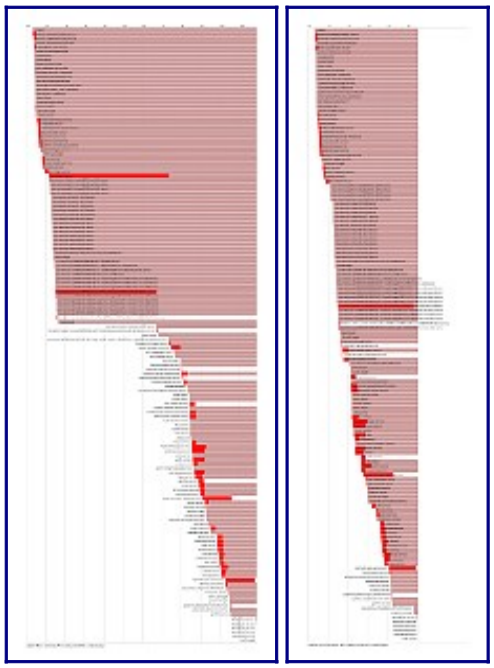
Nice! With a few fixes we took almost 7s off our boot-time. And these two changes are only fixes for the two most superficial problems. With a bit of love and detail work there's a lot of additional room for improvements. In fact, on a different machine, a more than two year old X300 laptop (which even back then wasn't the fastest machine on earth) and a bit of decrufting we have boot times of around 4s (total) now, with a resonably complete GNOME system. And there's still a lot of room in it.

systemd-analyze blame is a nice and simple tool for tracking down slow services. However, it suffers by a big problem: it does not visualize how the parallel execution of the services actually diminishes the price one pays for slow starting services. For that we have prepared systemd-analyize plot for you. Use it like this:

```
$ systemd-analyze plot > plot.svg
$ eog plot.svg
```

It creates pretty graphs, showing the time services spent to start up in relation to the other services. It currently doesn't visualize explicitly which services wait for which ones, but with a bit of guess work this is easily seen nonetheless.

To see the effect of our two little optimizations here are two graphs generated with systemd-analyze plot, the first before and the other after our change:



(For the sake of completeness, here are the two complete outputs of systemd-analyze blame for these two boots: before and after.)

The well-informed reader probably wonders how this relates to Michael Meeks' bootchart. This plot and bootchart do show similar graphs, that is true. Bootchart is by far the more powerful tool. It plots in all detail what is happening during the boot, how much CPU and IO is used. systemd-analyze plot shows more high-level data: which service took how much time to initialize, and what needed to wait for it. If you use them both together you'll have a wonderful toolset to figure out why your boot is not as fast as it could be.

Now, before you now take these tools and start filing bugs against the worst boot-up time offenders on your system: think twice. These tools give you raw data, don't misread it. As my optimization example above hopefully shows, the blame for the slow bootup was not actually with udev-settle.service, and not with the ModemManager prober run by it either. It is with the subsystem that pulled this service in in the first place. And that's where the problem needs to be fixed. So, file the bugs at the right places. Put the blame where the blame belongs.

As mentioned, these three utilities are available on your Fedora 15 system out-of-the-box.

And here's what to take home from this little blog story:

- systemd-analyze is a wonderful tool and systemd comes with profiling built in.
- Don't misread the data these tools generate!
- With two simple changes you might be able to speed up your system by 7s!
- Fix your software if it can't handle dynamic hardware properly!
- The Fedora default of installing the OS on an enterprise-level storage managing system might be something to rethink.

And that's all for now. Thank you for your interest.

Footnotes

[1] Also known as the greatest Free Software OS release ever.

[2] The right fix here is to improve the services in question to actively listen to hotplug events via libudev or similar and act on the devices showing up as they show up, so that we can continue with the bootup the instant everything we really need to

go on has shown up. To get a quick bootup we should wait for what we actually need to proceed, not for everything. Also note that the storage services are not the only services which do not cope well with modern dynamic hardware, and assume that the device list is static and stays unchanged. For example, in this example the reason the initrd is actually as slow as it is is mostly due to the fact that Plymouth expects to be executed when all video devices have shown up and have been probed. For an unknown reason (at least unknown to me) loading the video kernel modules for my Intel graphics cards takes multiple seconds, and hence the entire boot is delayed unnecessarily. (Here too I'd not put the blame on the probing but on the fact that we wait for it to complete before going on.)

[3] Well, to be precise, I actually did try to get this fixed. Most of the delay of crypsetup stems from the -- in my eyes -- unnecessarily high default values for --iter-time in cryptsetup. I tried to convince our cryptsetup maintainers that 100ms as a default here are not really less secure than 1s, but well, I failed.

[4] Of course, it's usually not our style to just tape over problems instead of fixing them, but this is such a nice occasion to show off yet another cool systemd feature...

systemd for Administrators, Part VIII

The New Configuration Files

One of the formidable new features of systemd is that it comes with a complete set of modular early-boot services that are written in simple, fast, parallelizable and robust C, replacing the shell "novels" the various distributions featured before. Our little *Project Zero Shell*[1] has been a full success. We currently cover pretty much everything most desktop and embedded distributions should need, plus a big part of the server needs:

- Checking and mounting of all file systems
- Updating and enabling quota on all file systems
- Setting the host name
- Configuring the loopback network device
- Loading the SELinux policy and relabelling /run and /dev as necessary on boot
- Registering additional binary formats in the kernel, such as Java, Mono and WINE binaries
- Setting the system locale
- Setting up the console font and keyboard map
- Creating, removing and cleaning up of temporary and volatile files and directories
- Applying mount options from /etc/fstab to pre-mounted API VFS
- Applying sysctl kernel settings
- Collecting and replaying readahead information
- Updating utmp boot and shutdown records
- Loading and saving the random seed
- Statically loading specific kernel modules
- Setting up encrypted hard disks and partitions
- Spawning automatic gettys on serial kernel consoles
- Maintenance of Plymouth
- Machine ID maintenance
- Setting of the UTC distance for the system clock

On a standard Fedora 15 install, only a few legacy and storage services still require shell scripts during early boot. If you don't need those, you can easily disable them end enjoy your shell-free boot (like I do every day). The shell-less boot systemd offers you is a unique feature on Linux.

Many of these small components are configured via configuration files in /etc. Some of these are fairly standardized among distributions and hence supporting them in the C implementations was easy and obvious. Examples include: /etc/fstab, /etc/crypttab or /etc/sysctl.conf. However, for others no standardized file or directory existed which forced us to add #ifdef orgies to our sources to deal with the different places the distributions we want to support store these things. All these configuration files have in common that they are dead-simple and there is simply no good reason for distributions to distuingish themselves with them: they all do the very same thing, just a bit differently.

To improve the situation and benefit from the unifying force that systemd is we thus decided to read the per-distribution configuration files only as *fallbacks* -- and to introduce new configuration files as primary source of configuration wherever applicable. Of course, where possible these standardized configuration files should not be new inventions but rather just standardizations of the best distribution-

specific configuration files previously used. Here's a little overview over these new common configuration files systemd supports on all distributions:

- /etc/hostname: the host name for the system. One of the most basic and trivial system settings. Nonetheless previously all distributions used different files for this. Fedora used /etc/sysconfig/network, OpenSUSE /etc/HOSTNAME. We chose to standardize on the Debian configuration file /etc/hostname.
- /etc/vconsole.conf: configuration of the default keyboard mapping and console font.
- /etc/locale.conf: configuration of the system-wide locale.
- /etc/modules-load.d/*.conf: a drop-in directory for kernel modules to statically load at boot (for the very few that still need this).
- /etc/sysctl.d/*.conf: a drop-in directory for kernel sysctl parameters, extending what you can already do with /etc/sysctl.conf.
- /etc/tmpfiles.d/*.conf: a drop-in directory for configuration of runtime files that need to be removed/created/cleaned up at boot and during uptime.
- /etc/binfmt.d/*.conf: a drop-in directory for registration of additional binary formats for systems like Java, Mono and WINE.
- /etc/os-release: a standardization of the various distribution ID files like /etc/fedora-release and similar. Really every distribution introduced their own file here; writing a simple tool that just prints out the name of the local distribution usually means including a database of release files to check. The LSB tried to standardize something like this with the lsb_release tool, but quite frankly the idea of employing a shell script in this is not the best choice the LSB folks ever made. To rectify this we just decided to generalize this, so that everybody can use the same file here.
- /etc/machine-id: a machine ID file, superseding D-Bus' machine ID file. This file is guaranteed to be existing and valid on a systemd system, covering also stateless boots. By moving this out of the D-Bus logic it is hopefully interesting for a lot of additional uses as a unique and stable machine identifier.
- /etc/machine-info: a new information file encoding meta data about a host, like a pretty host name and an icon name, replacing stuff like /etc/favicon.png and suchlike. This is maintained by systemd-hostnamed.

It is our definite intention to convince *you* to use these new configuration files in your configuration tools: if your configuration frontend writes these files instead of the old ones, it automatically becomes more portable between Linux distributions, and you are helping standardizing Linux. This makes things simpler to understand and more obvious for users and administrators. Of course, right now, only systemd-based distributions read these files, but that already covers all important distributions in one way or another, except for one. And it's a bit of a chicken-and-egg problem: a standard becomes a standard by being used. In order to gently push everybody to standardize on these files we also want to make clear that sooner or later we plan to drop the fallback support for the old configuration files from systemd. That means adoption of this new scheme can happen slowly and piece by piece. But the final goal of only having one set of configuration files must be clear.

Many of these configuration files are relevant not only for configuration tools but also (and sometimes even primarily) in upstream projects. For example, we invite projects like Mono, Java, or WINE to install a drop-in file in /etc/binfmt.d/ from their upstream build systems. Per-distribution downstream support for binary formats would then no longer be necessary and your platform would work the same on all distributions. Something similar applies to all software which need creation/cleaning of certain runtime files and directories at boot, for example beneath the /run hierarchy (i.e. /var/run as it used to be known). These projects should just drop in configuration files in /etc/tmpfiles.d, also from the

upstream build systems. This also helps speeding up the boot process, as separate per-project SysV shell scripts which implement trivial things like registering a binary format or removing/creating temporary/volatile files at boot are no longer necessary. Or another example, where upstream support would be fantastic: projects like X11 could probably benefit from reading the default keyboard mapping for its displays from /etc/vconsole.conf.

Of course, I have no doubt that not everybody is happy with our choice of names (and formats) for these configuration files. In the end we had to pick something, and from all the choices these appeared to be the most convincing. The file formats are as simple as they can be, and usually easily written and read even from shell scripts. That said,/etc/bikeshed.conf could of course also have been a fantastic configuration file name!

So, help us standardizing Linux! Use the new configuration files! Adopt them upstream, adopt them downstream, adopt them all across the distributions!

Oh, and in case you are wondering: yes, all of these files were discussed in one way or another with various folks from the various distributions. And there has even been some push towards supporting some of these files even outside of systemd systems.

Footnotes

[1] Our slogan: "*The only shell that should get started during boot is gnome-shell!*" -- Yes, the slogan needs a bit of work, but you get the idea.