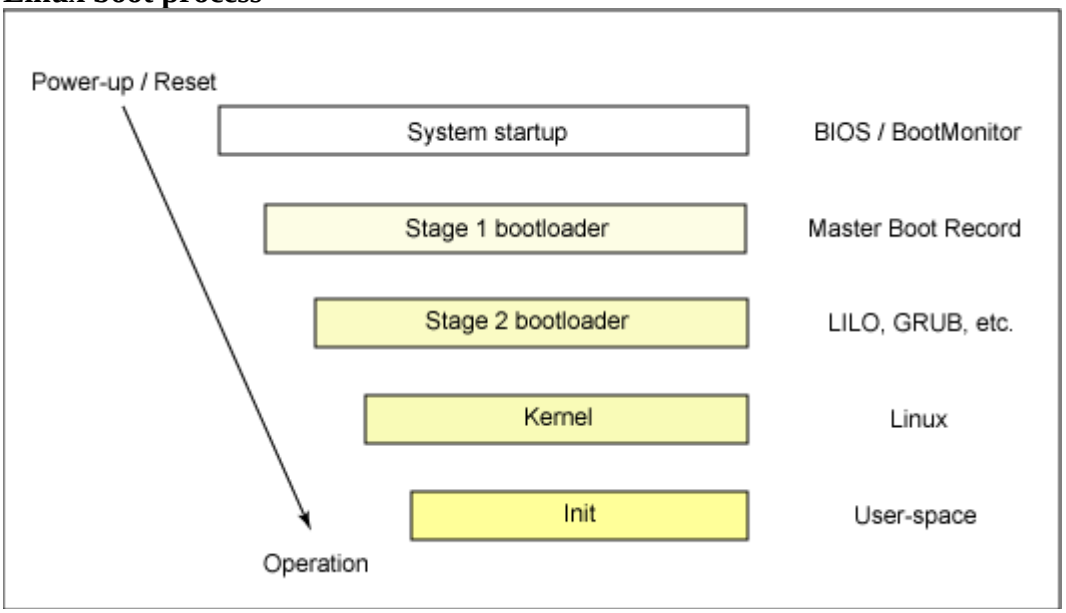
Booting the Linux operating system

Linux boot process



1) The first thing a computer does on start-up is a POST (Power On Self Test). Several devices are tested - processor, memory, graphics card and the keyboard. After compared to the CMOS configuration, the BIOS loads and locates BIOS addressable boot media (from the boot device list).

The job of the POST is to perform a check of the hardware. The second step of the BIOS is local device enumeration and initialization.

2) The basic input/output system (BIOS), which is stored in flash memory on the motherboard. The central processing unit (CPU) in an embedded system invokes the reset vector to start a program at a known address in flash/ROM.The boot block is always at track 0, cylinder 0, head 0 of the boot device.

Here is tested the *boot medium* (hard disk, floppy unit, CD-ROMs). The loader from a ROM loads the *boot sector (block)*, which in turn loads the operating system from the active partition. This block contains the Grub (GNU Grub Unified Boot Loader) for LINUX, which boots the operating system. This boot loader is less than 512 bytes in length (a single sector), and its job is to load the second-stage boot loader GRUB. The boot configuration is *grub.conf* . Grub is installed or at the MBR (Master Boot Record), or at the first sector of the active partition load the operating system

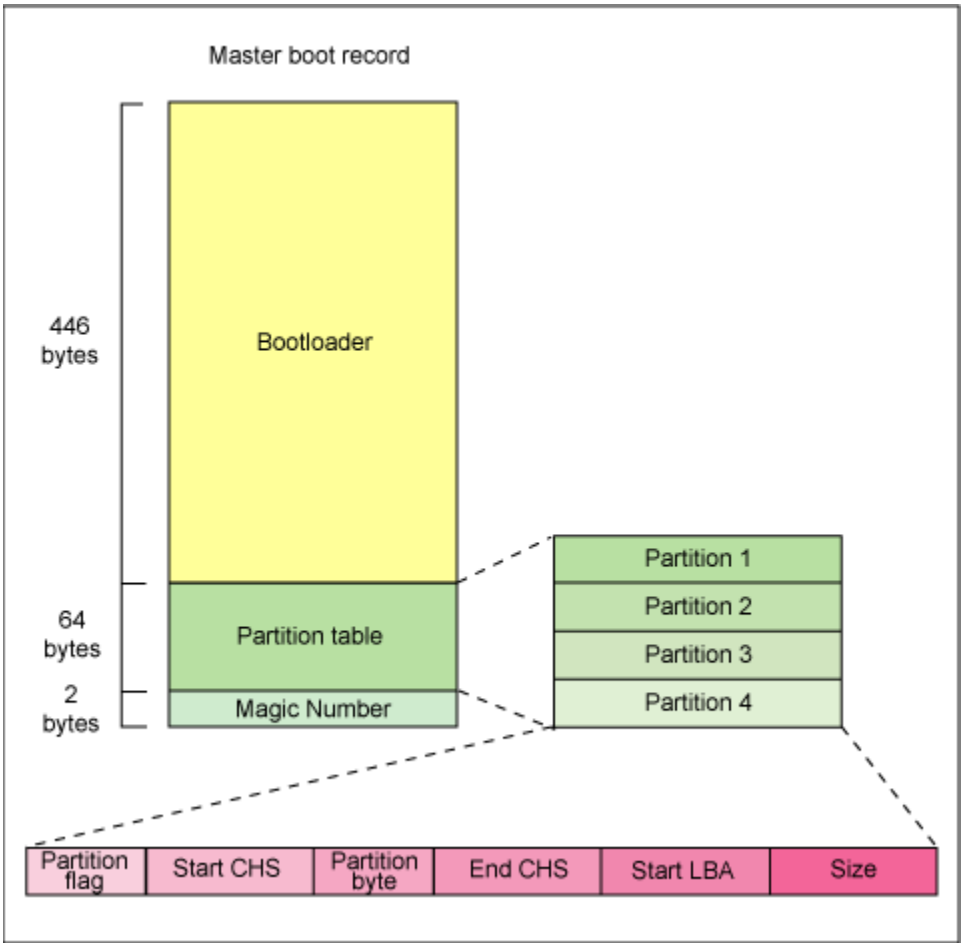
3) When a boot device is found, the first-stage boot loader is loaded into RAM and executed.

The primary boot loader that resides in the MBR is a 512-byte image containing both program code and a small partition table (see Figure 2). The first 446 bytes are the primary boot loader, which contains both executable code and error message text. The next sixty-four bytes are the partition table, which contains a record for each of four partitions (sixteen bytes each). The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR.

The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot record is read from the device into RAM and executed.

A **master boot record** (**MBR**), or **partition sector**, is the 512-byte boot sector that is the first sector ("LBA/absolute sector 0") of a partitioned data storage device such as a hard disk. (The boot sector of a non-partitioned device is a volume boot record. These are usually different, although it is possible to create a record that acts as both; it is called a multiboot record.)

MBR



Structure of a master boot record

Address Hex	Address Oct	Address Dec	Description	Size in bytes
0000	0000	0	code area	**440** (max. 446)
01B8	0670	440	disk signature (optional)	**4**
01BC	0674	444	Usually nulls; 0x0000	**2**
01BE	0676	446	**Table of primary partitions** (Four 16-byte entries, IBM partition table scheme)	**64**
01FE	0776	510	55h	MBR signature; 0xAA55	**2**
01FF	0777	511	AAh	
			MBR, total size: 446 + 64 + 2 =	**512**

The MBR may be used for one or more of the following:

- Holding a disk's primary partition table.
- Bootstrapping operating systems, after the computer's BIOS passes execution to machine code instructions contained within the MBR.
- Uniquely identifying individual disk media, with a 32-bit *disk signature*; even though it may never be used by the machine the disk is running on.

Due to the broad popularity of IBM PC-compatible computers, this type of MBR is widely used, to the extent of being supported by and incorporated into other computer types including newer cross-platform standards for bootstrapping and partitioning.

To see the contents of your MBR, use this command:

```
dd if=/dev/sda of=mbr.bin bs=512 count=1
od -xa mbr.bin
```

4) The secondary, or second-stage, boot loader could be more aptly called the kernel loader The task at this stage is to load the Linux kernel and optional initial RAM disk (initrd). The second-stage boot loader is in RAM and executed a splash screen is commonly displayed, and Linux and an optional initial RAM disk (temporary root file system) are loaded into memory. When the images are loaded, the second-stage boot loader passes control to the kernel image and the kernel is decompressed and initialized. At this stage, the second-stage boot loader checks the system hardware, enumerates the attached hardware devices, mounts the root device, and then loads the necessary kernel modules.

From the GRUB command-line, you can boot a specific kernel with a named initrd image as follows:

grub> **kernel /bzImage-2.6.14.2**
  [Linux-bzImage, setup=0x1400, size=0x29672e]

grub> **initrd /initrd-2.6.14.2.img**
  [Linux-initrd @ 0x5f13000, 0xcc199 bytes]

grub> **boot**

If you don't know the name of the kernel to boot, just type a forward slash (/) and press the Tab key. GRUB will display the list of kernels and initrd images.

The compressed Linux kernel is compressed is located in /boot and and contains a small bit of code which will decompress it and load it into memory.

5) After locating standard devices using initrd and verifying video capability, the kernel verifies hardware configuration (floppy drive, hard disk, network adapters, etc), configures the drivers for the system displaying messages on screen and system log.

During the boot of the kernel, the initial-RAM disk (initrd) that was loaded into memory by the stage 2 boot loader is copied into RAM and mounted. This initrd serves as a temporary root file system in RAM and allows the kernel to fully boot without having to mount any physical disks. Since the necessary modules needed to interface with peripherals can be part of the initrd, the kernel can be very small, but still support a large number of possible hardware configurations. After the kernel is booted, the root file system is pivoted where the initrd root file system is unmounted and the real root file system is mounted.

The initrd function allows you to create a small Linux kernel with drivers compiled as loadable modules. These loadable modules give the kernel the means to access disks and the file systems on those disks, as well as drivers for other hardware assets. Because the root file system is a *file system* on a disk, the initrd function provides a means of bootstrapping to gain access to the disk and mount the real root file system.

6) The kernel tries to mount the *filesystems from /etc/fstab* and the system files. The location of system files is configurable during recompilation, or with other programs - LiLo and rdev. The file system type is automatically detected from the partition table (commonly ext2 and ext3 in LINUX). If the mount fails, a so-called *kernel panic* will occur, and the system will "freeze".

FileSystems are initially mounted in *read-only* mode, to permit a verification of filesystem integrity (fsck) during the mount based on the value of field 6 (= 1) in the /etc/fstab table for the filesystem. This verification isn't indicated if the files were mounted in *read-write* mode. Active mount information is kept in the /etc/mtab file. /etc/fstab is maintained manually by the system administrator. /etc/mtab is maintained by the system

/etc/fstab fields

Name—The name, label, or UUID number of a local block device
Mount point—The name of the directory file that the filesystem/directory hierarchy is to be mounted on.
Type—The type of filesystem/directory hierarchy that is to be mounted as specified in the **mount** command. Local filesystems are of type **ext2**, **ext4**, or **iso9660**, and remote directory hierarchies are of type **nfs** or **cifs**.
Mount options—A comma-separated list of mount options, as specified in the **mount** command.
Dump—Previously used by dump to determine when to back up the filesystem.
Fsck—Specifies the order in which fsck checks filesystems. Root (*/*) is **1,**
Filesystems mounted to a directory just below the root directory should have a **2**.
Filesystems that are mounted on another mounted filesystem (other than root) should have a **3**.

/etc/fstab and mount command options

defaults
ro or **rw** Read only or read write
noauto Do not respond to **mount -a.** Used for external devices CDROMs ...
noexec Executables cannot be started from the device
nosuid Ignore SUID bit throughout the filesystem
nodev Special device files such as block or character devices are ignored
noatime Do not update atimes (performance gain)
owner The device can be mounted only by it's owner
user Implies **noexec**, **nosuid** and **nodev**. A single user's name is added to
mtab so that other users may not unmount the devices
users Same as **user** but the device may be unmounted by any other user
gid Same as **user** but the device may be unmounted by any other group
mode dtdefault file mode for system
soft Used for network file system mounts (NFS) in conjunction with the nofsck option
(/etc/fstab field 6 = 2)

7) The kernel starts *init*, which will become process number 1, and will start the rest of the system. Results of steps 1-7 are displayed with the **dmesg** command

UNIX/LINUX Startup

Linux is an implementation of the UNIX operating system V concept though not actually based on the UNIX Sourcecode License (USL) from AT&T Bell Labs, now owned by Netware (now Attachmate 2010). Some Linux distributions, like SlackWare, use the older BSD init system initialization process, developed at the University of California, Berkeley.

UNIX "Sys V" (sysvinit) initialization process is meant to control the starting and ending of services and/or daemons in a system, and permits different start-up configurations on different execution levels ("run levels").

Results of the sysvinit process are written to **/var/log/messages.** Or the equivalent in UNIX systems.

Current LINUX operating releases use the event based "upstart" method of initialization started by the Ubuntu distribution. Other UNIX releases are also going in this direction replacing the Sys V init process with so called "event" based startup such as Sun Solaris' SMF.

Note that all startup processes are child processes off of PID 1- init. Also you will find some init process run as "sourced" commands – i.r .as ". ./command".

BSD init process (Slackware, FreeBSD, OpenBSD, MacOSX)

Older UNIX system use the BSD INIT process. Basically this consists of several standalone scripts specified in /etc/inittab such as rc.sysinit, rc.network, rc.tcpip in the /etc directory to start different deamon services from a static script.

AT&T System V init process (Most UNIX systems, RHEL prior to FC 9)

The initialization process (*init*) is the parent of all the other processes. This process is the first running process on any Linux/UNIX system, and is started directly by the kernel. It is what loads the rest of the system, and always has a *PID* of 1.

First the *init* examines /etc/inittab to determine what processes have to be launched after. This file provides *init* information on runlevels, and on what process should be launched on each runlevel.

Second *init* looks up the first line with a *sysinit* (system initialization) action and executes the specified command file, in this case /etc/rc.d/rc.sysinit.

Third After the execution of the scripts in /etc/rc.d/rc.sysinit, *init* starts to launch the processes associated with each runlevel:
l3:3:/etc/rc.d/rc3.d:wait
the directoriesare run in sequence up to the the initial runlevel as specified in *initdefault*. Every line runs as a single script (/etc/rc.d/rc), which has a number from 1 to 6 as argument to specify the runlevel.

Standard runlevels

0: Halt (stops all running processes and executes *shutdown*)
1: "*Single-user mode*". The system runs with a reduced set of services and daemons. The root file system is mounted *read-only*.
2: Most of the services run exception of network services (*httpd*, *named*, *nfs*, etc), the filesystems are shared.
3: Multi-user mode, network support enabled. All filesystems available.
4: Unused in most distributions.
5: Complete multi-user mode, with network and graphic subsystem support enabled.
6: Reboot. Stops all running processes and reboots the system to the initial execution level.

The most used action in /etc/inittab is *wait*, which means *init* executes the command file for a specified runlevel, and then waits until that level is terminated.

The commands defined in /etc/inittab are executed only once, by the *init* process, every time when the operating system boots as a succession of commands (sourced) as follows:

- Determine whether the system takes part of a network, depending on the content of /etc/sysconfig/network
- Mount /proc, the file system used in Linux to determine the state of the diverse processes.
- Set the system time settings as retained by the BIOS settings.
- Enables virtual memory, activating and mounting the swap partition, specified in /etc/fstab)
- Sets the host name for the network and system wide authentication, like NIS and so on.
- Verifies the root filesystem, and if no problems, mounts it.
- Verifies the other filesystems specified in /etc/fstab.
- Identifies routines used by the OS to recognize installed hardware to using Plug'n'Play devices (kudzu)
- Verifies the state of special disk devices, like RAID (*Redundant Array of Inexpensive Disks*)
- Mounts all the specified file systems in /etc/fstab.
- Executes other system-specific tasks.

The directory /etc/rc.d/init.d contains all the commands which start or stop services which are associated with all the execution levels. All the files in /etc/rc.d/init.d have a short name which describes the services to which they're associated. For example, /etc/rc.d/init.d/amd starts and stops the *a*uto *m*ount *d*aemon, which mounts the NFS host and devices anytime when needed.

After the *init* process executes all the commands, files and scripts, the last few processes are the /sbin/mingetty ones, which shows the banner and log-in message of the distribution you have installed. The system is loaded and prepared so the user could log in.

Runlevels

The execution levels represent the mode in which the computer operates and are shown by the “runlevel” command. They are defined by a set of available services at any time they are started. The system boots into a runlevel specified in /etc/inittab- i*nitdefault* entry.

To change the current execution level for example to level 3, edit /etc/inittab in a text editor, and edit the following line (do *not* change the initial runlevel to 0 or 6!):

id:3:initdefault:

The most used facility of *init* after boot is to change from one runlevel to an other.
For example, to change the execution level to change the execution level to 3, type: **init 3.**

At the LiLo or Grub prompt you can change te rulevel dynamically before booting the operating system. To boot into runlevel 3, type: **linux 3**.

Runlevel directories

Every execution level has a directory with a symbolic links (*symlinks*) pointing to the corresponding scripts in /etc/rc.d/init.d. These directories are:

/etc/rc.d/rc0.d
/etc/rc.d/rc1.d
/etc/rc.d/rc2.d
/etc/rc.d/rc3.d
/etc/rc.d/rc4.d
/etc/rc.d/rc5.d
/etc/rc.d/rc6.d

The name of the symlinks specify which service has to be stopped, started and when. **The links starting with an "S" are programmed to start in various execution levels.** The links also have a number in their name (01-99). Now some examples of symlinks in the directory /etc/rc.d/rc2.d:

K20nfs -> ../init.d/nfs
K50inet -> ../init.d/inet
S60lpd -> ../init.d/lpd
S80sendmail -> ../init.d/sendmail

When operating systems change the execution level, init compares the list of the terminated processes (links which start with "K", so-called “kill” scripts_) from the directory of the current execution level with the list of processes which have to be started (starting with "S", so-called “start” scripts), found in the destination directory.

To remove a service from a runlevel, you might simply delete or rename the corresponding symlink to something other than beginning with K or S.

To add a service, create a symlink pointing to a corresponding scripts in /etc/rc.d/init.d assigning a number to be started in the proper sequence.

Some symlink commands are referenced repeatedly during the sysvinit process which can lead to long lead times. Hence the development of the “upstart” process.

Boot processing futures

System V boot processing is considered obsolete. It is being replaced by “event based” startup products like Upstart and Systemd in Linux (next sections), SMF in Solaris. Most systems do maintain backward facing interfaces to the original SysV bootup scripts for product installations and maintenance.

Because there is no standard, there are multiple products that substitute for SysV init processing and will remove and link **/sbin/init** to their own program(s). So if you need to know what a UNIX/LINUX system is using for startup processing, do **ls –al /sbin/init** and you’ll know. Though some systems actually maintain both (Solaris).

Upstart (skip this section after FC 15)

Beginning with FC 9, upstart (an Ubuntu developed package) is a replacement for the System V init daemon (sysvinit), the process spawned by the kernel that is responsible for starting, supervising and stopping all other processes on the system. The old approach works as long as you can guarantee when in the boot sequence things are available, so you can place your init script after that point and know that it will work. Typical ordering requirements are:

- Hard drive devices must have been discovered, initialised and partitions detected before we try and mount from /etc/fstab.
- Network devices must have been discovered and initialised before we try and start networking.

This worked ten years ago, why doesn't it work now? The simple answer is that our computer has become far more flexible:

- Drives can be plugged in and removed at any point, e.g. USB drives.
- Storage buses allow more than a fixed number of drives, so they must be scanned for; this operation frequently does not block.
- To reduce power consumption, the drive may not actually be spun up until the bus scan so will not appear for an even longer time.
- Network devices can be plugged in and removed at any point.
- Firmware may need to be loaded after the device has been detected, but before it is usable by the system.
- Mounting a partition in /etc/fstab may require tools in /usr which is a network filesystem that cannot be mounted until after networking has been brought up.

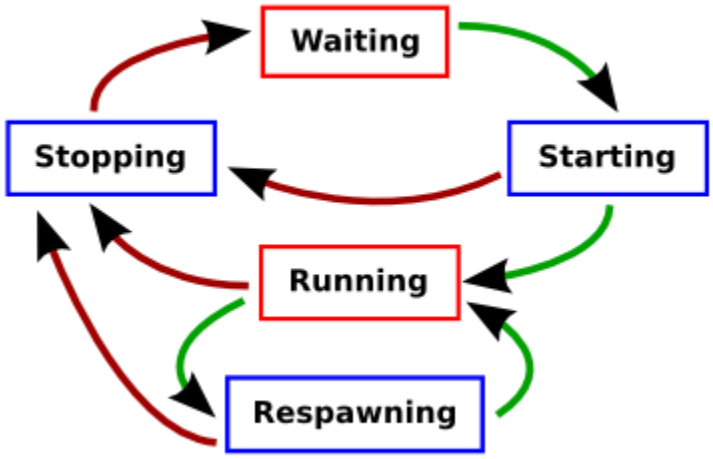
What was needed was an init system that could dynamically order the start up sequence based on the configuration and hardware found as it went along.

upstart is an event-based init daemon; events generated by the system cause jobs to be started and running jobs to be stopped. Events can include things such as:

- the system has started,
- the root filesystem is now writable,
- a block device has been added to the system,
- a filesystem has been mounted,
- at a certain time or repeated time period,
- another job has begun running or has finished,
- a file on the disk has been modified,
- there are files in a queue directory,
- a network device has been detected,
- the default route has been added or removed.

In fact, any process on the system may send events to the init daemon over its control socket (subject to security restrictions, of course) so there is no limit.

Each job has a life-cycle which is shown in the graph below:



The two states shown in red (”waiting” and “running”) are rest states, normally we expect the job to remain in these states until an event comes in, at which point we need to take actual to get the job into the next state. The other states are temporary states; these allow a job to run shell script to prepare for the job itself to be run (”starting”) and clean up afterwards (”stopping”). For services that should be respawned if they terminate before an event that stops them is received, they may run shell script before the process is started again (”respawning”).

Jobs leave a state because the process associated with them terminates (or gets killed) and move to the next appropriate state, following the green arrow if the job is to be started or the red arrow if it is to be stopped. When a script returns a non-zero exit status, or is killed, the job will always be stoped. When the main process terminates and the job should not be respawned, the job will also always be stopped. As already covered, events generated by the init daemon or received from other processes cause jobs to be started or stopped; also manual requests to start or stop a job may be received.

The communication between the init daemon and other processes is bi-directional, so the status of jobs may be queries and even changes of state to all jobs be received.

Upstart differs somewhat from other event based init replacements such as launchd (MacOS X “Open Source” project), Initng by Jimmy Wennlund or SMF developed by Sun for the Solaris operating system.

The goal of upstart is to replace those cron, at and inetd daemons, so that there is only one place (/etc/event.d) where system administrators need to configure when and how jobs should be run.. For compatibility, upstart will continue to run the existing sysvinit scripts for the forseeable future
Compatibility command-line tools that behave like their existing equivalents will also be implemented, a system administrator would never need to know that crontab -e is actually changing upstart jobs.

What Upstart looks like:

The Upstart system comprises five packages, all of which are installed by default:

upstart provides the Upstart init daemon and initctl utility.
upstart-logd provides the logd daemon and the job definition file for the logd service.
upstart-compat-sysv provides job definition files for the rc tasks as well as the reboot, runlevel, shutdown, and telinit utilities that provide compatibility with SysVinit.
startup-tasks provides job definition files for system startup tasks.
system-services provides job definition files for tty services.

There are several terms it helps to understand when talking about init.

An ***event*** is a change in state that init can be informed of. Almost any change in state -- either internal or external to the system -- can trigger an event. For example, the boot loader triggers the startup event, the system entering runlevel 2 triggers the runlevel 2 event, and a filesystem being mounted triggers the path-mounted event. Removing and installing a hotplug or USB device (such as a printer) can trigger an event. You can also trigger an event manually by using the initctl emit command.

A ***job*** is a series of instructions that init reads. The instructions typically include a program (binary file or shell script) and the name of an event. The Upstart init daemon runs the program when the event is triggered. You can run and stop a job manually using the initctl start and stop commands, respectively. Jobs are divided into tasks and services.

A ***task*** is a job that performs its work and returns to a waiting state when it is done.

A ***service*** is a job that does not normally terminate by itself. For example, the logd daemon and the gettys are implemented as services. The init daemon monitors each service, restarting the service if it fails and killing the service when it is stopped manually or by an event.

The /etc/event.d directory holds *job definition files* (files defining the jobs that the Upstart init daemon runs). Initially this directory is populated by Upstart software packages. With Ubuntu releases following Feisty, installing some services will add a file to this directory to control the service, replacing the files that installing a service had placed in the /etc/rc?.d and /etc/init.d directories.

At its core, the Upstart init daemon is a state machine. It keeps track of the state of jobs and, as events are triggered, tracks jobs as they change states. When init tracks a job from one state to another, it may execute the job's commands or terminate the job.

The Upstart init daemon, have no concept of runlevels (nor does Ubuntu as a workstation OS). To ease migration from a runlevel-based system to an event-based system the ***telinit*** command provides compatibility with software intended for sysvinit.

The rc? jobs, which are defined by the /etc/event.d/rc? files, run the /etc/init.d/rc script. This script runs the init scripts in /etc/init.d from the links in the /etc/rc?.d directories, emulating the functionality of these links under SysVinit. The rc? jobs run these scripts as the system enters a runlevel; they take no action when the system leaves a runlevel. Upstart implements the runlevel and telinit utilities to provide compatibility with SysVinit systems.

The initctl (init control) utility allows a system administrator working with root privileges to communicate with the Upstart init daemon. This utility can start, stop, and report on jobs.

Upstart is based on the idea of "jobs" as defined in the /etc/event.d directory. The jobs have names relating to all processes to be run. Each file in the /etc/event.d directory defines a job and usually has at least an event and a command. When the event is triggered, init executes the command. This section describes examples of both administrator-defined jobs and jobs installed with the Upstart packages.

The telinit and shutdown utilities emit runlevel events that include arguments. For example, shutdown emits runlevel 0, and telinit 2 emits runlevel 2. You can match these events within a job definition using the following syntax:

start | stop on *event* [*arg*]

where *event* is an event such as runlevel and *arg* is an optional argument. To stop a job when the system enters runlevel 2, specify stop on runlevel 2. You can also specify runlevel [235] to match runlevels 2, 3, and 5 or runlevel [!2] to match any runlevel except 2.

Although Upstart ignores additional arguments in an event, additional arguments in an event name within a job definition file must exist in the event. For example, runlevel (no argument) in a job definition file matches all runlevel events (regardless of arguments) whereas runlevel S arg2 does not match any runlevel event because the runlevel event takes only one argument.

Here's what a typical directory contains:

```
# rcS - runlevel compatibility
#
# This task runs the old sysv-rc startup scripts.

start on startup

stop on runlevel

# Note: there can be no previous runlevel here, if we have one it's bad
# information (we enter rc1 not rcS for maintenance).  Run /etc/rc.d/rc
# without information so that it defaults to previous=N runlevel=S.
console output
script
      runlevel --set S >/dev/null || true

      /etc/rc.d/rc.sysinit
      runlevel --reboot || true
end script
post-stop script
      if [ "$UPSTART_EVENT" == "startup" ]; then
            runlevel=$(/bin/awk -F ':' '$3 == "initdefault" { print $2 }' /etc/inittab)
            [ -z "$runlevel" ] && runlevel="3"
            for t in $(cat /proc/cmdline); do
                  case $t in
                        -s|single|S|s) runlevel="S" ;;
                        [1-9])       runlevel="$t" ;;
                  esac
            done
            exec telinit $runlevel
      fi
end script
```

```
# rc3 - runlevel 3 compatibility
#
# This task runs the old sysv-rc runlevel 3 (user defined) scripts.  It
# is usually started by the telinit compatibility wrapper.

start on runlevel 3

stop on runlevel

console output
script
      set $(runlevel --set 3 || true)
      if [ "$1" != "unknown" ]; then
          PREVLEVEL=$1
          RUNLEVEL=$2
          export PREVLEVEL RUNLEVEL
      fi

      exec /etc/rc.d/rc 3
end script
```

```
# tty1 - getty
#
# This service maintains a getty on tty1 from the point the system is
# started until it is shut down again.

start on stopped rc2
start on stopped rc3
start on stopped rc4

stop on runlevel 0
stop on runlevel 1
stop on runlevel 6

respawn
exec /sbin/mingetty tty1
```

Systemd

systemd is a replacement for the Linux init daemon, (either System V or BSD-style). It is intended to provide a better framework for expressing services' dependencies, allow more work to be done in parallel (concurrently) at system startup, and to reduce shell overhead. The name comes from the Unix convention of suffixing the names of system daemons (background processes) with the letter "d".

systemd is maintained by Lennart Poettering and Kay Sievers, with many other contributors. It is published as free software under the terms of the GNU Lesser General Public License version 2.1 or later.

Distributions in which systemd is enabled by default:

Fedora since Fedora 15]
Frugalware since Frugalware 1.5
Mageia since Mageia 2
Mandriva since Mandriva 2011
openSUSE since openSUSE 12.1
Arch Linux since October 2012

Distributions where systemd is available, but not enabled by default: Debian GNU/Linux, Gentoo. A conspicuous omission from distros that support systemd is Ubuntu (see Upstart). Support in Red Hat Enterprise Linux 7 is also planned.

Definitions are kept in /etc/systemd.

The following systemctl show command displays the Requires properties of the graphical.target unit. It shows that graphical.target requires multi-user.target:

$ systemctl show --property "Requires" graphical.target

You can also use the systemctl show command to display the Wants properties of a target:
$ systemctl show --property "Wants" multi-user.target

systemd target units as runlevels. Target units are not true runlevels, but they perform a function similar to the function performed by SysVinit runlevels.

Formatting output:

$ systemctl show --property "Wants" multi-user.target | fmt -10

For a specific service:

systemctl show --property "Wants" multi-user.target | fmt -10 | grep ntpd

Using directory cmds:

$ ls -ld /etc/systemd/system/*.wants

$ ls /etc/systemd/system/*.wants/ntpd.service

Managing Services on Linux with systemd

Starting and Stopping Services

systemadm is a nice graphical systemd manager but is still in deveelopment.

You can see the status of everything systemd controls by running systemctl with no options:

$ systemctl

How do you see only available services, running or not?

$ systemctl list-units -t service --all

Want to see only active services?

$ systemctl list-units -t service

You can check the status of any individual service, such as the sshd or cman.service:

$ systemctl status cman.service

$ systemctl status sshd.service

On Fedora you can still use the old service and chkconfig commands or start with the new systemd commands?

This is how to start a service:

systemctl start sshd.service

Or use restart to restart a running service. This stops a service:

systemctl stop sshd.service

Those are in effect only for the current session, so if you want a service to start at boot do this:

systemctl enable sshd.service

And that's all. No hassling with startup scripts. This disables it from starting at boot:

systemctl disable sshd.service

You can check to see if a service is already running; 0 means it is currently running and 1 means it is not:

$ systemctl is-enabled sshd.service; echo $?

You can use systemctl halt, poweroff, or reboot to shutdown or reboot the system. All three send a wall message to users warning that the system is going down.

Processes, cgroups, and Killing

systemd organizes processes with cgroups, and you can see this with the ps command, which has been updated to show cgroups. Run this command to see which service owns which processes:

$ ps xawf -eo pid,user,cgroup,args

cgroups were introduced into the Linux kernel a few years ago, and they are an interesting mechanism for allocating and limiting kernel resources. In systemd, cgroups are used to corral and manage processes. When new processes are spawned they become members of the parent's cgroup. The cgroup is named for the service it belongs to, and services cannot escape from their cgroups so you always know what service they belong to. When you need to kill a service you can kill the cgroup, and snag all of its processes in one swoop instead of playing find-the-fork-or-renamed-process.

Instead of hunting down and killing processes the old-fashioned way, systemd lets you do it in one command:

systemctl kill some_process.service

See also: **service** and **chkconfig** cmds.