The Linux Boot Sequence

* A good understanding of how Linux boots will be useful as the course progresses.

* First, the actions which start network communication occur during the boot sequence. You'll want to know how this happens so that you can control it. Also, the same shell scripts which start networking during the system boot can be used manually to stop and start network communication. This is useful when you're experimenting with the network configuration.

* Second, many of the activities that you'll be doing over the course of the semester require you to modify system files. It's entirely likely that at some point you'll find yourself sitting in front of a workstation that no longer works properly but will still talk to you. It's a little ironic, but a good understanding of how Linux boots may allow you to avoid actually rebooting the workstation. Instead, you can invoke the necessary startup actions manually. This may not seem important in the Network Lab, but it can be critically important if rebooting a production system is just not an option.

* The material in this lecture is drawn from the Linux System Administrator's Guide and various other pieces of documentation, from the boot scripts themselves, and from the source code of some of the programs involved. Much of the documentation is available online from the Linux Documentation Project (www.tldp.org). The System Administrator's Guide contains useful introductory material on Linux and pointers to other introductory material. If you're new to Linux, this is mandatory reading.

It's also good to know that many Linux software packages include documentation in the distribution. Most often this will end up in /usr/share/program or /usr/share/doc.

The locate and find commands are helpful when you're looking for a particular file.

The man and info commands will give you access to much of the online documentation. Try 'man man' and 'info info'.

* Back to the boot process. Let's begin at the beginning. When a PC (or clone) powers up, the hardware transfers control to a program called the BIOS, which resides in ROM. The BIOS is not too bright, but it knows enough to do two important tasks.
The first is what’s called power on self test (POST), a combined sanity check and initialisation of the computer’s hardware. The BIOS scans the components in the system, executing any initialisation actions that are required. In today’s systems, complex components will provide their own initialisation routines, and the BIOS will arrange to execute them.

Finally, the BIOS will scan the list of boot devices looking for a boot loader. Typically the list of boot devices will include the floppy disk drive (if your system still has one), the CD and/or DVD drive, and one or more hard disk drives, in that order. For each device, the BIOS will read the very first sector of the drive, called the boot sector, looking for the first one that provides a boot loader. The boot loader is a slightly more intelligent program whose job is to load the operating system into memory and start it running.

This description captures the spirit of system startup, but falls a fair bit short of the reality of what a modern BIOS and bootloader can do.

For Linux, the common bootloader is GRUB (Grand Unified Bootloader). Your best source of information about GRUB is the documentation that accompanies the software distribution.

GRUB is actually too big to fit in a single sector of a disk. It is divided into two parts: a small piece, placed in the boot sector, whose primary purpose is to find and load the second, larger piece, which completes the boot by finding and loading the operating system.

If your computer is set up to run multiple operating systems (Linux and Windows, for example), this is where you get to choose which one to run.

Once loaded into memory and started by the bootloader, the Linux kernel performs a sequence of actions as it starts:

- It scans the hardware configuration, initialises device drivers, and takes the CPU into protected mode with virtual memory.
- It mounts the root file system (read only).
- Once these preparations are completed, the kernel starts a user process, /sbin/init, which is responsible for starting everything else.

The init program has process id 1, the root of the process tree. All processes running on the computer are descended from init. init remains in existence until the computer is halted, and we’ll see that it has overall responsibility for the state of the system.

For many years, there was one init, now referred to as SysV (Unix System V) init, which dated from the early 1980’s. Recently, there have been
some changes, but all of the newer implementations that I'm familiar with still retain a considerable amount of the SysV structure as 'legacy' support.

**SysV init**

When it is started, *init* looks for the file /etc/inittab for instructions on how to proceed. Here's the inittab from an Ubuntu workstation in the CSIL, edited a bit to reduce the size.

```bash
#/etc/inittab: init(8) configuration.
#
# The default runlevel.
#id:2:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
#si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin
~~:S:respawn:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
#
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin

# What to do when CTRL-ALT-DEL is pressed.
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now

# Action on special keypress (ALT-UpArrow).
#kb::kbrequest:/bin/echo "Keyboard Request--edit /etc/inittab to let this work."

# What to do when the power fails/returns.
pf::powerwait:/etc/init.d/powerfail start
pn::powerfailnow:/etc/init.d/powerfail now
po::powerokwait:/etc/init.d/powerfail stop

# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
```

3 May, 2008
# Note that on most Debian systems tty7 is used by the X Window System, # so if you want to add more getty’s go ahead but skip tty7 if you run X.
1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6

Lines in inittab have the form

```
 id : runlevel(s) : action : command
```

* The *id* is just a convenient identifier, with no particular meaning to init.
* The *command* is a command to be executed, either a program or a shell script.
* *runlevel* and *action* control when and how *command* is executed. This will become clear as we continue.

* The first thing that init looks for in inittab is a line with the keyword ‘sysinit’ in the *action* field. This is the command that should be executed when the system is booting.

* In the Ubuntu 6 distribution, the command associated with this line is the script /etc/init.d/rcS. A quick check shows that rcS actually runs the command ‘/etc/init.d/rc S’. The rc script is a generic script which looks for a directory, /etc/rcN and executes the scripts that it finds in that directory.

* So, if the system is to go to run level 5, rc looks for a directory /etc/rc5.d.

* In the directories /etc/rcN.d are scripts with names of the form [S|K]ddcommand.

* The initial S or K specifies whether the script is used to start (S) or kill (K) a particular service or set of services. When entering a run level, the rc script will execute the kill scripts first, followed by the start scripts.

* The two digits *dd* specify a number between 00 and 99. This number determines the order of execution within the start and kill groups.

* The *command* portion of the file name is chosen to indicate which system or group of systems is manipulated by the script.
If you use `ls -F` or `ls -l` to list one of the `rcN.d` directories, you’ll see that these scripts are actually symbolic links to scripts in `/etc/init.d`.

Typically, each script can be called with one of four parameters: start, stop, status, and restart; some have additional parameters.

But let’s get back to the actions performed as a consequence of the command `rc S`. An incomplete list of functions performed by the shell scripts specified in the `rcS.d` directory includes:

- set system configuration variables for use as the boot sequence progresses, using values held in files in `/etc/default`;
- continue device initialisations: usb, clock, keyboard, and disk handling optimisations;
- enable paging to disk; check the root file system and remount it read/write; check and mount other local file systems; enable logical volume management;
- clean up system status files;
- bring up network interfaces (note that `/etc/sysctl.conf` can be important here);
- mount remote file systems using NFS.

For the purposes of Cmpt 471, the point about network interfaces is one of the key actions in the boot sequence.

When `rc S` finishes executing the shell scripts specified in `/etc/rcS.d`, control returns to `init`, which then looks for another line in `inittab` with the keyword `initdefault` in the `action` field. This entry has no associated `command`; its purpose is to specify the default run level which should be entered to complete the boot process.

This is a good point to explain the concept of a run level. By convention in a unix system, a run level is an integer between 0 and 6 which specifies the overall system state (usually called a mode).

Here’s the conventional set of run levels for Ubuntu Linux:
0: Halt the system; setting the run level to 0 will cause Linux to do an orderly shutdown and halt the computer.

1: Single-user mode; in this state only the kernel and a bare minimum of essential system services are running. This state is used for system maintenance, upgrade, and repair which cannot be performed during normal system operation. (Usually because the system state will not be consistent, and thus unsafe, while the work is in progress. Also, excluding other users avoids unnecessary system activity.)

2 – 5: Multi-user mode; in this state the system is capable of supporting multiple users and provides a GUI interface.

6: Reboot; the system is halted and then an immediate reboot is automatically started.

These run levels are just conventions; while all unix systems follow the same general startup sequence, the specific mapping of run levels to system modes will vary from one to the next. (Fedora, for instance, has several gradations of multi-user, starting from a minimal level of support and adding remote file system mounts and a GUI.)

* While we’re explaining, might as well deal with the remainder of the *action* labels.

- **sysinit** Specifies the command to be performed at boot time.
- **initdefault** Specifies the default run level; no associated command.
- **powerfailnow** Specifies the command to be executed if a power failure is detected.
- **ctrlaltdel** Specifies the command to be executed if the user tries to force a reboot by pressing the ctl-alt-delete key sequence.

- **once** Specifies that the command is to be started exactly once; *init* will not wait for completion.
- **wait** Specifies that the command is to be executed exactly once, and that *init* should wait for it to complete before proceeding. (This is the usual behaviour, also applied to the commands specified in the sysinit, ctrlaltdel, and powerfail lines.)
- **respawn** Start the command and monitor it; if it ever terminates, immediately execute the command again. This behaviour is used for programs which monitor the console and other ports for login attempts.

There are other actions in addition to those listed here. Consult the *man* page for *inittab*.
Once init knows the default run level, it looks for lines which specify that run level in the runlevel field and executes the command specified in the action field for each line.

As with the initial set of actions for run level S, the work for a given run level is handed off to /etc/init.d/rc, which is executed with the run level as the single parameter.

What other functions are controlled by the scripts in /etc/init.d? An incomplete list includes

- Internet services (DNS, SNMP, SSH, plus the meta-daemon xinetd), and a firewall.
- Remote file sharing using Samba.
- Display management (responsible for putting up the GUI login screen).
- The cron daemon, which can run a job at a specified time (handy for scheduling backups, accounting, etc.).
- Printing and logging services.
- The Network Information Service (NIS), formerly known as the yellow pages (YP). This is a set of databases maintained by a central server which can be queried over the network.
- Remote procedure call and remote execution daemons.

There’s one more thing to notice in initab. Down at the bottom of the file are a set of lines with numbers in the id position. These lines control the program which is started up to listen for login attempts.

- For each port where a user can log in to the computer — typically the console and perhaps serial ports connected to terminals — there must be a program which listens for login attempts.
- Debian uses a program called getty; other unix systems have slightly different names. This program is responsible for listening for activity at the port and presenting a login prompt when it senses activity. Once you’ve authenticated yourself with a login id and a password, getty starts up a command shell for you to begin work.

**upstart**

Recently, a new implementation of init, called upstart, is coming into use. For compatibility with the SysV version of init, the upstart binary is also named init. To try and avoid some of the inevitable confusion, the notes in this section will refer to upstart or init and will simply avoid using the name
of the binary. Upstart is used in Ubuntu 7, which is installed on the virtual
workstations in the Network Lab.

* The operation of upstart is based on the concept of events, and actions
  are triggered by the occurrence of events.
* At startup, upstart processes a set of jobs defined by files held in /etc/event.d.
* To get things rolling, upstart will produce a startup event when it
  recognises that it is running as process 1.

* For compatibility with SysV init, the current set of jobs defined for upstart
  are very close to a one-to-one match with the scripts executed by SysV init.

  * There is a job called rcS, triggered by the startup event, which executes /etc/init.d/rcS.
  * When rcS finishes, a stopped rcS event is produced, which triggers
    the execution of the rc-default job. As you might guess from the
    name, this job determines a default run level. It produces a runlevel
    N event.
  * The runlevel N event triggers the execution of a job which runs the
    appropriate rcN script.
  * Finally, there is a set of jobs which initiate getty programs to listen for
    login attempts.

* There are other variations on the init process. Sun Solaris, for example, now
  uses something called the Service Management Facility.

**Beyond Init**

* You'll notice that there are six entries in initab which spawn getty pro-
  cesses to listen for logins.

* Let's deal first with why there are six entries when there's only one console
  terminal on a workstation.

  * Linux provides a facility called a virtual console (often referred to as
    virtual terminals). Most Linux systems are configured for seven.
  * Each provides an entirely independent port, just as if the computer pro-
    vided seven separate i/o connections for terminals. In particular, log-
    ging off on one virtual console does not automatically log you off from
    other virtual consoles.
  * By convention, virtual consoles 1 – 6 are used for command line login.
    Virtual console 7 is used for the GUI login.
To change from one to another, use the key combination ‘alt-Fn’, replacing n with 1 – 7. It’s a little harder to actually escape from the GUI login screen; you’ll need to type ‘ctl-alt-Fn’ to get to one of the other virtual consoles.

Virtual consoles 1 – 6 provide what’s known as a command line login.

Once you’ve authenticated yourself to the system, a command shell is started for you. A unix command shell is the analog of the DOS command shell, but bears about the same relationship as modern man does to a flatworm.

There are many different command shells available; the one that is started for you is determined by your entry in the /etc/passwd file. This first shell is called the login shell.

Here’s a quick taxonomy of unix command shells:

<table>
<thead>
<tr>
<th>Shell</th>
<th>Init File</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>.profile</td>
<td>The original shell, and the lowest common denominator. Still important because the majority of system shell scripts are written for sh.</td>
</tr>
<tr>
<td>ksh</td>
<td>.profile</td>
<td>(Korn shell) Enhanced version of sh</td>
</tr>
<tr>
<td>bash</td>
<td>.bashrc</td>
<td>Greatly enhanced version of sh</td>
</tr>
<tr>
<td>csh</td>
<td>.cshrc</td>
<td>&quot;C&quot; shell. Comparable in power to sh, but with a more friendly syntax.</td>
</tr>
<tr>
<td>tcsh</td>
<td>.tcshrc</td>
<td>&quot;Tenex&quot; C shell. Greatly enhanced version of csh.</td>
</tr>
</tbody>
</table>

sh and its descendants ksh and bash share a common syntactic structure. The primitive operations provided by these shells to break up text strings tend to be character oriented. They offer excellent control of i/o redirection.

csh and its descendant tcsh also share a common syntactic structure, different from sh and descendants. The primitive operations provided by these shells to break up text strings tend to be word oriented. Control of i/o redirection is somewhat clumsy.

Once the login shell is started, it will search for an initialisation file. As shown above, the name varies from shell to shell. (Notice the leading ‘.’ in these file names; they are hidden files, to cut down on the clutter in your home directory. As you’ll see, unix has lots of these.) If there’s no initialisation file in your home directory, a default file will be used, usually found in /etc.
The shell initialisation file is executed every time you start a new command shell (which you'll do often in unix). But the first command shell that's created when you log in (the login shell) also executes a second initialisation file, called `.login` (.bash_login for bash). Again, it will look for this in your home directory; there may or may not be a system default.

When the login shell finishes executing the shell initialisation file and the `.login` file, it will present you with a prompt and you can get down to work.

Virtual console 7 is different; it provides a GUI login interface.

This login screen will look generally familiar — there are spaces to type your id and password, and some option menus.

The screen is presented courtesy of an X Window System display manager, of which Linux provides several to choose from. The one running in CSIL is called gdm, the Gnome Display Manager.

To explain any further about what's going on here, we need to take a step back and talk about the X Window System.