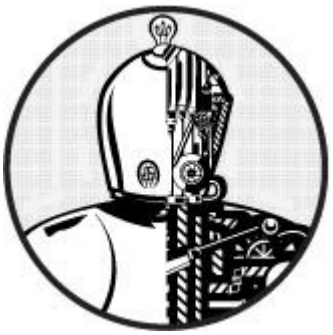
- Don’t run commands in a startup file that print to the standard output.
- Never set `LD_LIBRARY_PATH` in a shell startup file (see 15.1.4 Shared Libraries).

13.7 Further Startup Topics

Because this book deals only with the underlying Linux system, we won’t cover windowing environment startup files. This is a large issue indeed, because the display manager that logs you in to a modern Linux system has its own set of startup files, such as *.xsession*, *.xinitrc*, and the endless combinations of GNOME- and KDE-related items.

The windowing choices may seem bewildering, and there is no one common way to start a windowing environment in Linux. The next chapter describes some of the many possibilities. However, when you determine what your system does, you may get a little carried away with the files that relate to your graphical environment. That’s fine, but don’t carry it over to new users. The same tenet of keeping things simple in shell startup files works wonders for GUI startup files, too. In fact, you probably don’t need to change your GUI startup files at all.

Chapter 14. A Brief Survey of the Linux Desktop



This chapter is a quick introduction to the components found in a typical Linux desktop system. Of all of the different kinds of software that you can find on Linux systems, the desktop arena is one of the wildest and most colorful because there are so many environments and applications to choose from, and most distributions make it relatively easy for you to try them out.

Unlike other parts of a Linux system, such as storage and networking, there isn't much of a hierarchy of layers involved in creating a desktop structure. Instead, each component performs a specific task, communicating with other components as necessary. Some components do share common building blocks (in particular, libraries for graphical toolkits), and these can be thought of as simple abstraction layers, but that's about as deep as it goes.

This chapter offers a high-level discussion of desktop components in general, but we'll look at two pieces in a little more detail: the X Window System, which is the core infrastructure behind most desktops, and D-Bus, an interprocess communication service used in many parts of the system. We'll limit the hands-on discussion and examples to a few diagnostic utilities that, while not terribly useful day-to-day (most GUIs don't require you to enter shell commands in order to interact with them), will help you understand the underlying mechanics of the system and perhaps provide some entertainment along the way. We'll also take a quick look at printing.

14.1 Desktop Components

Linux desktop configurations offer a great deal of flexibility. Most of what the Linux user experiences (the "look and feel" of the desktop) comes from applications or building blocks of applications. If you don't like a particular application, you can usually find an alternative. And if what you're looking for doesn't exist, you can write it yourself. Linux developers tend to have a wide variety of preferences for how a desktop should act, which makes for a lot of choices.

In order to work together, all applications need to have something in common, and at the core of nearly everything on most Linux desktops is the X (X Window System) server. Think of X as sort of the "kernel" of the desktop that manages everything from rendering windows to configuring displays to handling input from devices such as keyboards and mice. The X server is also the one component that you won't easily find a replacement for (see 14.4 The Future of X).

The X server is just a server and does not dictate the way anything should act or appear. Instead, X *client* programs handle the user interface. Basic X client applications, such as terminal windows and web browsers, make connections to the X server and ask to draw windows. In response, the X server figures out where to place the windows and renders them. The X server also channels input back to the client when appropriate.

14.1.1 Window Managers

X clients don't have to act like windowed user applications; they can act as services for other clients or provide

other interface functions. A *window manager* is perhaps the most important client service application because it figures out how to arrange windows on screen and provides interactive decorations like title bars that allow the user to move and minimize windows. These elements are central to the user experience.

There are many window manager implementations. Examples such as Mutter/GNOME Shell and Compiz are meant to be more or less standalone, while others are built into environments such as Xfce. Most window managers included in the standard Linux distributions strive for maximum user comfort, but others provide specific visual effects or take a minimalist approach. There's not likely to ever be a standard Linux window manager because user tastes and requirements are diverse and constantly changing; as a result, new window managers appear all the time.

14.1.2 Toolkits

Desktop applications include certain common elements, such as buttons and menus, called *widgets*. To speed up development and provide a common look, programmers use graphical *toolkits* to provide these elements. On operating systems such as Windows or Mac OS X, the vendor provides a common toolkit, and most programmers use that. On Linux, the GTK+ toolkit is one of the most common, but you'll also frequently see widgets built on the Qt framework and others.

Toolkits usually consist of shared libraries and support files such as images and theme information.

14.1.3 Desktop Environments

Although toolkits provide the user with a uniform outward appearance, some details of a desktop require a degree of cooperation between different applications. For example, one application may wish to share data with another or update a common notification bar on a desktop. To provide for these needs, toolkits and other libraries are bundled into larger packages called *desktop environments*. GNOME, KDE, Unity, and Xfce are some common Linux desktop environments.

Toolkits are at the core of most desktop environments, but to create a unified desktop, environments must also include numerous support files, such as icons and configurations, that make up themes. All of this is bound together with documents that describe design conventions, such as how application menus and titles should appear and how applications should react to certain system events.

14.1.4 Applications

At the top of the desktop are applications, such as web browsers and the terminal window. X applications can range from crude (such as the ancient `xclock` program) to complex (such as the Chrome web browser and LibreOffice suite). These applications normally stand alone, but they often use interprocess communication to become aware of pertinent events. For example, an application can express interest when you attach a new storage device or when you receive new email or an instant message. This communication usually occurs over D-Bus, described in 14.5 D-Bus.

14.2 A Closer Look at the X Window System

The X Window System (*http://www.x.org/*) has historically been very large, with the base distribution including the X server, client support libraries, and clients. Due to the emergence of desktop environments such as GNOME and KDE, the role of the X distribution has changed over time, with the focus now more on the core server that manages rendering and input devices, as well as a simplified client library.

The X server is easy to identify on your system. It's called `X`. Check for it in a process listing; you'll usually see it running with a number of options like this:

```
/usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -
novtswitch
```

The `:0` shown here is called the *display*, an identifier representing one or more monitors that you access with a common keyboard and/or mouse. Usually, the display just corresponds to the single monitor you attach to your computer, but you can put multiple monitors under the same display. When using an X session, the `DISPLAY` environment variable is set to the display identifier.

NOTE

Displays can be further subdivided into screens, such as :0.0 and :0.1, but this has become increasingly rare because X extensions, such as RandR, can combine multiple monitors into one larger virtual screen.

On Linux, an X server runs on a virtual terminal. In this example, the `vt7` argument tells us that it's been told to run on */dev/tty7* (normally, the server starts on the first virtual terminal available). You can run more than one X server at a time on Linux by running them on separate virtual terminals, but if you do, each server needs a unique display identifier. You can switch between the servers with the CTRL-ALT-FN keys or the `chvt` command.

14.2.1 Display Managers

You normally don't start the X server with a command line because starting the server doesn't define any clients that are supposed to run on the server. If you start the server by itself, you'll just get a blank screen. Instead, the most common way to start an X server is with a *display manager*, a program that starts the server and puts a login box on the screen. When you log in, the display manager starts a set of clients, such as a window manager and file manager, so that you can start to use the machine.

There are many different display managers, such as `gdm` (for GNOME) and `kdm` (for KDE). The `lightdm` in the argument list for the X server invocation above is a cross-platform display manager meant to be able to start GNOME or KDE sessions.

To start an X session from a virtual console instead of using a display manager, you can run the `startx` or `xinit` command. However, the session you get will likely be a very simple one that looks completely unlike that of a display manager, because the mechanics and startup files are different.

14.2.2 Network Transparency

One feature of X is network transparency. Because clients talk to the server using a protocol, it's possible to run clients across a network to a server running on a different machine directly over the network, with the X server listening for TCP connections on port 6000. Clients connecting to that port could authenticate, then send windows to the server.

Unfortunately, this method does not normally offer any encryption and is insecure as a result. To close this hole, most distributions now disable the X server's network listener (with the `-nolisten tcp` option to the server, as seen in Note). However, you can still run X clients from a remote machine with SSH tunneling, as described in Chapter 10, by connecting the X server's Unix domain socket to a socket on the remote machine.

14.3 Exploring X Clients

Although one doesn't normally think of working with a graphical user interface from the command line, there are several utilities that allow you to explore the parts of the X Window System. In particular, you can inspect clients as they run.

One of the simplest tools is `xwininfo`. When run without any arguments, it asks you to click on a window:

```
$ xwininfo
```

```
xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.
```

After you click, it prints a list of information about the window, such as its location and size:

```
xwininfo: Window id: 0x5400024 "xterm"

  Absolute upper-left X: 1075
  Absolute upper-left Y: 594
--snip--
```

Notice the window ID here—the X server and window managers use this identifier to keep track of windows. To get a list of all window IDs and clients, use the `xlsclients -l` command.

NOTE

There is a special window called the root window; it's the background on the display. However, you may never see this window (see Desktop Background).

14.3.1 X Events

X clients get their input and other information about the state of the server through a system of events. X events work like other asynchronous interprocess communication events such as udev events and D-Bus events: The X server receives information from a source such as an input device, then redistributes that input as an event to any interested X client.

You can experiment with events with the `xev` command. Running it opens a new window that you can mouse into, click, and type. As you do so, `xev` generates output describing the X events that it receives from the server. For example, here's sample output for mouse movement:

```
$ xev
--snip--
MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937883, (47,174), root:(1692,486),
    state 0x0, is_hint 0, same_screen YES

MotionNotify event, serial 36, synthetic NO, window 0x6800001,
    root 0xbb, subw 0x0, time 43937891, (43,177), root:(1688,489),
    state 0x0, is_hint 0, same_screen YES
```

Notice the coordinates in parentheses. The first pair represents the x-and y-coordinates of the mouse pointer inside the window, and the second (`root:`) is the location of the pointer on the entire display.

Other low-level events include key presses and button clicks, but a few more advanced ones indicate whether the mouse has entered or exited the window, or if the window has gained or lost focus from the window manager. For example, here are corresponding exit and unfocus events:

```
LeaveNotify event, serial 36, synthetic NO, window 0x6800001,
```

```
    root 0xbb, subw 0x0, time 44348653, (55,185), root:(1679,420),
    mode NotifyNormal, detail NotifyNonlinear, same_screen YES,
    focus YES, state 0

FocusOut event, serial 36, synthetic NO, window 0x6800001,
    mode NotifyNormal, detail NotifyNonlinear
```

One common use of `xev` is to extract keycodes and key symbols for different keyboards when remapping the keyboard. Here's the output from pressing the L key; the keycode here is 46:

```
KeyPress event, serial 32, synthetic NO, window 0x4c00001,
    root 0xbb, subw 0x0, time 2084270084, (131,120), root:(197,172),
    state 0x0, keycode 46 (keysym 0x6c, l), same_screen YES,
    XLookupString gives 1 bytes: (6c) "l"
    XmbLookupString gives 1 bytes: (6c) "l"
    XFilterEvent returns: False
```

You can also attach `xev` to an existing window ID with the `-id` *`id`* option. (Use the ID that you get from `xwininfo` as *`id`*) or monitor the root window with `-root`.)

14.3.2 Understanding X Input and Preference Settings

One of the most potentially baffling characteristics of X is that there's often more than one way to set preferences, and some methods may not work. For example, one common keyboard preference on Linux systems is to remap the Caps Lock key to a Control key. There are a number of ways to do this, from making small adjustments with the old `xmodmap` command to providing an entirely new keyboard map with the `setxkbmap` utility. How do you know which ones (if any) to use? It's a matter of knowing which pieces of the system have responsibility, but determining this can be difficult. Keep in mind that a desktop environment may provide its own settings and overrides.

With this said, here are a few pointers on the underlying infrastructure.

Input Devices (General)

The X server uses the *X Input Extension* to manage input from many different devices. There are two basic types of input device—keyboard and pointer (mouse)—and you can attach as many devices as you like. In order to use more than one of the same type of device simultaneously, the X Input Extension creates a "virtual core" device that funnels device input to the X server. The core device is called the master; the physical devices that you plug in to the machine become slaves.

To see the device configuration on your machine, try running the `xinput --list` command:

```
$ xinput --list
  Virtual core pointer                    id=2    [master pointer  (3)]
     Virtual core XTEST pointer           id=4    [slave pointer   (2)]
     Logitech Unifying Device             id=8    [slave pointer   (2)]
  Virtual core keyboard                   id=3    [master keyboard (2)]
```

```
    Virtual core XTEST keyboard      id=5    [slave keyboard  (3)]
    Power Button                     id=6    [slave keyboard  (3)]
    Power Button                     id=7    [slave keyboard  (3)]
    Cypress USB Keyboard             id=9    [slave keyboard  (3)]
```

Each device has an associated ID that you can use with `xinput` and other commands. In this output, IDs 2 and 3 are the core devices, and IDs 8 and 9 are the real devices. Notice that the power buttons on the machine are also treated as X input devices.

Most X clients listen for input from the core devices, because there is no reason for them to be concerned about which particular device originates an event. In fact, most clients know nothing about the X Input Extension. However, a client can use the extension to single out a particular device.

Each device has a set of associated *properties*. To view the properties, use `xinput` with the device number, as in this example:

```
$ xinput --list-props 8
Device 'Logitech Unifying Device. Wireless PID:4026':
        Device Enabled (126): 1
        Coordinate Transformation Matrix (128): 1.000000, 0.000000,
0.000000,
0.000000, 1.000000, 0.000000, 0.000000, 0.000000, 1.000000
        Device Accel Profile (256): 0
        Device Accel Constant Deceleration (257): 1.000000
        Device Accel Adaptive Deceleration (258): 1.000000
        Device Accel Velocity Scaling (259): 10.000000
--snip--
```

As you can see, there are a number of very interesting properties that you can change with the `--set-prop` option (See the xinput(1) manual page for more information.)

Mouse

You can manipulate device-related settings with the `xinput` command, and many of the most useful pertain to the mouse (pointer). You can change many settings directly as properties, but it's usually easier with the specialized `--set-ptr-feedback` and `--set-button-map` options to `xinput`. For example, if you have a three-button mouse at *`dev`* on which you'd like to reverse the order of buttons (this is handy for left-handed users), try this:

```
$ xinput --set-button-map dev 3 2 1
```

Keyboard

The many different keyboard layouts available internationally present particular difficulties for integration into any windowing system. X has always had an internal keyboard-mapping capability in its core protocol that you can manipulate with the `xmodmap` command, but any reasonably modern system uses the XKB (the X keyboard extension) to gain finer control.

XKB is complicated, so much so that many people still use `xmodmap` when they need to make quick changes.

The basic idea behind XKB is that you can define a keyboard map and compile it with the `xkbcomp` command, then load and activate that map in the X server with the `setxkbmap` command. Two especially interesting features of the system are these:

- You can define partial maps to supplement existing maps. This is especially handy for tasks such as changing your Caps Lock key into a Control key, and it is used by many graphical keyboard preference utilities in desktop environments.
- You can define individual maps for each attached keyboard.

Desktop Background

The old X command `xsetroot` allows you to set the background color and other characteristics of the root window, but it produces no effect on most machines because the root window is never visible. Instead, most desktop environments place a big window in the back of all of your other windows in order to enable features such as "active wallpaper" and desktop file browsing. There are ways to change the background from the command line (for example, with the `gsettings` command in some GNOME installations), but if you actually *want* to do this, you probably have too much time on your hands.

xset

Probably the oldest preference command is `xset`. It's not used much anymore, but you can run a quick `xset q` to get the status of a few features. Perhaps the most useful are the screensaver and *Display Power Management Signaling (DPMS)* settings.

14.4 The Future of X

As you were reading the preceding discussion, you may have gotten the feeling that X is a really old system that's been poked at a lot in order to get it to do new tricks. You wouldn't be far off. The X Window System was first developed in the 1980s. Although its evolution over the years has been significant (flexibility was an important part of its original design), you can push the original architecture only so far.

One sign of the age of the X Window System is that the server itself supports an extremely large number of libraries, many for backward compatibility. But perhaps more significantly, the idea of having a server manage clients, their windows, and act as an intermediary for the window memory has become a burden on performance. It's much faster to allow applications to render the contents of their windows directly in the display memory, with a lighter-weight window manager, called a *compositing window manager*, to arrange the windows and do minimal management of the display memory.

A new standard based on this idea, Wayland, has started to gain traction. The most significant piece of Wayland is a protocol that defines how clients talk to the compositing window manager. Other pieces include input device management and an X-compatibility system. As a protocol, Wayland also maintains the idea of network transparency. Many pieces of the Linux desktop now support Wayland, such as GNOME and KDE.

But Wayland isn't the only alternative to X. As of this writing, another project, Mir, has similar goals, though its architecture takes a somewhat different approach. At some point, there will be widespread adoption of at least one system, which may or may not be one of these.

These new developments are significant because they won't be limited to the Linux desktop. Due to its poor performance and gigantic footprint, the X Window System is not suitable for environments such as tablets and smartphones, so manufacturers have so far used alternative systems to drive embedded Linux displays. However, standardized direct rendering can make for a more cost-effective way to support these displays.

14.5 D-Bus

One of the most important developments to come out of the Linux desktop is *the Desktop Bus (D-Bus)*, a message-passing system. D-Bus is important because it serves as an interprocess communication mechanism that allows desktop applications to talk to each other, and because most Linux systems use it to notify processes of system events, such as inserting a USB drive.

D-Bus itself consists of a library that standardizes interprocess communication with a protocol and supporting functions for any two processes to talk to each other. By itself, this library doesn't offer much more than a fancy version of normal IPC facilities such as Unix domain sockets. What makes D-Bus useful is a central "hub" called `dbus-daemon`. Processes that need to react to events can connect to `dbus-daemon` and register to receive certain kinds of events. Processes also create the events. For example, the process `udisks-daemon` listens to ubus for disk events and sends them to `dbus-daemon`, which then retransmits the events to applications interested in disk events.

14.5.1 System and Session Instances

D-Bus has become a more integral part of the Linux system, and it now goes beyond the desktop. For example, both systemd and Upstart have D-Bus channels of communication. However, adding dependencies to desktop tools inside the core system goes against a core design rule of Linux.

To address this problem, there are actually two kinds of `dbus-daemon` instances (processes) that can run. The first is the system instance, which is started by init at boot time with the `--system` option. The system instance usually runs as a D-Bus user, and its configuration file is */etc/dbus-1/system.conf* (though you probably shouldn't change the configuration). Processes can connect to the system instance through the */var/run/dbus/system_bus_socket* Unix domain socket.

Independent of the system D-Bus instance, there is an optional session instance that runs only when you start a desktop session. Desktop applications that you run connect to this instance.

14.5.2 Monitoring D-Bus Messages

One of the best ways to see the difference between the system and session `dbus-daemon` instances is to monitor the events that go over the bus. Try using the `dbus-monitor` utility in system mode like this:

```
$ dbus-monitor --system
signal sender=org.freedesktop.DBus -> dest=:1.952 serial=2 path=/org/
freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
   string ":1.952"
```

The startup message here indicates that the monitor connected and acquired a name. You shouldn't see much activity when you run it like this, because the system instance usually isn't very busy. To see something happen, try plugging in a USB storage device.

By comparison, session instances have much more to do. Assuming you've logged in to a desktop session, try this:

```
$ dbus-monitor --session
```

Now move your mouse around to different windows; if your desktop is D-Bus aware, you should get a flurry of messages indicating activated windows.

14.6 Printing

Printing a document on Linux is a multistage process. It goes like this:

1. The program doing the printing usually converts the document into PostScript form. This step is optional.
2. The program sends the document to a print server.
3. The print server receives the document and places it on a print queue.
4. When the document's turn in the queue arrives, the print server sends the document to a print filter.
5. If the document is not in PostScript form, a print filter might perform a conversion.
6. If the destination printer does not understand PostScript, a printer driver converts the document to a printer-compatible format.
7. The printer driver adds optional instructions to the document, such as paper tray and duplexing options.
8. The print server uses a backend to send the document to the printer.

The most confusing part of this process is why so much revolves around PostScript. PostScript is actually a programming language, so when you print a file using it, you're sending a program to the printer. PostScript serves as a standard for printing in Unix-like systems, much as the *.tar* format serves as an archiving standard. (Some applications now use PDF output, but this is relatively easy to convert.)

We'll talk more about the print format later; first, let's look at the queuing system.

14.6.1 CUPS

The standard printing system in Linux is *CUPS* (*http://www.cups.org/*), which is the same system used on Mac OS X. The CUPS server daemon is called `cupsd`, and you can use the `lpr` command as a simple client to send files to the daemon.

One significant feature of CUPS is that it implements *Internet Print Protocol (IPP)*, a system that allows for HTTP-like transactions among clients and servers on TCP port 631. In fact, if you have CUPS running on your system, you can probably connect to *http://localhost:631/* to see your current configuration and check on any printer jobs. Most network printers and print servers support IPP, as does Windows, which can make setting up remote printers a relatively simple task.

You probably won't be able to administer the system from the web interface, because the default setup isn't very secure. Instead, your distribution likely has a graphical settings interface to add and modify printers. These tools manipulate the configuration files, normally found in */etc/cups*. It's usually best to let these tools do the work for you, because configuration can be complicated. And even if you do run into a problem and need to configure manually, it's usually best to create a printer using the graphical tools so that you have somewhere to start.

14.6.2 Format Conversion and Print Filters

Many printers, including nearly all low-end models, do not understand PostScript or PDF. In order for Linux to support one of these printers, it must convert documents to a format specific to the printer. CUPS sends the document to a Raster Image Processor (RIP) to produce a bitmap. The RIP almost always uses the Ghostscript (`gs`) program to do most of the real work, but it's somewhat complicated because the bitmap must fit the format of the printer. Therefore, the printer drivers that CUPS uses consult the PostScript Printer Definition (PPD) file for the specific printer to figure out settings such as resolution and paper sizes.

14.7 Other Desktop Topics

One interesting characteristic of the Linux desktop environment is that you can generally choose which pieces you want to use and stop using the ones that you dislike. For a survey of many of the desktop projects, have a look at the mailing lists and project links for the various projects at *http://www.freedesktop.org/*. Elsewhere, you'll find other desktop projects, such as Ayatana, Unity, and Mir.

Another major development in the Linux desktop is the Chromium OS open source project and its Google Chrome OS counterpart found on Chromebook PCs. This is a Linux system that uses much of the desktop technology described in this chapter but is centered around the Chromium/ Chrome web browsers. Much of what's found on a traditional desktop has been stripped away in Chrome OS.