# Introduction

Linux is an incredible piece of software. It’s an operating system that’s just as at home running on IBM’s zSeries supercomputers as it is on a cell phone, manufacturing device, network switch, or even cow milking machine. What’s more incredible is that this software is currently maintained by thousands of the best software engineers and it is available for free.

Linux is now installed on millions of systems and is used by home users and professionals alike for a wide range of tasks. From miniature embedded systems in wristwatches to massively parallel mainframes, there are countless ways of exploiting Linux productively. And this makes the sources so interesting. A sound, well-established concept (UNIX) melded with powerful innovations and a strong penchant for dealing with problems that do not arise in academic teaching systems — this is what makes Linux so fascinating.

Technically speaking, Linux is a true UNIX kernel, although it is not a full UNIX operating system because it does not include all the UNIX applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on. However, because most of these programs are freely available under the GPL, they can be installed in every Linux-based system.

## A Brief History of UNIX and C

The first UNIX implementation was developed in 1969 (the same year that Linus Torvalds was born) by Ken Thompson at Bell Laboratories, a division of the tele- phone corporation, AT&T. It was written in assembler for a Digital PDP-7 mini- computer. The name UNIX was a pun on MULTICS (Multiplexed Information and Computing Service), the name of an earlier operating system project in which AT&T collaborated with Massachusetts Institute of Technology (MIT) and General Electric. (AT&T had by this time withdrawn from the project in frustration at its initial failure to develop an economically useful system.) Thompson drew several ideas for his new operating system from MULTICS, including a tree-structured file sys- tem, a separate program for interpreting commands (the shell), and the notion of files as unstructured streams of bytes.

In 1970, UNIX was rewritten in assembly language for a newly acquired Digital PDP-11 minicomputer, then a new and powerful machine. Vestiges of this PDP-11 heritage can be found in various names still used on most UNIX implementations, including Linux.

A short time later, Dennis Ritchie, one of Thompson’s colleagues at Bell Laboratories and an early collaborator on UNIX, designed and implemented the C programming language. This was an evolutionary process; C followed an earlier interpreted language, B. B was initially implemented by Thompson and drew many of its ideas from a still earlier programming language named BCPL. By 1973, C had matured to a point where the UNIX kernel could be almost entirely rewritten in the new language. UNIX thus became one of the earliest operating systems to be written in a high-level language, a fact that made subsequent porting to other hardware architectures possible.

## A Brief History of Linux

In 1991, Linus Torvalds was a computer science student at the University of Helsinki in Finland. He wanted an operating system that was like the UNIX system that he’d grown fond of at the university, but both UNIX and the hardware it ran on were prohibitively expensive. A UNIX version called MINIX was available for free, but it didn’t quite meet his needs. So, as a computer science student, Torvalds studied MINIX and then set out to write a new version himself. In his own words (recorded for posterity on the Internet because this was in an early version of an online chat room), his work was “just a hobby, won’t be big and professional like GNU.”

### The GNU Project

The speed of Linux’s popularity also wouldn’t be possible without the vision of a man whom Steven Levy (author of the book Hackers) refers to as “The Last of the Great MIT AI-LAB Hackers” — in the original sense of the word hackeras someone who plays with code, not the current popular meaning that implies criminal intent. This pioneer and advocate of *freedom* software is Richard Stallman.

The GNU project did not initially produce a working UNIX kernel, but did produce a wide range of other programs. Since these programs were designed to run on a UNIX-like operating system, they could be, and were, used on existing UNIX implementations and, in some cases, even ported to other operating systems. Among the more well-known programs produced by the GNU project are the Emacs text editor, GCC (originally the GNU C compiler, but now renamed the GNU compiler collection, comprising compilers for C, C++, and other languages), the bash shell, and glibc (the GNU C library). By the early 1990s, the GNU project had produced a system that was virtually complete, except for one important component: a working UNIX kernel. The GNU project had started work on an ambitious kernel design, known as the GNU/HURD, based on the Mach microkernel. However, the HURD was far from being in a form that could be released. The GNU Hurd is under active development.

# The Linux Kernel

The term operating system is commonly used with two different meanings:

* To denote the entire package consisting of the central software managing a computer’s resources and all of the accompanying standard software tools, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
* More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

The term kernel is often used as a synonym for the second meaning, and it is with this meaning of the term operating system that Linux is concerned with.

Although it is possible to run programs on a computer without a kernel, the presence of a kernel greatly simplifies the writing and use of other programs, and increases the power and flexibility available to programmers. The kernel does this by providing a software layer to manage the limited resources of a computer by providing a software layer to manage the limited resources of a computer.

## Tasks Performed by Kernel

Among other things, the kernel performs the following tasks:

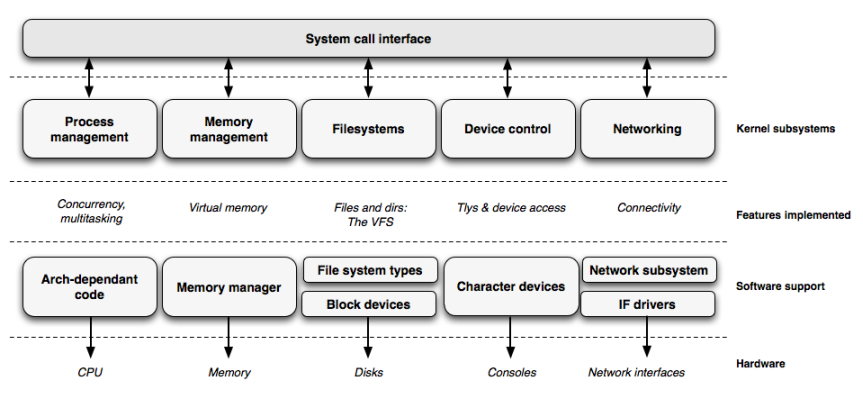
* Process scheduling: A computer has one or more central processing units (CPUs), which execute the instructions of programs. Like other UNIX systems, Linux is a preemptive multitasking operating system, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). Preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time.
* Memory management: While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the kernel must share among processes in an equitable and efficient fashion. Like most modern operating systems, Linux employs virtual memory management, a technique that confers two main advantages:
  + Processes are isolated from one another and from the kernel, so that one process can’t read or modify the memory of another process or the kernel.
  + Only part of a process needs to be kept in memory, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.
* Provision of a file system: The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted and so on.
* Creation and termination of processes: The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an instance of a running program is termed a process. Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.
* Access to devices: The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. The kernel provides programs with an interface that standardizes and simplifies access to devices, while at the same time arbitrating access by multiple processes to each device.
* Networking: The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system.
* Provision of a system call application programming interface (API): Processes can request the kernel to perform various tasks using kernel entry points known as sys

Fig. A split view of kernel

## Process and Kernel Space

A running system typically has numerous processes. For a process, many things happen asynchronously. An executing process doesn’t know when it will next time out, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. The delivery of signals and the occurrence of inter process communication events are mediated by the kernel, and can occur at any time for a process. Many things happen transparently for a process. A process doesn’t know where it is located in RAM or, in general, whether a particular part of its memory space is currently resident in memory or held in the swap area (a reserved area of disk space used to supplement the computer’s RAM). Similarly, a process doesn’t know where on the disk drive the files it accesses are being held; it simply refers to the files by name. A process operates in isolation; it can’t directly communicate with another process. A process can’t itself create a new process or even end its own existence. Finally, a process can’t communicate directly with the input and output devices attached to the computer.

The kernel facilitates the running of all processes on the system. The kernel decides which process will next obtain access to the CPU, when it will do so and for how long. The kernel maintains data structures containing information about all running processes and updates these structures as processes are created, change state, and terminate. The kernel maintains all of the low-level data structures that enable the filenames used by programs to be translated into physical locations on the disk. The kernel also maintains data structures that map the virtual memory of each process into the physical memory of the computer and the swap area(s) on disk. All communication between processes is done via mechanisms provided by the kernel. In response to requests from processes, the kernel creates new processes and terminates existing processes. Lastly, the kernel (in particular, device drivers) performs all direct communication with input and output devices, transferring information to and from user processes as required.

## Linux versus Other Unix-Like Kernels

The various Unix-like systems on the market, some of which have a long history and show signs of archaic practices, differ in many important respects. All commercial variants were derived from either SVR4 or 4.4BSD, and all tend to agree on some common standards like IEEE’s Portable Operating Systems based on Unix (POSIX) and X/Open’s Common Applications Environment (CAE).

The 2.6 version of the Linux kernel aims to be compliant with the IEEE POSIX standard. This, of course, means that most existing UNIX programs can be compiled and executed on a Linux system with very little effort or even without the need for patches to the source code. Moreover, Linux includes all the features of a modern UNIX operating system, such as virtual memory, a virtual filesystem, lightweight processes, UNIX signals, SVR4 interprocess communications, support for Symmetric Multiprocessor (SMP) systems, and so on. The following list describes how Linux competes against some well-known commercial UNIX kernels:

* Monolithic kernel: It is a large, complex do-it-yourself program, composed of several logically different components. In this, it is quite conventional; most commercial UNIX variants are monolithic. (Notable exceptions are the Apple Mac OS X and the GNU Hurd operating systems, both derived from the Carnegie-Mellon’s Mach, which follow a microkernel approach.)
* Compiled and statically linked traditional UNIX kernels: Most modern kernels can dynamically load and unload some portions of the kernel code (typically, device drivers), which are usually called modules. Linux’s support for modules is very good, because it is able to automatically load and unload modules on demand. Among the main commercial UNIX variants, only the SVR4.2 and Solaris kernels have a similar feature.
* Kernel threading: Some UNIX kernels, such as Solaris and SVR4.2/MP, are organized as a set of kernel threads. A kernel thread is an execution context that can be independently scheduled; it may be associated with a user program, or it may run only some kernel functions. Context switches between kernel threads are usually much less expensive than context switches between ordinary processes, because the former usually operate on a common address space. Linux uses kernel threads in a very limited way to execute a few kernel functions periodically; however, they do not represent the basic execution context abstraction. (That’s the topic of the next item.)
* Multithreaded application support: Most modern operating systems have some kind of support for multithreaded applications—that is, user programs that are designed in terms of many relatively independent execution flows that share a large portion of the application data structures. A multithreaded user application could be composed of many lightweight processes (LWP), which are processes that can operate on a common address space, common physical memory pages, common opened files, and so on. Linux defines its own version of lightweight processes, which is different from the types used on other systems such as SVR4 and Solaris. While all the commercial Unix variants of LWP are based on kernel threads, Linux regards lightweight processes as the basic execution context and handles them via the nonstandard clone( ) system call.
* Preemptive kernel: When compiled with the “Preemptible Kernel” option, Linux 2.6 can arbitrarily interleave execution flows while they are in privileged mode. Besides Linux 2.6, a few other conventional, general-purpose UNIX systems, such as Solaris and Mach 3.0, are fully preemptive kernels. SVR4.2/MP introduces some fixed pre-emption points as a method to get limited preemption capability.
* Multiprocessor support: Several Unix kernel variants take advantage of multiprocessor systems. Linux 2.6 supports symmetric multiprocessing (SMP) for different memory models, including NUMA: the system can use multiple processors and each processor can handle any task—there is no discrimination among them. Although a few parts of the kernel code are still serialized by means of a single “big kernel lock,” it is fair to say that Linux 2.6 makes a near optimal use of SMP.
* Filesystem: Linux’s standard filesystems come in many flavors. You can use the plain old Ext2 filesystem if you don’t have specific needs. You might switch to Ext3 if you want to avoid lengthy filesystem checks after a system crash. If you’ll have to deal with many small files, the ReiserFS filesystem is likely to be the best choice. Besides Ext3 and ReiserFS, several other journaling filesystems can be used in Linux; they include IBM AIX’s Journaling File System (JFS) and Silicon Graphics IRIX’s XFS filesystem. Thanks to a powerful object-oriented Virtual File System technology (inspired by Solaris and SVR4), porting a foreign filesystem to Linux is generally easier than porting to other kernels.
* STREAMS: Linux has no analog to the STREAMS I/O subsystem introduced in SVR4, although it is included now in most UNIX kernels and has become the preferred interface for writing device drivers, terminal drivers, and network protocols.

Linux has several features that make it an exciting operating system. Commercial UNIX kernels often introduce new features to gain a larger slice of the market, but these features are not necessarily useful, stable, or productive. As a matter of fact, modern UNIX kernels tend to be quite bloated. By contrast, Linux together with the other open source operating systems—doesn’t suffer from the restrictions and the conditioning imposed by the market, hence it can freely evolve according to the ideas of its designers (mainly Linus Torvalds). Specifically, Linux offers the following advantages over its commercial competitors:

* Linux is cost-free: You can install a complete UNIX system at no expense other than the hardware (of course).
* Linux is fully customizable in all its components: Thanks to the compilation options, you can customize the kernel by selecting only the features really needed. Moreover, thanks to the GPL, you are allowed to freely read and modify the source code of the kernel and of all system programs.
* Linux runs on low-end, inexpensive hardware platforms: You are able to build a network server using an old Intel 80386 system with 4 MB of RAM.
* Linux is powerful: Linux systems are very fast, because they fully exploit the features of the hardware components. The main Linux goal is efficiency, and indeed many design choices of commercial variants, like the STREAMS I/O subsystem, have been rejected by Linus because of their implied performance penalty.
* Linux developers are excellent programmers: Linux systems are very stable; they have a very low failure rate and system maintenance time.
* The Linux kernel can be very small and compact. It is possible to fit a kernel image, including a few system programs, on just one 1.44 MB floppy disk. As far as we know, none of the commercial UNIX variants is able to boot from a single floppy disk.
* Linux is highly compatible with many common operating systems: Linux lets you directly mount filesystems for all versions of MS-DOS and Microsoft Windows, SVR4, OS/2, Mac OS X, Solaris, SunOS, NEXTSTEP, many BSD variants, and so on. Linux also is able to operate with many network layers, such as Ethernet (as well as Fast Ethernet, Gigabit Ethernet, and 10 Gigabit Ethernet), Fiber Distributed Data Interface (FDDI), High Performance Parallel Interface (HIPPI), IEEE 802.11 (Wireless LAN), and IEEE 802.15 (Bluetooth). By using suitable libraries, Linux systems are even able to directly run programs written for other operating systems. For example, Linux is able to execute some applications written for MS-DOS, Microsoft Windows, SVR3 and R4, 4.4BSD, SCO Unix, Xenix, and others on the 80x86 platform.
* Linux is well supported. Believe it or not, it may be a lot easier to get patches and updates for Linux than for any proprietary operating system. The answer to a problem often comes back within a few hours after sending a message to some newsgroup or mailing list. Moreover, drivers for Linux are usually available a few weeks after new hardware products have been introduced on the market. By contrast, hardware manufacturers release device drivers for only a few commercial operating systems—usually Microsoft’s. Therefore, all commercial UNIX variants run on a restricted subset of hardware components.
* Linux supports most of Hardware: Linux tries to maintain a neat distinction between hardware-dependent and hardware- independent source code. To that end, both the arch and the include directories include 23 subdirectories that correspond to the different types of hardware platforms supported. Alpha, arm, arm26, “Code Reduced Instruction Set” CPUs, embedded systems based on microprocessors of the Fujitsu’s FR-V family, Hitachi h8/300 and h8S RISC 8/16-bit microprocessors, i386, ia64, Renesas M32R family of microprocessors, Motorola MC680×0 microprocessors, MIPS microprocessors, Hewlett Packard HP 9000 PA-RISC microprocessors, sparc, sparc64, x86\_64 and many more.

# Devices on Linux

Earlier versions of the Linux kernel offered little basic functionality to the device driver developers: allocating dynamic memory, reserving a range of I/O addresses or an IRQ line, activating an interrupt service routine in response to a device’s interrupt. Older hardware devices, in fact, were cumbersome and difficult to program, and two different hardware devices had little in common even if they were hosted on the same bus. Thus, there was no point in trying to offer a unifying model to the device driver developers. Things are different now. Bus types such as PCI put strong demands on the internal design of the hardware devices; as a consequence, recent hardware devices, even of different classes, support similar functionalities. Drivers for such devices should typically take care of:

* Power management: handling of different voltage levels on the device’s power line.
* Plug and play: transparent allocation of resources when configuring the device.
* Hot-plugging: support for insertion and removal of the device while the system is running.

## Classes of Devices and Modules

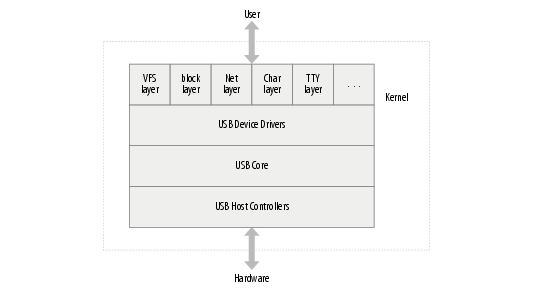
The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendibility. The three classes are:

* Character devices: A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.
* Block devices Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.
* Network interfaces: Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

## **USB Device Basics**

The universal serial bus (USB) is a connection between a host computer and a number of peripheral devices. It was originally created to replace a wide range of slow and different buses—the parallel, serial, and keyboard connections—with a single bus type that all devices could connect to. USB has grown beyond these slow connections and now supports almost every type of device that can be connected to a PC. The latest revision of the USB specification added high-speed connections with a theoretical speed limit of 480 MBps.

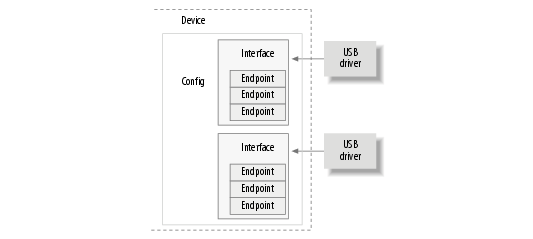
The Linux kernel supports two main types of USB drivers: drivers on a host system and drivers on a device. The USB drivers for a host system control the USB devices that are plugged into it, from the host’s point of view (a common USB host is a desktop computer.) The USB drivers in a device, control how that single device looks to the host computer as a USB device. As the term “USB device drivers” is very confusing, the USB developers have created the term “USB gadget drivers” to describe the drivers that control a USB device that connects to a computer (remember that Linux also runs in those tiny embedded devices, too.)



### Endpoints

USB devices consist of configurations, interfaces, and endpoints and how USB drivers bind to USB interfaces, not the entire USB device. The most basic form of USB communication is through something called an endpoint. A USB endpoint can carry data in only one direction, either from the host computer to the device (called an OUT endpoint) or from the device to the host computer (called an IN endpoint). Endpoints can be thought of as unidirectional pipes. A USB endpoint can be one of four different types that describe how the data is transmitted:

* Control: Control endpoints are used to allow access to different parts of the USB device. They are commonly used for configuring the device, retrieving information about the device, sending commands to the device, or retrieving status reports about the device. These endpoints are usually small in size. Every USB device has a control endpoint called “endpoint 0” that is used by the USB core to configure the device at insertion time. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through to the device.
* Interrupt: Interrupt endpoints transfer small amounts of data at a fixed rate every time the USB host asks the device for data. These endpoints are the primary transport method for USB keyboards and mice. They are also commonly used to send data to USB devices to control the device, but are not generally used to transfer large amounts of data. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through.
* Bulk: Bulk endpoints transfer large amounts of data. These endpoints are usuallymuch larger (they can hold more characters at once) than interrupt endpoints. They are common for devices that need to transfer any data that must get through with no data loss. These transfers are not guaranteed by the USB protocol to always make it through in a specific amount of time. If there is not enough room on the bus to send the whole BULK packet, it is split up across multiple transfers to or from the device. These endpoints are common on printers, storage, and network devices.
* Isochronous: Isochronous endpoints also transfer large amounts of data, but the data is not always guaranteed to make it through. These endpoints are used in devices that can handle loss of data, and rely more on keeping a constant stream of data flowing. Real-time data collections, such as audio and video devices, almost always use these endpoints.



## **Udev and Sysfs**

On UNIX and Unix-like systems, hardware devices are accessed through special files (also called device files or nodes) located in the /dev directory. These files are read from and written to just like normal files, but instead of writing and reading data on a disk, they communicate directly with a kernel driver which then communicates with the hardware. There are many online resources describing /dev files in more detail. Traditionally, these special files were created at install time by the distribution, using the mknod command. In recent years, Linux systems began using udev to manage these /dev files at runtime. For example, udev will create nodes when devices are detected and delete them when devices are removed (including hotplug devices at runtime). This way, the /dev directory contains (for the most part) only entries for devices which actually exist on the system at the current time, as opposed to devices which could exist.

### Udev

Udev is the device manager for the Linux 2.6 kernel that creates/removes device nodes in the /dev directory dynamically. It is the successor of devfs and hotplug. It runs in userspace and the user can change device names using udev rules. Udev depends on the sysfs file system which was introduced in the 2.5 kernel. It is sysfs which makes devices visible in user space. When a device is added or removed, kernel events are produced which will notify udev in user space. The external binary /sbin/hotplug was used in earlier releases to inform udev about device state change. That has been replaced and udev can now directly listen to those events through Netlink. Its features are the following:

* Udev completely run in user space.
* Udev create dynamic entries in /dev.
* Udev provides consistent device naming, if wanted.
* Udev provides a userspace API to access info about current system devices.

### Sysfs

Sysfs is a virtual filesystem exported by the kernel, similar to /proc. The files in Sysfs contain information about devices and drivers. Some files in Sysfs are even writable, for configuration and control of devices attached to the system. Sysfs is always mounted on /sys.

The directories in Sysfs contain the heirarchy of devices, as they are attached to the computer. For example: /sys/class/usb/lp0/device/. Fortunately, Sysfs also provides a large number of symlinks, for easy access to devices without having to know which PCI and USB ports they are connected to. In /sys/class there is a directory for each different class of device.

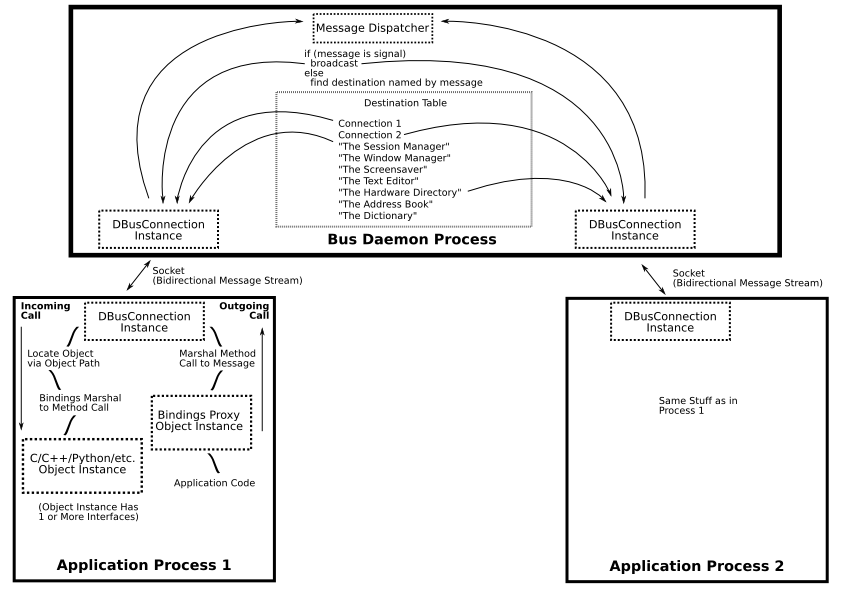
# D-BUS

D-BUS is an interprocess communication (IPC) system, providing simple yet powerful mechanism allowing applications to talk to one another, communicate information and request services. D-BUS was designed from scratch to fulfil the needs of a modern Linux system. D-BUS' initial goal is to be a replacement for CORBA and DCOP, the remote object systems used in GNOME and KDE, respectively. Ideally, D-BUS can become a unified and agnostic IPC mechanism used by both desktops, satisfying their needs and ushering in new features.

## Features of D-BUS

There are many, many technologies in the world that have "Inter-process communication" or "networking" in their stated purpose: CORBA, DCE, DCOM, DCOP, XML-RPC, SOAP, MBUS, Internet Communications Engine (ICE), and probably hundreds more. Each of these is tailored for particular kinds of application. Following features make D-BUS unique:

* The basic unit of IPC in D-BUS is a message, not a byte stream. In this manner, D-BUS breaks up IPC into discrete messages complete with headers (metadata) and a payload (the data). The message format is binary, typed, fully aligned and simple. It is an inherent part of the wire protocol. This approach contrasts with other IPC mechanisms where the lingua franca is a random stream of bytes, not a discrete message.
* D-BUS is bus-based. The simplest form of communication is process to process. D-BUS, however, provides a daemon, known as the message bus daemon, which routes messages between processes on a specific bus. In this fashion, a bus topology is formed, allowing processes to speak to one or more applications at the same time. Applications can send to or listen for various events on the bus.
* A final unique feature is the creation of not one but two of these buses, the system bus and the session bus. The system bus is global, system-wide and runs at the system level. All users of the system can communicate over this bus with the proper permissions, allowing the concept of system-wide events. The session bus, however, is created during user login and runs at the user, or session, level. This bus is used solely by a particular user, in a particular login session, as an IPC and remote object system for the user's applications.



## Language Bindings

Application Programming Interfaces for D-BUS, or bindings, are available in several languages—typically one per language, but not necessarily. Each presents its own API as suits the language, hiding the details of working with D-BUS from the programmer to different extents. The ideal is to fit the D-BUS API into the native language and libraries as naturally as possible.

Using D-BUS should feel more like object-oriented programming than like communication. In some bindings, a programmer may hardly notice that D-BUS is there at all. When that happens, a program that uses D-BUS to communicate will for the most part look as if the counterparts it communicates with were regular components (libraries, modules, packages, objects, functions—whatever the language uses) of the program itself. This is also why some aspects of D-BUS that may seem very basic can differ greatly depending on programming language.

D-BUS bindings are available for an increasing number of languages. There is a low-level C binding, but that is probably too detailed and cumbersome for anything but writing other bindings. A more practical C binding is based on GLib. There are also Java, Perl and Python bindings. There are also D-BUS notes and examples, and so on.

## Buses

There are two major components to D-BUS: a point-to-point communication dbus library, which in theory could be used by any two processes in order to exchange messages among themselves; and a dbus daemon. The daemon runs an actual bus, a kind of "street" that messages are transported over, and to which any number of processes may be connected at any given time. Those processes connect to the daemon using the library, and it probably wouldn't make much sense to use the library for anything else. We'll be looking mostly at the situation where applications (or more generally, clients) connect to a full-blown bus.

Multiple buses may be active simultaneously on a single system. D-BUS was first built to replace the CORBA-like component model underlying the GNOME desktop environment. Similar to DCOP (which is used by KDE), D-BUS is set to become a standard component of the major free desktop environments for GNU/Linux and other platforms. A GNOME environment normally runs two kinds of buses: a single system bus for miscellaneous system-wide communication, e.g. notifications when a new piece of hardware is hooked up; and a session bus used by a single user's ongoing GNOME session. A session bus normally carries traffic under only a single user identity, but D-BUS is aware of user identities and does support flexible authentication mechanisms and access controls. The system bus may see traffic from and to any number of user identities.

## Connections

Every connection to a bus can be addressed on that bus under one or more names. These names are known as the connection's bus names. (Note that bus names are the names of connections on the bus, not names of buses.) Bus names consist of a series of identifiers separated by dots, e.g. "com.acme.Foo" and the identifiers themselves may contain letters, digits, dashes, and underscores. The connection is said to own its bus names.

When a connection is set up, the bus immediately assigns it an immutable bus name that it will retain for as long as the bus exists. This bus name is called a unique connection name, because no other connection will ever have that same name on the same bus--even if the connection is closed down and other ones are created. It can be recognized by the fact that it starts with a colon, which is otherwise not possible: ":34-907" (the other parts of the name have no particular meaning).

A connection may also request additional names, e.g. to offer services under well-known names that are agreed upon by convention. These names must consist of two or more dot-separated elements: "com.acme.PortableHole". Only one connection can hold a given name on the bus at any time, but except for unique connection names, bus names can be relinquished and grabbed by other clients. (Whether a client currently holding it is willing to give it up is, of course, another question, but there are ways of arbitrating this.)

## Object Model

Message exchange on protocols like TCP or UDP is symmetric; in those examples, data is always transferred from one "port" to another. D-BUS presents a more sophisticated model where the sending and the receiving side of a message are never quite of the same type.

In the following we'll borrow from object-oriented terminology. Many terms such as "object" and "method" have more specific meanings in the context of D-BUS, and may have nothing to do with whatever else is going on in client applications. We'll write these terms in italics when they are introduced. All of them are used here only in their D-BUS specific sense, never for their general meanings.

## Objects

One end of any exchange on a bus will always be a communications endpoint that in D-BUS parlance is called an object. An object is created by a client process and exists within the context of that client's connection to the bus. The object is a way for the client process to offer its services on the bus--but one client may create any number of objects.

The bus imposes an object-centric view of communications, where any message carried by the bus is of one of three kinds:

* Requests sent to objects by client processes.
* Replies to requests, going from an object back to a requesting process.
* One-way messages emanating from objects, broadcast to any connected clients that have registered an interest in them. Thus at a higher level of abstraction, the bus supports two forms of communication that we could call "1:1 request-reply" going to an object, and "1:n publish-subscribe" coming from an object.

Every bus has at least one object, representing the bus itself. Clients can obtain information about the status of the bus by sending requests to this object. As you'll see later on, it represents the bus in other useful ways as well.

## Proxies

Objects on the bus can be accessed through references that we call proxies. We call them that because a proxy is a local representation inside your own program of an object that is really accessed through the bus, and typically lives outside your program: you literally access the object "by proxy." Whether you need to know the difference between an object and a proxy depends on how you talk to D-BUS. The Java binding hides the difference, making it look like you're dealing with the objects directly, but the GLib binding makes the existence of proxies very visible and even offers two kinds of proxies. A proxy exists only inside your client, and the details of how it works depend entirely on the binding you use.

Objects have names, also called paths because they look like Unix-style, slash-separated filesystem paths. An object that represents a particular cell in a particular spreadsheet might be called "/org/kde/kspread/sheets/3/cells/4/5", for instance. An object's name needs to be unique only within the context of its connection to the bus. To obtain a proxy to that spreadsheet cell, you would ask the bus to look up object /org/kde/kspread/sheets/3/cells/4/5 for you, to be found in the context of the spreadsheet's connection.

Since any object "lives within" the context of a connection, it takes a combination of that connection's bus name and the object's own name to find it. Once you have found the object you want, if you'll be using it again soon, you'll usually want to keep a proxy to that object around as a variable in your program. That will save you having to look up the object time and again.

Some bindings' proxies may support failover. If you have a proxy to an object exported by some client connected to the bus under a well-known bus name, and that client disconnects (removing the object), reconnects under the same well-known name, and revivies the object, your own program may continue to use the proxy without ever noticing that the object went away for a while. Not all bindings support this, and of the two kinds of proxy in the GLib binding, only one does. It is also not always desirable, e.g. when subsequent operations on an object are meant to be a whole, and it's not acceptable for the object to be disbanded and later reinstituted without your noticing. In those cases, you may need to use a unique connection name in obtaining the proxy rather than a well-known one.

## Methods

When a client sends a request to an object, it sees this request as invoking a method on the object: the object is asked to perform a specific, named action. Normally, if a client tries to invoke a method on an object that the object does not provide, this will raise an error.

The method's definition may require certain information to be passed with the request as arguments (input parameters). For every request, a reply message carries the result back to the requester, along with either result data (output parameters) or, if the action could not be performed, exception information. Exceptions will contain at least an exception name and an error message.

Most D-BUS bindings make all this fit in with their environment's native mechanisms, hiding the finicky details of encapsulating parameters in messages and translating exception messages into exceptions (or whatever the native error-handling mechanism is). For example, passing a string argument to a method of some remote object will look to your program just like passing a string argument to a function in your own program. There is no need for tedious conversions and copying of the data into messages, and there is usually no need to concern yourself with the sending of the underlying message. The binding takes care of all that; the work of encapsulating your data into the messages is called marshalling.

There is one interesting difference with conventional function calls: when sending a request to an object, you don't necessarily have to sit around and wait for a reply. In more complex programs you'll usually find other useful things to do until the method completes. You may want to be ready to handle user interaction, for example, or availability of data from a file or a network connection; you may even have multiple method invocations "in flight" and want to handle the results as they come in, rather than in some pre-defined order. This style of invocation, where you go on to do other things while waiting for an answer, is called asynchronous method invocation. If you use the simple call-and-wait style (synchronous invocation), any other messages that come in while you wait will be queued up and delivered to your program when it's ready for them.

## Signals

The other form of communication also follows the object-oriented mould. Called signals, these one-way communications come from an object and go nowhere in particular. Client processes can register an interest in signals of a particular name coming from a particular object. Whenever an object emits a signal, all interested clients will receive a copy of the signal. There may be one client receiving it, or there may be many--or nobody may be listening. There are no replies to signals: the object emitting the signal would not know how many replies to expect, or where to expect them from. Signals can carry parameters, just like method invocations. Of course, since signals are a strictly one-way form of communication, signals do not have input and output parameters like methods do. More recent versions of D-BUS also allow clients to restrict their interest to cases where certain of the signal's parameters match given values; they will only receive instances of the signal that match those expectations.

Signals are used to publish the occurrence of events that clients may be interested in, such as the closing of some other client's connection to the bus. That particular kind of signal is sent by the object representing the bus itself. Because of this, the event can be announced properly regardless of whether the departing client closed its connection in an orderly fashion, or was killed, or crashed spectacularly.

## Interfaces

So every object supports particular methods and may emit particular signals. These are known collectively as the object's members.

All of an object's members are specified in interfaces. Like their namesakes in the Java language, interfaces are sets of declarations. "Implementing" an interface is tantamount to promising to provide all methods specified in the interface, and announcing the availability of its signals for listeners. Each of these members must accept and/or provide parameters exactly as specified in the interface.

Any object may implement a given interface, just as in Java any number of classes may implement the same interface. Conversely, a single object may implement any number of interfaces. (With D-BUS it probably wouldn't make much sense to have an object that implemented no interfaces at all, though, which is perfectly normal with Java classes.) The combination of all interfaces supported by the object is called the object's type.

When a client invokes a method or listens for a signal, it must indicate the object and the member it is referring to. In addition to object and member, the client may also name the interface in which that member was specified. This can be necessary in some cases. If an object implements two interfaces, for example, that both specify a method named foo, then the object may have separate implementations for foo in the two interfaces. When a client tries to invoke foo on the object without specifying which interface it had in mind, there is no guarantee as to which of the two foo methods will be invoked. The D-BUS implementation may even refuse to carry the request message in the first place. Similarly, you wouldn't want to receive signals that looked like what you're listening for but are really different ones that happened to have the same name.

Older versions of D-BUS also had a bug where request or signal messages could be lost if they failed to specify an interface. Whether there is "overloading" of members within interfaces, i.e. whether multiple members of the same interface may have the same name, depends on the binding.

# Conclusion

Linux is a great piece of software as some might say “the greatest ever written”. This report is about different functionality provided by the kernel to support interaction between its activity and user space especially in recognizing the presence and absence of devices in system using D-BUS and udev. For demo purpose I build a USB blocking application, it has a user interface written entirely in python (uses PyGTK, python-dbus, glib and python in-built libraries) and a service (daemon) written in C (uses Glib libudev and glib-dbus). The code is on GitHub (ketan936/college-works) please feel free to fork it.