Anatomy of the Linux kernel

History and architectural decomposition

M. Tim Jones
Consultant Engineer
Emulex Corp.

06 June 2007

The Linux® kernel is the core of a large and complex operating system, and while it's huge, it is well organized in terms of subsystems and layers. In this article, you explore the general structure of the Linux kernel and get to know its major subsystems and core interfaces. Where possible, you get links to other IBM articles to help you dig deeper.

Given that the goal of this article is to introduce you to the Linux kernel and explore its architecture and major components, let's start with a short tour of Linux kernel history, then look at the Linux kernel architecture from 30,000 feet, and, finally, examine its major subsystems. The Linux kernel is over six million lines of code, so this introduction is not exhaustive. Use the pointers to more content to dig in further.

A short tour of Linux history

Linux or GNU/Linux?

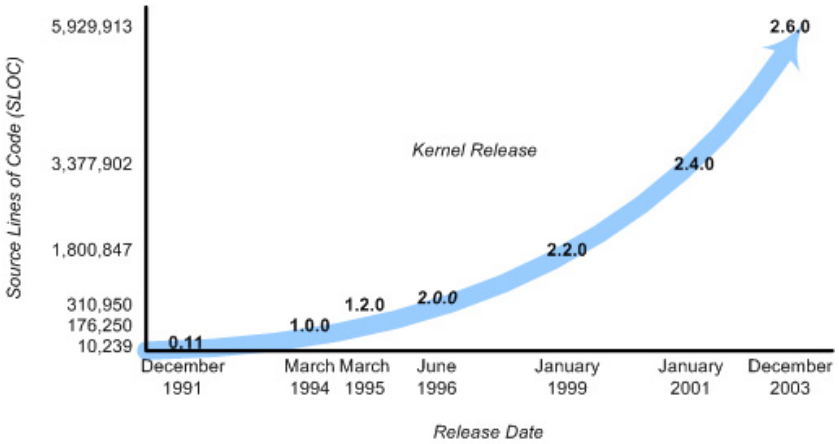
You've probably noticed that Linux as an operating system is referred to in some cases as "Linux" and in others as "GNU/Linux." The reason behind this is that Linux is the *kernel* of an operating system. The wide range of applications that make the operating system useful are the *GNU software*. For example, the windowing system, compiler, variety of shells, development tools, editors, utilities, and other applications exist outside of the kernel, many of which are GNU software. For this reason, many consider "GNU/Linux" a more appropriate name for the operating system, while "Linux" is appropriate when referring to just the kernel.

While Linux is arguably the most popular open source operating system, its history is actually quite short considering the timeline of operating systems. In the early days of computing, programmers developed on the bare hardware in the hardware's language. The lack of an operating system meant that only one application (and one user) could use the large and expensive device at a time. Early operating systems were developed in the 1950s to provide a simpler development experience. Examples include the General Motors Operating System (GMOS) developed for the IBM 701 and the FORTRAN Monitor System (FMS) developed by North American Aviation for the IBM 709.

In the 1960s, Massachusetts Institute of Technology (MIT) and a host of companies developed an experimental operating system called Multics (or Multiplexed Information and Computing Service) for the GE-645. One of the developers of this operating system, AT&T, dropped out of Multics and developed their own operating system in 1970 called Unics. Along with this operating system was the C language, for which C was developed and then rewritten to make operating system development portable.

Twenty years later, Andrew Tanenbaum created a microkernel version of UNIX®, called MINIX (for minimal UNIX), that ran on small personal computers. This open source operating system inspired Linus Torvalds' initial development of Linux in the early 1990s (see Figure 1).

Figure 1. Short history of major Linux kernel releases

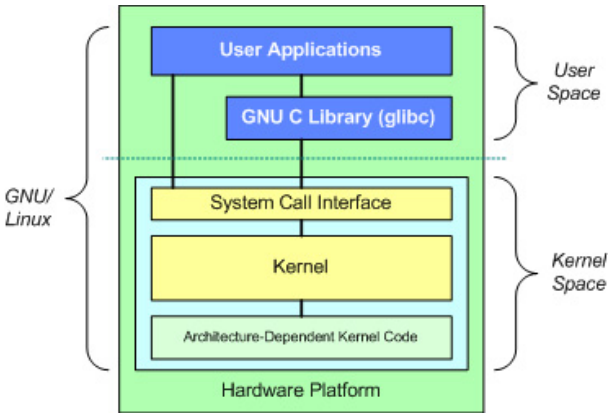


Linux quickly evolved from a single-person project to a world-wide development project involving thousands of developers. One of the most important decisions for Linux was its adoption of the GNU General Public License (GPL). Under the GPL, the Linux kernel was protected from commercial exploitation, and it also benefited from the user-space development of the GNU project (of Richard Stallman, whose source dwarfs that of the Linux kernel). This allowed useful applications such as the GNU Compiler Collection (GCC) and various shell support.

Introduction to the Linux kernel

Now on to a high-altitude look at the GNU/Linux operating system architecture. You can think about an operating system from two levels, as shown in Figure 2.

Figure 2. The fundamental architecture of the GNU/Linux operating system



Methods for system call interface (SCI)

In reality, the architecture is not as clean as what is shown in Figure 2. For example, the mechanism by which system calls are handled (transitioning from the user space to the kernel space) can differ by architecture. Newer x86 central processing units (CPUs) that provide support for virtualization instructions are more efficient in this process than older x86 processors that use the traditional int 80h method.

At the top is the user, or application, space. This is where the user applications are executed. Below the user space is the kernel space. Here, the Linux kernel exists.

There is also the GNU C Library (glibc). This provides the system call interface that connects to the kernel and provides the mechanism to transition between the user-space application and the kernel. This is important because the kernel and user application occupy different protected address spaces. And while each user-space process occupies its own virtual address space, the kernel occupies a single address space. For more information, see the links in the Resources section.

The Linux kernel can be further divided into three gross levels. At the top is the system call interface, which implements the basic functions such as `read` and `write`. Below the system call interface is the kernel code, which can be more accurately defined as the architecture-independent kernel code. This code is common to all of the processor architectures supported by Linux. Below this is the architecture-dependent code, which forms what is more commonly called a BSP (Board Support Package). This code serves as the processor and platform-specific code for the given architecture.

Properties of the Linux kernel

When discussing architecture of a large and complex system, you can view the system from many perspectives. One goal of an architectural decomposition is to provide a way to better understand the source, and that's what we'll do here.

The Linux kernel implements a number of important architectural attributes. At a high level, and at lower levels, the kernel is layered into a number of distinct subsystems. Linux can also be considered monolithic because it lumps all of the basic services into the kernel. This differs from a microkernel architecture where the kernel provides basic services such as communication,

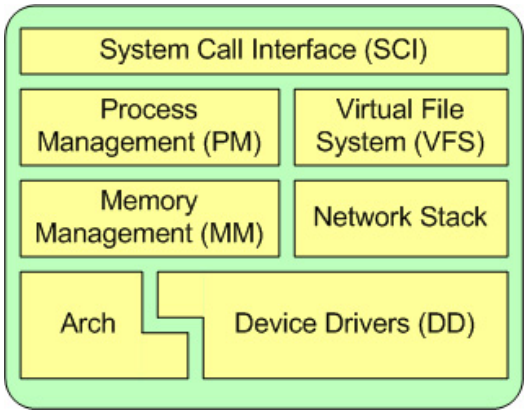
I/O, and memory and process management, and more specific services are plugged in to the microkernel layer. Each has its own advantages, but I'll steer clear of that debate.

Over time, the Linux kernel has become efficient in terms of both memory and CPU usage, as well as extremely stable. But the most interesting aspect of Linux, given its size and complexity, is its portability. Linux can be compiled to run on a huge number of processors and platforms with different architectural constraints and needs. One example is the ability for Linux to run on a process with a memory management unit (MMU), as well as those that provide no MMU. The uClinux port of the Linux kernel provides for non-MMU support. See the Resources section for more details.

Major subsystems of the Linux kernel

Now let's look at some of the major components of the Linux kernel using the breakdown shown in Figure 3 as a guide.

Figure 3. One architectural perspective of the Linux kernel



System call interface

The SCI is a thin layer that provides the means to perform function calls from user space into the kernel. As discussed previously, this interface can be architecture dependent, even within the same processor family. The SCI is actually an interesting function-call multiplexing and demultiplexing service. You can find the SCI implementation in ./linux/kernel, as well as architecture-dependent portions in ./linux/arch. More details for this component are available in the Resources section.

Process management

What is a kernel?

As shown in Figure 3, a kernel is really nothing more than a resource manager. Whether the resource being managed is a process, memory, or hardware device, the kernel manages and arbitrates access to the resource between multiple competing users (both in the kernel and in user space).

Process management is focused on the execution of processes. In the kernel, these are called *threads* and represent an individual virtualization of the processor (thread code, data, stack, and CPU registers). In user space, the term *process* is typically used, though the Linux implementation

does not separate the two concepts (processes and threads). The kernel provides an application program interface (API) through the SCI to create a new process (fork, exec, or Portable Operating System Interface [POSIX] functions), stop a process (kill, exit), and communicate and synchronize between them (signal, or POSIX mechanisms).

Also in process management is the need to share the CPU between the active threads. The kernel implements a novel scheduling algorithm that operates in constant time, regardless of the number of threads vying for the CPU. This is called the O(1) scheduler, denoting that the same amount of time is taken to schedule one thread as it is to schedule many. The O(1) scheduler also supports multiple processors (called Symmetric MultiProcessing, or SMP). You can find the process management sources in ./linux/kernel and architecture-dependent sources in ./linux/arch). You can learn more about this algorithm in the Resources section.

Memory management

Another important resource that's managed by the kernel is memory. For efficiency, given the way that the hardware manages virtual memory, memory is managed in what are called *pages* (4KB in size for most architectures). Linux includes the means to manage the available memory, as well as the hardware mechanisms for physical and virtual mappings.

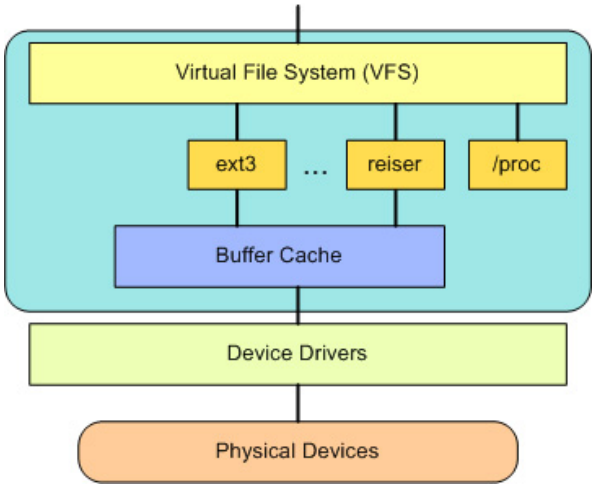
But memory management is much more than managing 4KB buffers. Linux provides abstractions over 4KB buffers, such as the slab allocator. This memory management scheme uses 4KB buffers as its base, but then allocates structures from within, keeping track of which pages are full, partially used, and empty. This allows the scheme to dynamically grow and shrink based on the needs of the greater system.

Supporting multiple users of memory, there are times when the available memory can be exhausted. For this reason, pages can be moved out of memory and onto the disk. This process is called *swapping* because the pages are swapped from memory onto the hard disk. You can find the memory management sources in ./linux/mm.

Virtual file system

The virtual file system (VFS) is an interesting aspect of the Linux kernel because it provides a common interface abstraction for file systems. The VFS provides a switching layer between the SCI and the file systems supported by the kernel (see Figure 4).

Figure 4. The VFS provides a switching fabric between users and file systems



At the top of the VFS is a common API abstraction of functions such as open, close, read, and write. At the bottom of the VFS are the file system abstractions that define how the upper-layer functions are implemented. These are plug-ins for the given file system (of which over 50 exist). You can find the file system sources in ./linux/fs.

Below the file system layer is the buffer cache, which provides a common set of functions to the file system layer (independent of any particular file system). This caching layer optimizes access to the physical devices by keeping data around for a short time (or speculatively read ahead so that the data is available when needed). Below the buffer cache are the device drivers, which implement the interface for the particular physical device.

Network stack

The network stack, by design, follows a layered architecture modeled after the protocols themselves. Recall that the Internet Protocol (IP) is the core network layer protocol that sits below the transport protocol (most commonly the Transmission Control Protocol, or TCP). Above TCP is the sockets layer, which is invoked through the SCI.

The sockets layer is the standard API to the networking subsystem and provides a user interface to a variety of networking protocols. From raw frame access to IP protocol data units (PDUs) and up to TCP and the User Datagram Protocol (UDP), the sockets layer provides a standardized way to manage connections and move data between endpoints. You can find the networking sources in the kernel at ./linux/net.

Device drivers

The vast majority of the source code in the Linux kernel exists in device drivers that make a particular hardware device usable. The Linux source tree provides a drivers subdirectory that is further divided by the various devices that are supported, such as Bluetooth, I2C, serial, and so on. You can find the device driver sources in ./linux/drivers.

Architecture-dependent code

While much of Linux is independent of the architecture on which it runs, there are elements that must consider the architecture for normal operation and for efficiency. The ./linux/arch subdirectory defines the architecture-dependent portion of the kernel source contained in a number of subdirectories that are specific to the architecture (collectively forming the BSP). For a typical desktop, the i386 directory is used. Each architecture subdirectory contains a number of other subdirectories that focus on a particular aspect of the kernel, such as boot, kernel, memory management, and others. You can find the architecture-dependent code in ./linux/arch.

Interesting features of the Linux kernel

If the portability and efficiency of the Linux kernel weren't enough, it provides some other features that could not be classified in the previous decomposition.

Linux, being a production operating system and open source, is a great test bed for new protocols and advancements of those protocols. Linux supports a large number of networking protocols, including the typical TCP/IP, and also extension for high-speed networking (greater than 1 Gigabit Ethernet [GbE] and 10 GbE). Linux also supports protocols such as the Stream Control Transmission Protocol (SCTP), which provides many advanced features above TCP (as a replacement transport level protocol).

Linux is also a dynamic kernel, supporting the addition and removal of software components on the fly. These are called dynamically loadable kernel modules, and they can be inserted at boot when they're needed (when a particular device is found requiring the module) or at any time by the user.

A recent advancement of Linux is its use as an operating system for other operating systems (called a hypervisor). Recently, a modification to the kernel was made called the Kernel-based Virtual Machine (KVM). This modification enabled a new interface to user space that allows other operating systems to run above the KVM-enabled kernel. In addition to running another instance of Linux, Microsoft® Windows® can also be virtualized. The only constraint is that the underlying processor must support the new virtualization instructions. See the Resources section for more information.

Going further

This article just scratched the surface of the Linux kernel architecture and its features and capabilities. You can check out the Documentation directory that's provided in every Linux distribution for detailed information about the contents of the kernel. Be sure to check out the Resources section at the end of this article for more detailed information about many of the topics discussed here.

Resources

Learn

- The GNU site describes the GNU GPL that covers the Linux kernel and most of the useful applications provided with it. Also described is a less restrictive form of the GPL called the Lesser GPL (LGPL).
- UNIX, MINIX and Linux are covered in Wikipedia, along with a detailed family tree of the operating systems.
- The GNU C Library, or glibc, is the implementation of the standard C library. It's used in the GNU/Linux operating system, as well as the GNU/Hurd microkernel operating system.
- uClinux is a port of the Linux kernel that can execute on systems that lack an MMU. This allows the Linux kernel to run on very small embedded platforms, such as the Motorola DragonBall processor used in the PalmPilot Personal Digital Assistants (PDAs).
- "Kernel command using Linux system calls" (developerWorks, March 2007) covers the SCI, which is an important layer in the Linux kernel, with user-space support from glibc that enables function calls between user space and the kernel.
- "Inside the Linux scheduler" (developerWorks, June 2006) explores the new O(1) scheduler introduced in Linux 2.6 that is efficient, scales with a large number of processes (threads), and takes advantage of SMP systems.
- "Access the Linux kernel using the /proc filesystem" (developerWorks, March 2006) looks at the /proc file system, which is a virtual file system that provides a novel way for user-space applications to communicate with the kernel. This article demonstrates /proc, as well as loadable kernel modules.
- "Server clinic: Put virtual filesystems to work" (developerWorks, April 2003) delves into the VFS layer that allows Linux to support a variety of different file systems through a common interface. This same interface is also used for other types of devices, such as sockets.
- "Inside the Linux boot process" (developerWorks, May 2006) examines the Linux boot process, which takes care of bringing up a Linux system and is the same basic process whether you're booting from a hard disk, floppy, USB memory stick, or over the network.
- "Linux initial RAM disk (initrd) overview" (developerWorks, July 2006) inspects the initial RAM disk, which isolates the boot process from the physical medium from which it's booting.
- "Better networking with SCTP" (developerWorks, February 2006) covers one of the most interesting networking protocols, Stream Control Transmission Protocol, which operates like TCP but adds a number of useful features such as messaging, multi-homing, and multi-streaming. Linux, like BSD, is a great operating system if you're interested in networking protocols.
- "Anatomy of the Linux slab allocator" (developerWorks, May 2007) covers one of the most interesting aspects of memory management in Linux, the slab allocator. This mechanism originated in SunOS, but it's found a friendly home inside the Linux kernel.
- "Virtual Linux" (developerWorks, December 2006) shows how Linux can take advantage of processors with virtualization capabilities.
- "Linux and symmetric multiprocessing" (developerWorks, March 2007) discusses how Linux can also take advantage of processors that offer chip-level multiprocessing.

- "Discover the Linux Kernel Virtual Machine" (developerWorks, April 2007) covers the recent introduction of virtualization into the kernel, which turns the Linux kernel into a hypervisor for other virtualized operating systems.
- Check out Tim's book *GNU/Linux Application Programming* for more information on programming Linux in user space.
- In the developerWorks Linux zone, find more resources for Linux developers, including Linux tutorials, as well as our readers' favorite Linux articles and tutorials over the last month.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through our developer blogs, forums, podcasts, and community topics.

About the author

M. Tim Jones

M. Tim Jones is an embedded software engineer and the author of *GNU/Linux Application Programming, AI Application Programming* (now in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.