ANATOMY OF THE LINUX KERNEL

The Linux® kernel is the core of a large and complex operating system, and while it is huge, it is well organized in terms of subsystems and layers. In this article, you can 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

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, the 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.

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 operating system the software that manages the sharing of the resources of a computer and provides programmers with an interface used to access those resources 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.

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 the 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 understand the source better 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 buffer a region of memory used to temporarily hold data while it is being moved from one place to another VFS (Virtual File System) an abstraction layer on top of a more concrete file system 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 of 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.

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

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 there is a 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.

thread – подпроцесс, scheduler – планировщик (программа)