ISSN 0976 – 1411

Available online at www.internationaleJournals.com

International eJournals

INTERNATIONAL eJOURNAL OF MATHEMATICS AND ENGINEERING

www.InternationaleJournals.com

International eJournal of Mathematics and Engineering 114 (2011) 1036 - 1045

EXPLOSER ON LINUX KERNEL ARCHITECTURE

T. V. SURYA NARAYANA, V. MURALIDHAR, K. S. VIJAYA SIMHA
Tenali Engineering College, Anumarllapudi (v), Guntur
Don Bosco Engineering college, Guntur
Email: vmdha.research.@gmail..com, tvenkata.surya@gmail.com, s.kollam@.yahoo.com

Abstract:

This paper describes the abstract or conceptual software architecture of the Linux kernel. This level of architecture is concerned with the large-scale subsystems within the kernel, but not with particular procedures or variables. One of the purposes of such an abstract architecture is to form a mental model for Linux developers and architects. The model may not reflect the as-built architecture perfectly, but it provides a useful way to think about the overall structure. This model is most useful for entry-level developers, but is also a good way for experienced developers to maintain a consistent and accurate system vocabulary. The architecture presented here is the result of reverse engineering an existing Linux implementation; the primary sources of information used were the documentation and source code. The Linux kernel is composed of five main subsystems that communicate using procedure calls. Four of these five subsystems are discussed at the module interconnection level. At all times the relation of particular subsystems to the overall Linux system is considered. The architecture of the kernel is one of the reasons that many users have successfully adopted Linux. In particular, the Linux kernel architecture was designed to support a large number of volunteer developers. Further, the subsystems that are most likely to need enhancements were architected to easily support extensibility. These two qualities are factors in the success of the overall system.

Keywords: Device Driver, I-Node or index nodes, Network File System (NFS), Process, Ram disk, Swapping, Task

Introduction:

The goal of this paper is to present the *abstract* architecture of the Linux kernel. By concentrating on high-level design, this architecture is useful to entry-level developers that need to see the high level architecture before understanding where their changes fit in. In addition, the conceptual architecture is a good way to create a formal system vocabulary that is shared by experienced developers and system designers. This architectural description may not perfectly reflect the actual implementation architecture, but can provide a useful mental model for all developers to share. Ideally, the conceptual architecture should be created before the system is implemented, and should be updated to be an ongoing system conscience.

Challenges of this Paper

This presentation is somewhat unusual, in that the conceptual architecture is usually formed before the as-built architecture is complete; this paper is the result of reverse engineering, kernel source and documentation. A few architectural descriptions were used, but these descriptions were also based on the existing system implementation. By deriving the conceptual

architecture from an existing implementation, this paper probably presents some implementation details of conceptual architecture.

In addition, the mechanisms used to derive the information in this paper omitted the best source of information, the live knowledge of the system architects and developers. Only in this way can an accurate mental model of the system architecture be described. Despite these problems, this paper offers a useful conceptualization of the Linux kernel software, although it cannot be taken as an accurate depiction of the system as implemented.

System Architecture

The Linux kernel is useless in isolation; it participates as one part in a larger system that, as a hole, is useful. As such, it makes sense to discuss the kernel in the context of the entire system.

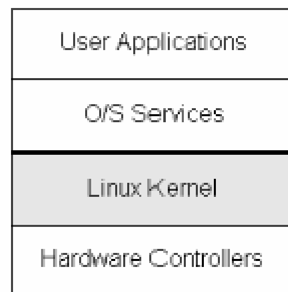

Figure: Shows a decomposition of the entire Linux operating system.

The Linux operating system is composed of four major subsystems:

1. **User Applications:** The set of applications in use on a particular Linux system will be different depending on what the computer system is used for, but typical examples include a word processing application and a web-browser.
2. **O/S Services:** These are services that are typically considered part of the operating system (a windowing system, command shell, etc.); also, the programming interface to the kernel compiler tool and library) is included in this subsystem.
3. **Linux Kernel:** This is the main area of interest in this paper; the kernel abstracts and mediates access to the hardware resources, including the CPU.
4. **Hardware Controllers:** This subsystem is comprised of all the possible physical devices in a Linux installation; for example, the CPU, memory hardware, hard disks, and network hardware is all members of this subsystem

This decomposition follows each subsystem layer can only communicate with the subsystem layers that are immediately adjacent to it. In addition, the dependencies between subsystems are from the top down, layers pictured near the top depend on lower layers, but subsystems nearer the bottom do not depend on higher layers. Since the primary interest of this paper is the Linux kernel, we will completely ignore the User Applications subsystem, and only consider the Hardware and O/S Services subsystems to the extent that they interface with the Linux kernel subsystem.

Purpose of the Kernel

The Linux kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge of what physical hardware is installed on a computer the Linux kernel abstracts all hardware into a consistent virtual interface. In addition, Linux supports multi-tasking in a manner that is transparent to user processes, each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and is responsible for mediating access to hardware resources so that each process has fair access while inter-process security is maintained.

Overview of the Kernel Structure

The Linux kernel is composed of five main subsystems:

1. The **Process Scheduler** is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time.

2. The **Memory Manager** (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.

3. The **Virtual File System** (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.

4. The **Network Interface** (NET) provides access to several networking standards and a variety of network hardware.

5. The **Inter-Process Communication** (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

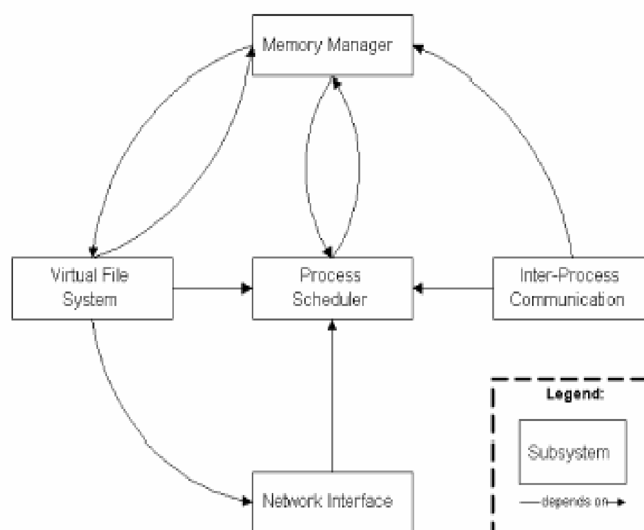

Figure: Kernel Subsystem Overview

This diagram emphasizes that the most central subsystem is the process scheduler: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and resume the process when the operation is finished. For example, when a process attempts to send a message across the network, the network interface may need to suspend the process until the hardware has completed sending the message successfully. After the message has been sent (or the hardware returns a failure), the network interface then resumes the process with a return code indicating the success or failure of the operation. The other subsystems (memory manager, virtual file system, and inter-process communication) all depend on the process scheduler for similar reasons.

The other dependencies are somewhat less obvious, but equally important:

- The process-scheduler subsystem uses the memory manager to adjust the hardware memory map for a specific process when that process is resumed.
- The inter-process communication subsystem depends on the memory manager to support a shared-memory communication mechanism. This mechanism allows two processes to access an area of common memory in addition to their usual private memory.
- The virtual file system uses the network interface to support a network file system (NFS), and also uses the memory manager to provide a ram disk device.
- The memory manager uses the virtual file system to support swapping; this is the only reason that the memory manager depends on the process scheduler. When a process accesses memory that is currently swapped out, the memory manager makes a request to the file system to fetch the memory from persistent storage, and suspends the process.

In addition to the dependencies that are shown explicitly, all subsystems in the kernel rely on some common resources that are not shown in any subsystem. These include procedures that all kernel subsystems use to allocate and free memory for the kernel's use, procedures to print warning or error messages, and system debugging routines. These resources will not be referred to explicitly since they are assumed ubiquitously available and used within the kernel layer.

Each of the depicted subsystems contains state information that is accessed using a procedural interface, and the subsystems are each responsible for maintaining the integrity of their managed resources.

Process Scheduler Architecture

Goals: The process scheduler is the most important subsystem in the Linux kernel. Its purpose is to control access to the computer's CPU(s). This includes not only access by user processes, but also access for other kernel subsystems.

Modules: The scheduler is divided into four main modules:

1. The scheduling policy module is responsible for judging which process will have access to the CPU; the policy is designed so that processes will have fair access to the CPU.
2. Architecture-specific modules are designed with a common interface to abstract the details of any particular computer architecture. These modules are responsible for communicating with a CPU to suspend and resume a process. These operations involve knowing what registers and state information need to be preserved for each process and executing the assembly code to effect a suspend or resume operation.

3. The architecture-independent module communicates with the policy module to determine which process will execute next, then calls the architecture-specific module to resume the appropriate process. In addition, this module calls the memory manager to ensure that the memory hardware is restored properly for the resumed process.

4. The system call interface module permits user processes access to only those resources that are explicitly exported by the kernel. This limits the dependency of user processes on the kernel to a well-defined interface that rarely changes, despite changes in the implementation of other kernel modules.

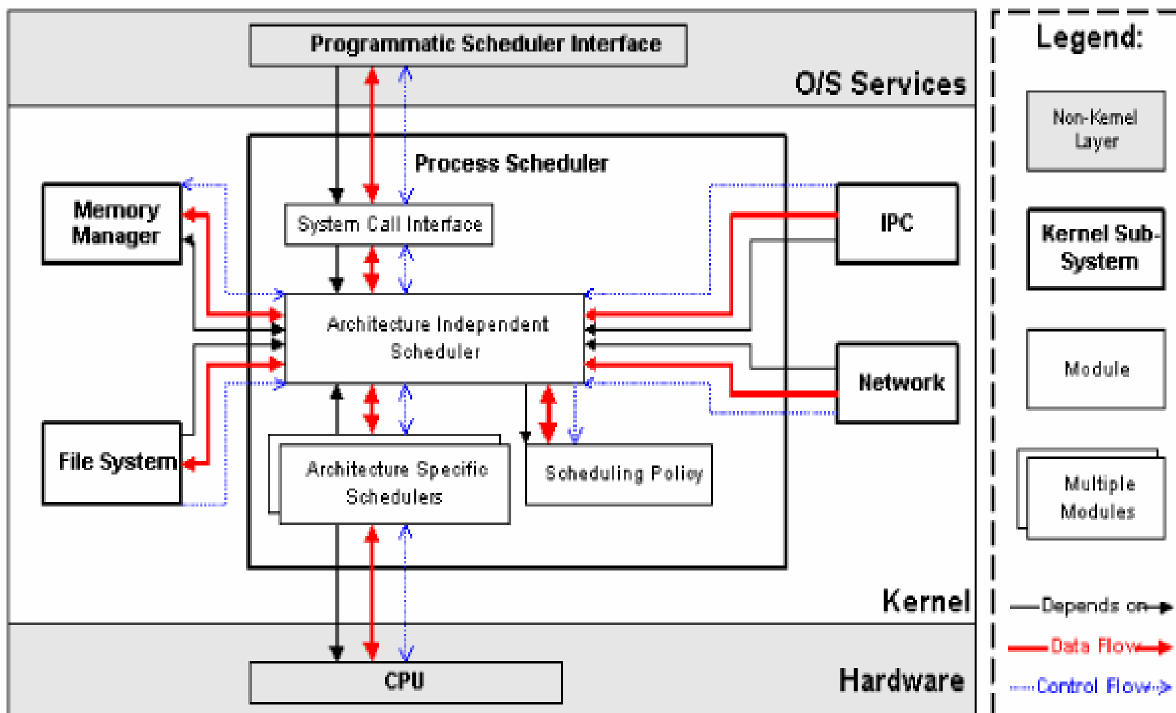

Figure: Process Scheduler Subsystem in Context

Representation

The scheduler maintains a data structure, the task list, with one entry for each active process. This data structure contains enough information to suspend and resume the processes, but also contains additional accounting and state information. This data structure is publicly available throughout the kernel layer

Memory Management Architecture

Goals: The memory management subsystem is responsible for controlling process access to the hardware memory resources. This is accomplished through a hardware memory-management system that provides a mapping between process memory references and the machine's physical memory. The memory manager subsystem maintains this mapping on a per process basis, so that two processes can access the same virtual memory address and actually use different physical memory locations. In addition, the memory manager subsystem supports swapping; it moves unused memory pages to persistent storage to allow the computer to support more virtual memory than there is physical memory.

Modules

The memory manager subsystem is composed of three modules:

- The architecture specific module presents a virtual interface to the memory management hardware.
- The architecture independent manager performs all of the per-process mapping and virtual memory swapping. This module is responsible for determining which memory pages will be evicted when there is a page fault, there is no separate policy module since it is not expected that this policy will need to change.
- A system call interface is provided to provide restricted access to user processes. This interface allows user processes to allocate and free storage, and also to perform memory mapped file I/O.

Data Representation

The memory manager stores a per-process mapping of physical addresses to virtual addresses. This mapping is stored as a reference in the process scheduler's task list data structure. In addition to this mapping, additional details in the data block tell the memory manager how to Fetch and store pages. For example, executable code can use the executable image as a backing store, but dynamically allocated data must be backed to the system-paging file. Finally, the memory manager stores permissions and accounting information in this data structure to ensure System security.

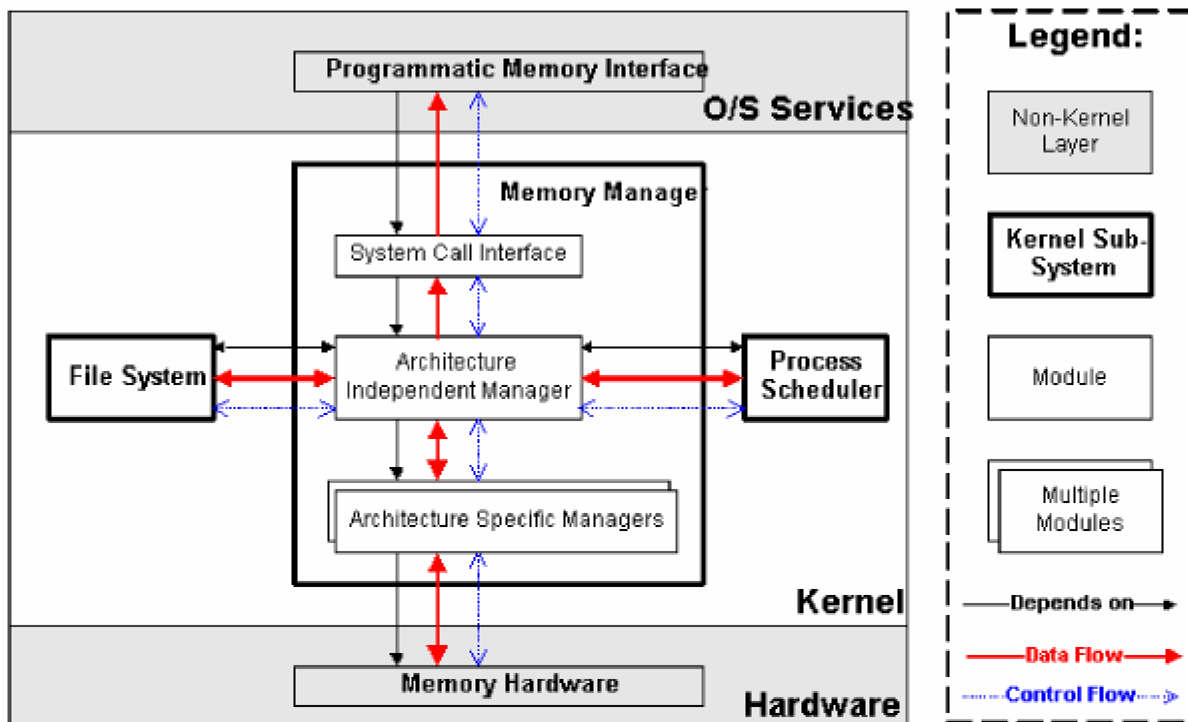

Figure: Memory Manager subsystem in context

Virtual File System Architecture

Goals: The virtual file system is designed to present a consistent view of data as stored on hardware devices. Almost all hardware devices in a computer are represented using a generic device driver interface. The virtual file system goes further, and allows the system administrator to mount any of a set of logical file systems on any physical device. Logical file systems promote compatibility with other operating system standards, and permit developers to implement file systems with different policies. The virtual file system abstracts the details of both physical device and logical file system, and allows user processes to access files using a common interface, without necessarily knowing what physical or logical system the file resides on. In addition to traditional file-system goals, the virtual file system is also responsible for loading new executable programs. This responsibility is accomplished by the logical file system module, and this allows Linux to support several executable formats.

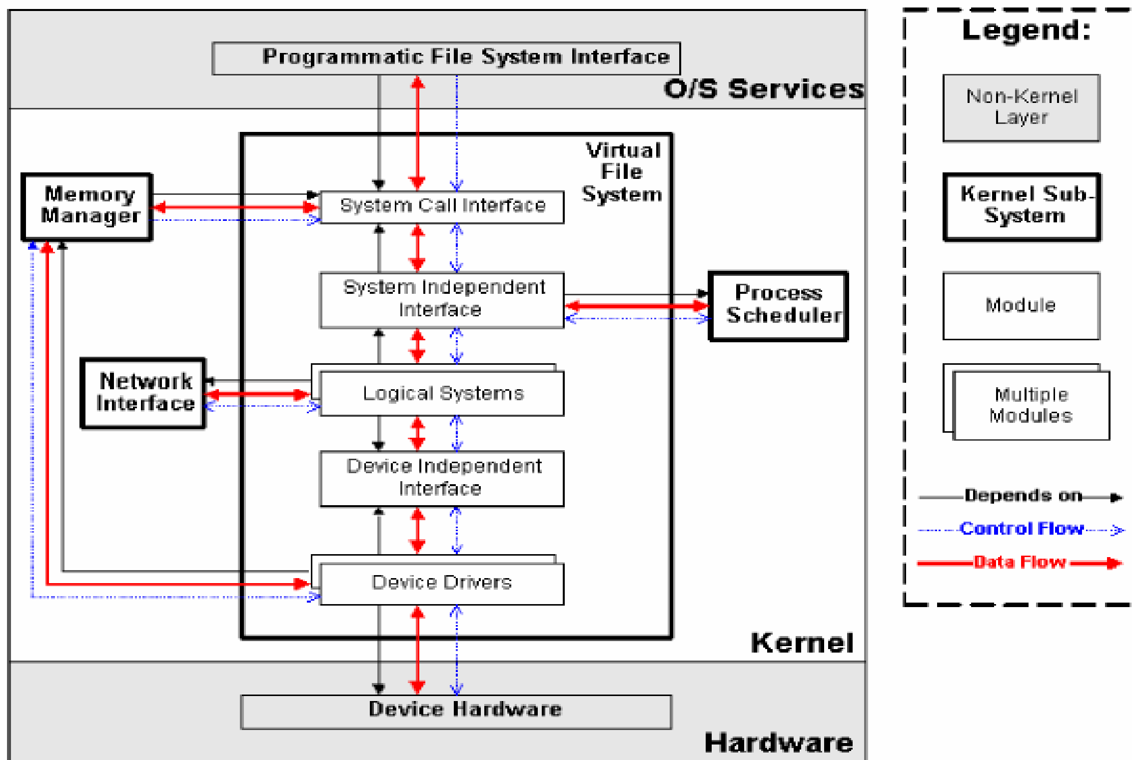

Figure: Virtual File System in Context

Modules:

1. There is one device driver module for each supported hardware controller. Since there are a large number of incompatible hardware devices, there are a large number of device drivers. The most common extension of a Linux system is the addition of a new device driver.
2. The Device Independent Interface module provides a consistent view of all devices.
3. There is one logical file system module for each supported file system.The system independent interface presents a hardware and logical-file-system independent view of the hardware resources. This module presents all resources using either a block-oriented or character-oriented file interface.

Finally, the system call interface provides controlled access to the file system for user processes. The virtual file system exports only specific functionality to user processes.

Data Representation

All files are represented using i-nodes. Each i-node structure contains location information for specifying where on the physical device the file blocks are. In addition, the i-node stores pointers to routines in the logical file system module and device driver that will perform required read and write operations. By storing function pointers in this fashion, logical file systems and device drivers can register themselves with the kernel without having the kernel depend on any specific module.

Network Interface Architecture

Goals: The network subsystem allows Linux systems to connect to other systems over a network. There are a number of possible hardware devices that are supported, and a number of network protocols that can be used. The network subsystem abstracts both of these implementation details so that user processes and other kernel subsystems can access the network without necessarily knowing what physical devices or protocol is being used.

Modules

1. Network device drivers communicate with the hardware devices. There is one device driver module for each possible hardware device.
2. The device independent interface module provides a consistent view of all of the hardware devices so that higher levels in the subsystem don't need specific knowledge of the hardware in use.
3. The network protocol modules are responsible for implementing each of the possible network transport protocols.
4. The protocol independent interface module provides an interface that is independent of hardware devices and network protocol. This is the interface module that is used by other kernel subsystems to access the network without having a dependency on particular protocols or hardware.

Finally, the system calls interface module restricts the exported routines that user processes can access.

Data Representation

Each network object is represented as a socket. Sockets are associated with processes in the same way that i-nodes are associated; sockets can be share amongst processes by having both of the task data structures pointing to the same socket data structure.

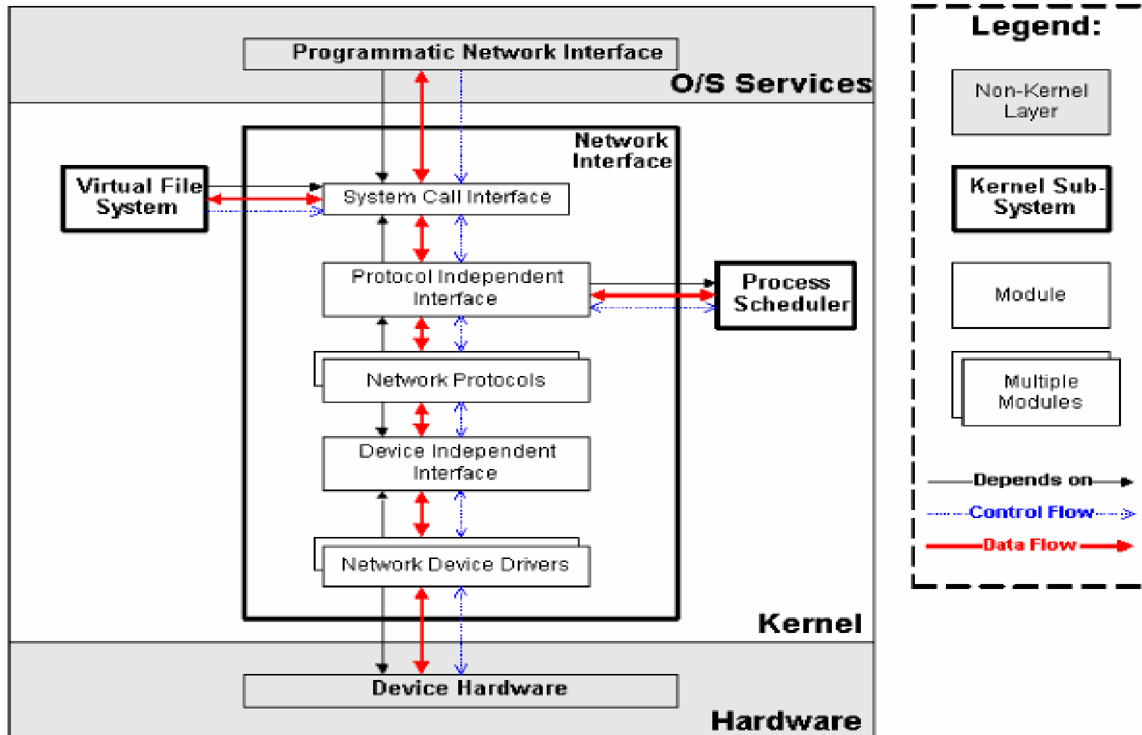

Figure: Network Interface Subsystem in Context

Conclusions

The Linux kernel is one layer in the architecture of the entire Linux system. The kernel is conceptually composed of five major subsystems: the process scheduler, the memory manager, the virtual file system, the network interface, and the inter-process communication interface. These subsystems interact with each other using function calls and shared data structures, the kernel is composed of subsystems that maintain internal representation consistency by using a specific procedural interface. As each of the subsystems is elaborated, we see an architectural style that is similar to the *layered* style. Each of the subsystems is composed of modules that communicate only with adjacent layers. The conceptual architecture of the Linux kernel has proved its success; essential factors for this success were the provision for the organization of developers, and the provision for system extensibility. The Linux kernel architecture was required to support a large number of independent volunteer developers. This requirement suggested that the system portions that require the most development, the hardware device drivers and the file and network protocols, be implemented in an extensible fashion. The Linux architect chose to make these systems be extensible using a data abstraction technique: each hardware device driver is implemented as a separate module that supports a common interface. In this way, a single developer can add a new device driver, with minimal interaction required with other developers of the Linux kernel. The success of the kernel implementation by a large number of volunteer developers proves the correctness of this strategy. Another important extension to the Linux kernel is the addition of more supported hardware platforms. The architecture of the system supports this extensibility by separating all hardware-specific code into distinct modules within each subsystem. In this way, a small group of developers can effect a port of the Linux kernel to a new hardware architecture by re-implementing only the machine-specific portions of the kernel.

References:

1. David Garlan and Mary Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company.
2. Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan, Architectural Styles, Design Patterns, and Objects, IEEE Software, January 1997
3. *Slackware Linux Unleashed*, by Timothy Parker, et al, Sams Publishing, 201 West 103rd Street, Indianapolis.
4. Dewayne E. Perry and Alexander L. Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17:4, October 1992 pp.
5. *The New Hackers Dictionary*, Second Edition, compiled by Eric S. Raymond. The MIT Press, Cambridge Massachusetts, 1993.
6. *The Linux Kernel*, by David A. Rusling, draft, version 0.1-13(19), ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linuxkernel/ http://www.linuxhq.com/guides/TLK/index.html.
7. Soni, D.; Nord, R. L.; Hofmeister, C., Software Architecture in Industrial Applications, IEEE ICSE 1995, pp. 196-210.
8. *Modern Operating Systems*, by Andrew S. Tanenbaum, Prentice Hall, 1992.
9. *Linux System Administrators' Guide 0.6*, by Lars Wirzenius,
10. http://www.linuxhq.com/LDP/LDP/sag/index.html.