OPEN SOURCE SUMMIT
China 2019

Anatomy Of The Linux Kernel

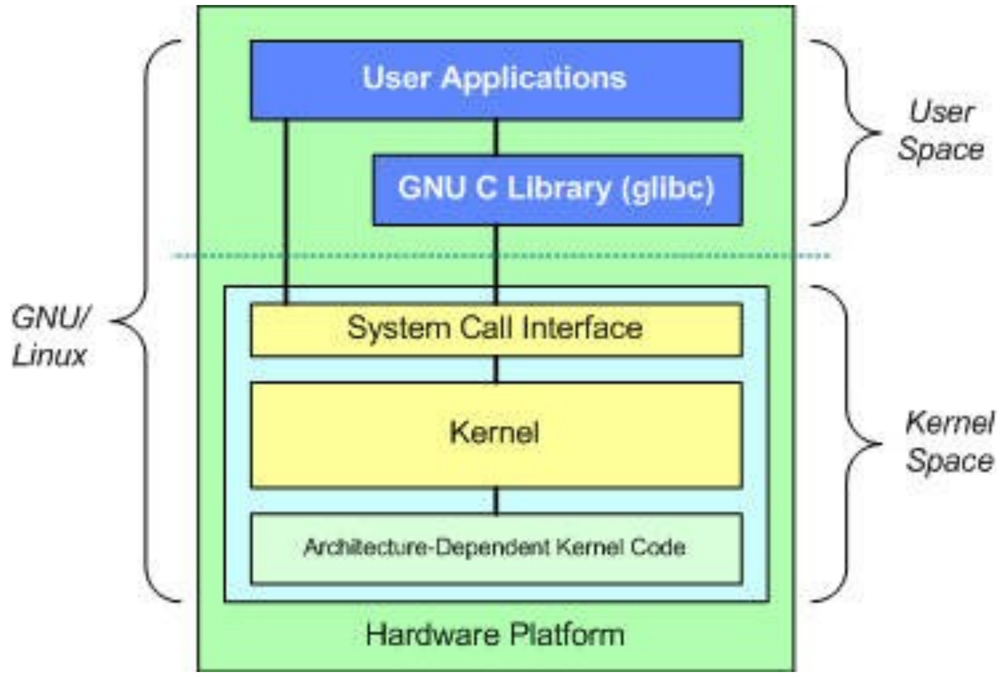
Tanish Shinde

Fundamental Architecture OfLinux

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 (G-libc). 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.

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.



OPEN SOURCE SUMMIT

China 2019

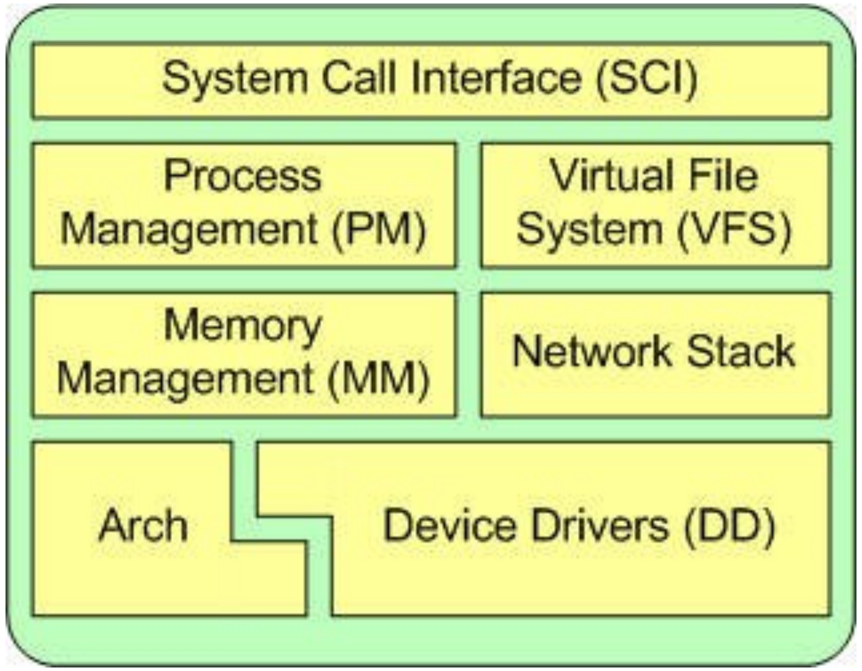
Structural Decomposition Of Linux System Call Interface, Process Management & Network Stack

System Call Interface & Process Management

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.

Process management is focused on the execution of processes. In the kernel, these are called *threads* and represent an individual virtualisation 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 synchronise between them (signal, or POSIX mechanisms).

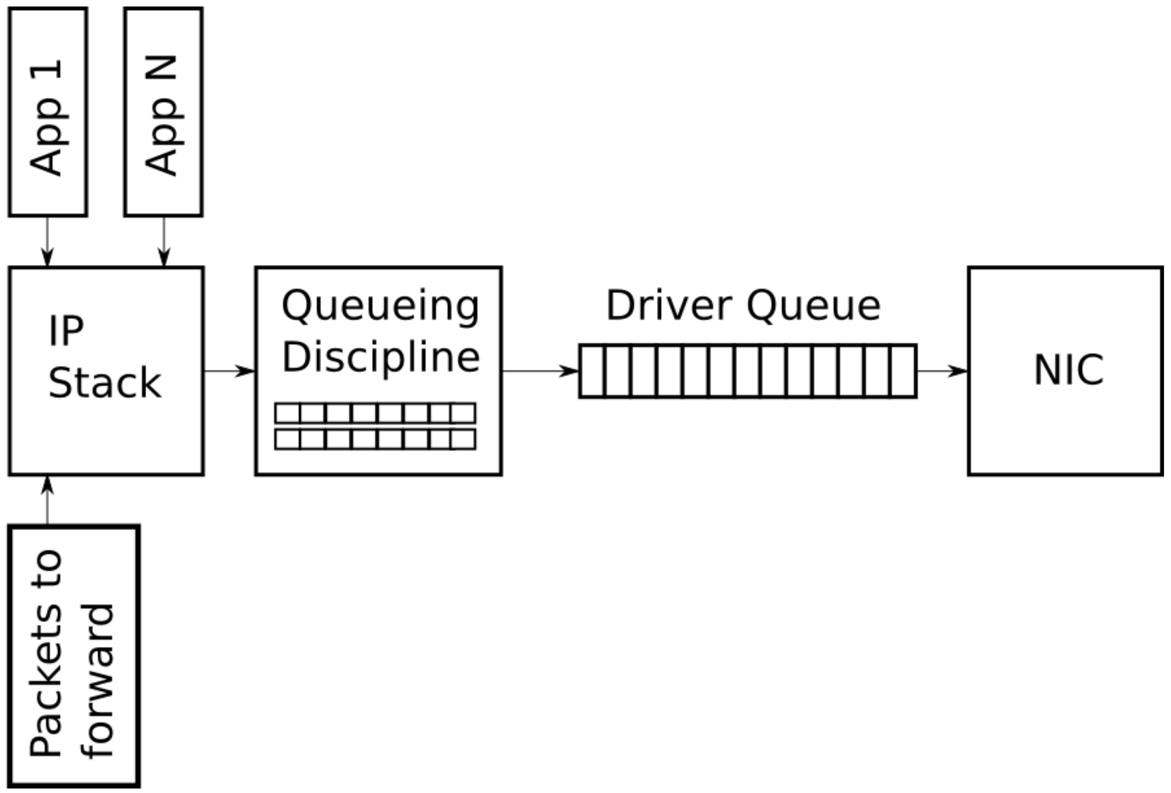
Also in process management is the need to share the CPU between the active threads. The kernel implements a novel scheduling algorithm that operates in constant time, regardless of the number of threads vying for the CPU. This is called the O(1) scheduler, denoting that the same amount of time is taken 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).



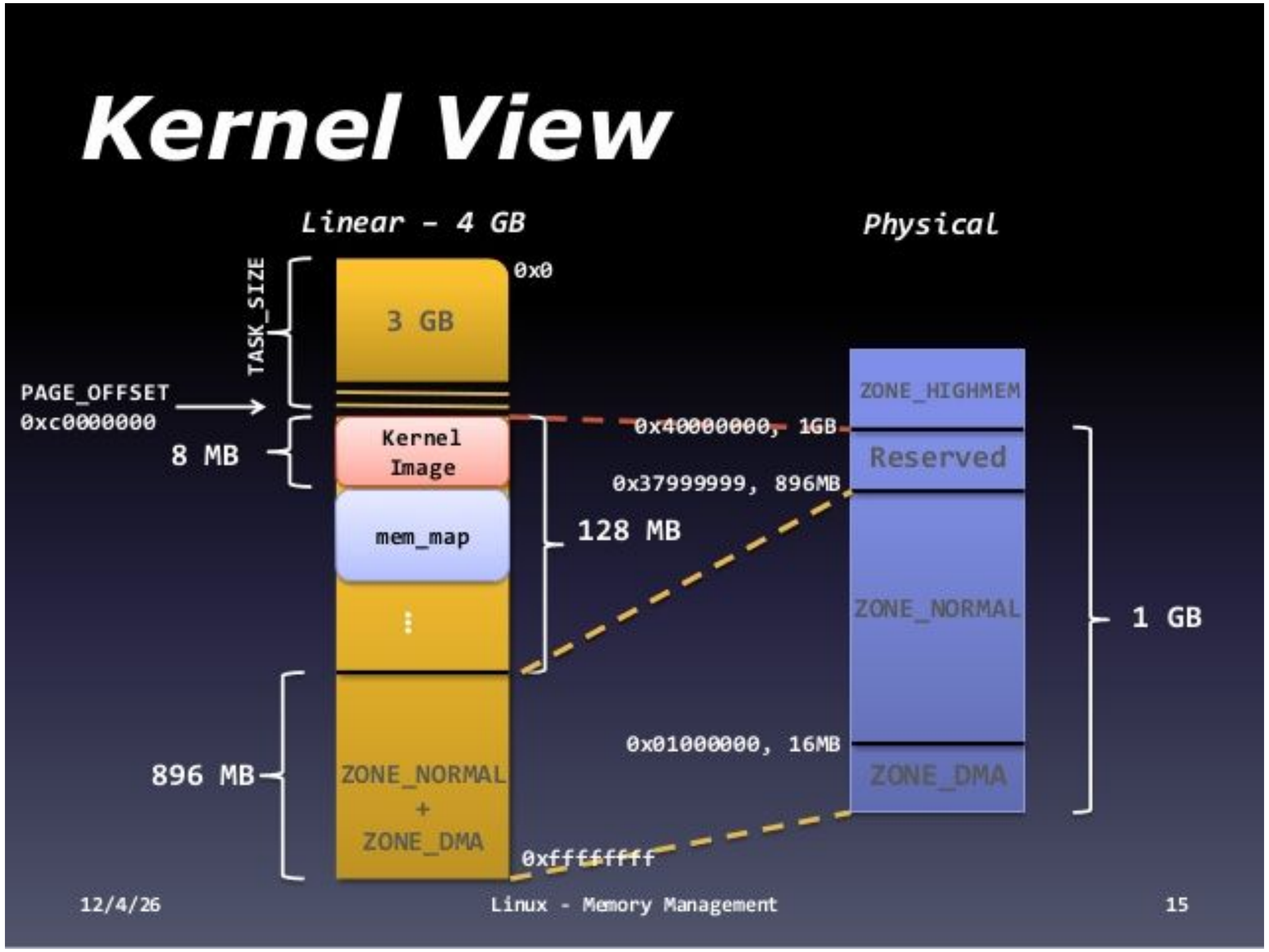
Network Stack or Protocol Stack

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 standardised way to manage connections and move data between endpoints. You can find the networking sources in the kernel at ./linux/ net.

The protocol stack or network stack is an implementation of a computer networking protocol suite or protocol family . Some of these terms are used interchangeably but strictly speaking, the *suite* is the definition of the communication protocols , and the *stack* is the software implementation of them. Individual protocols within a suite are often designed with a single purpose in mind. This modularisation simplifies design and evaluation. Because each protocol module usually communicates with two others, they are commonly imagined as layers in a stack of protocols. The lowest protocol always deals with low-level interaction with the communications hardware. Each higher layer adds additional capabilities. User applications usually deal only with the topmost layers.

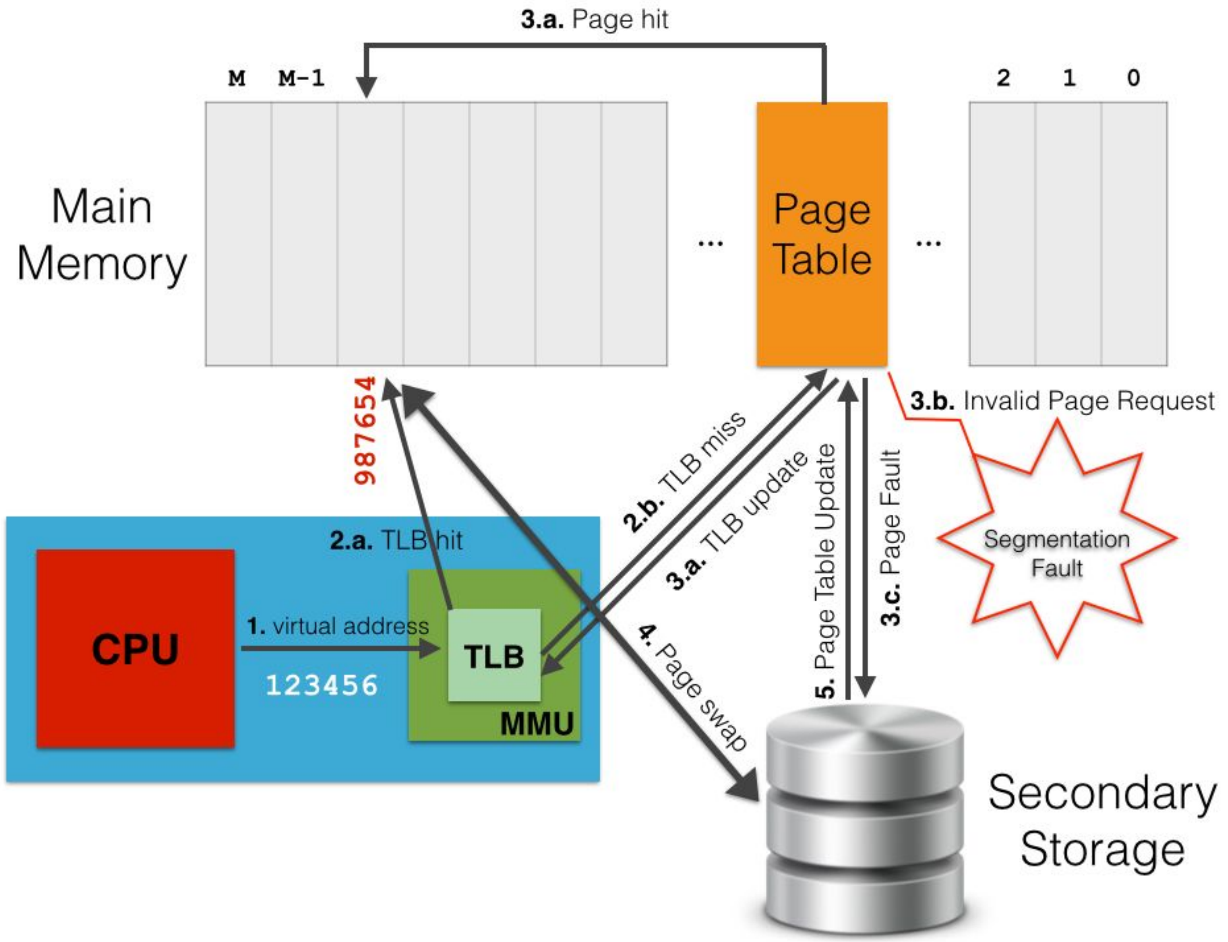


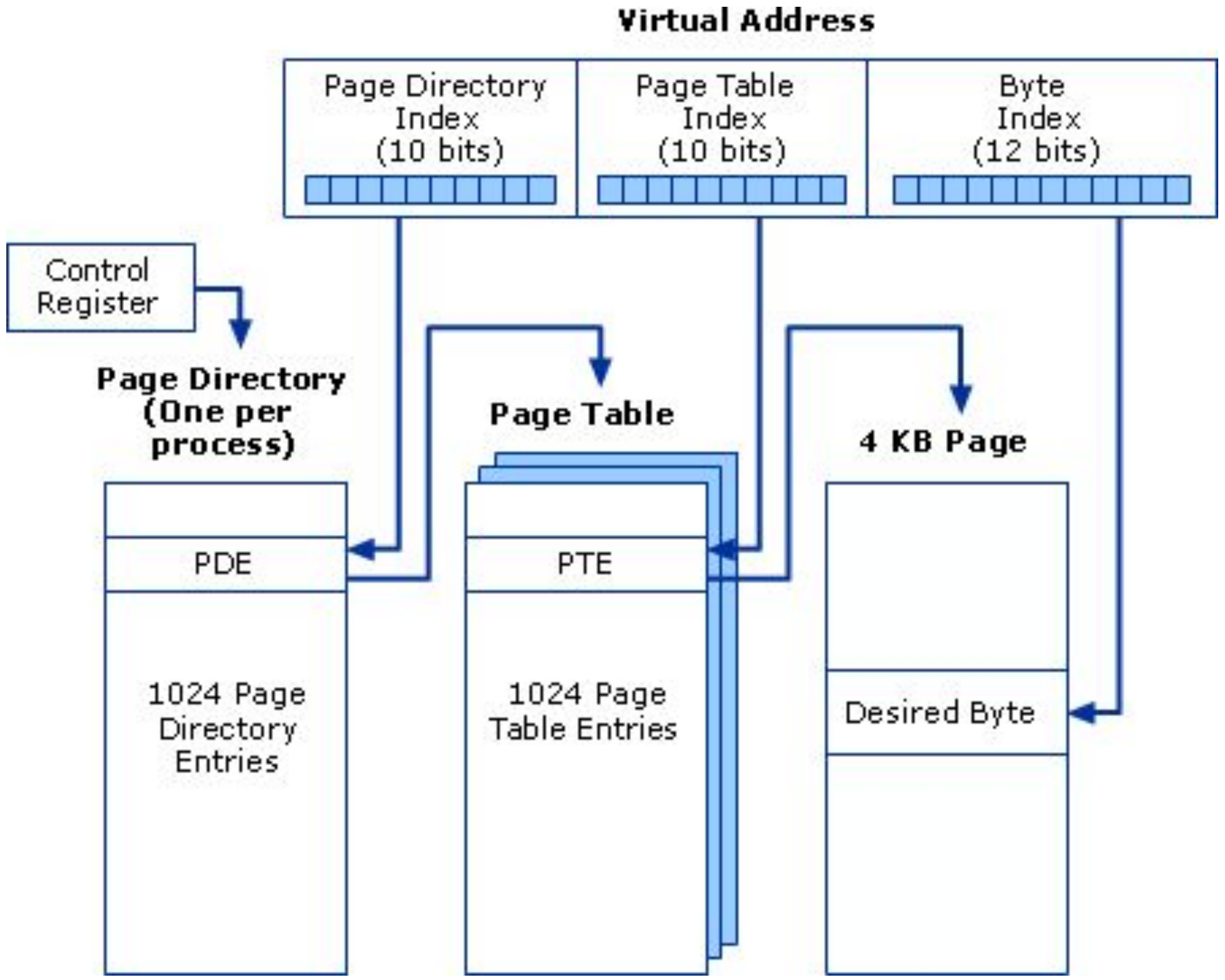
Structural Decomposition Of Linux Memory Management

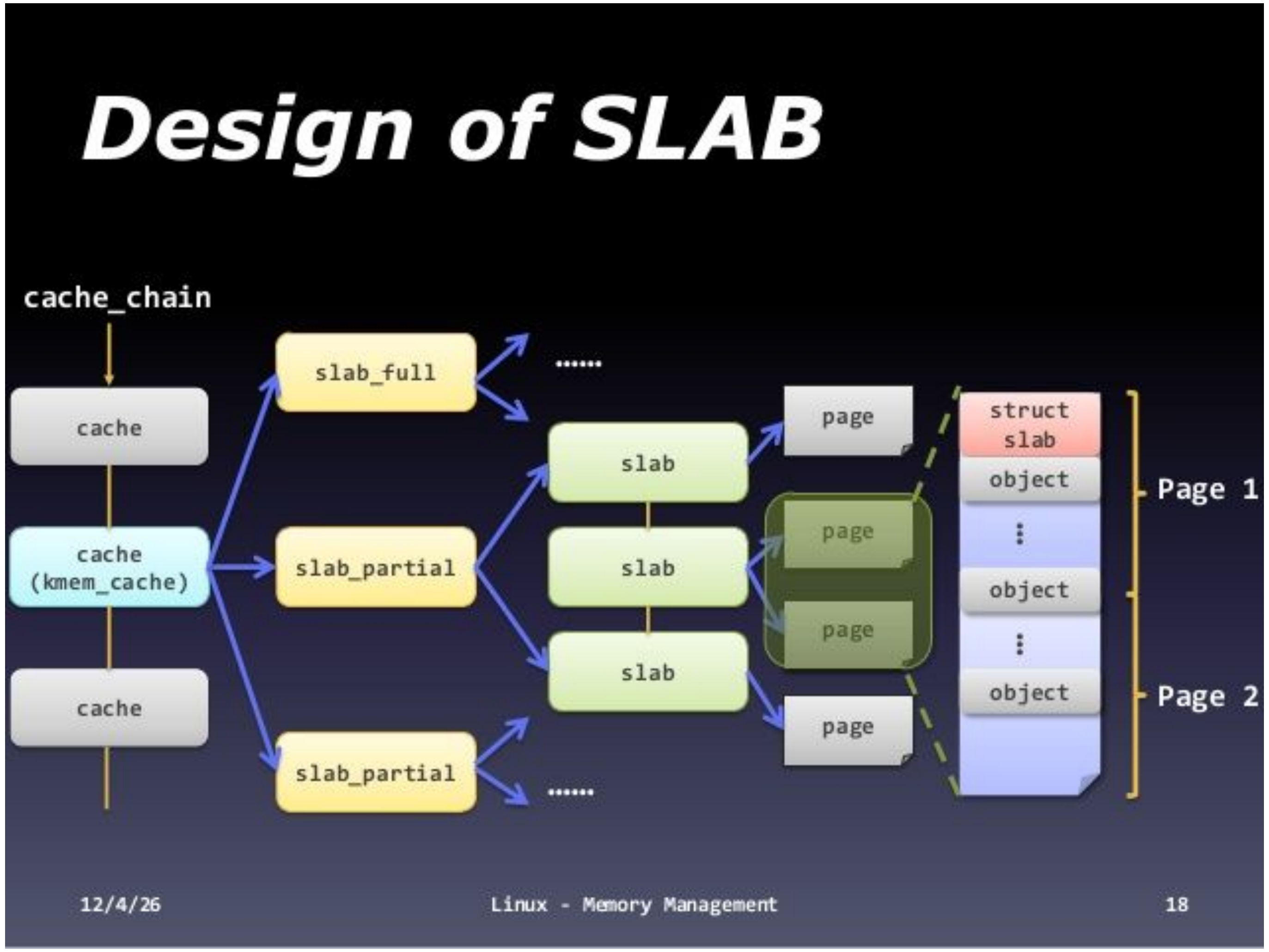


Kernel View

12/4/26 Linux - Memory Management 15


3.a. Page hit
M
M-1
2
1
0
Main Memory
...
Page Table
...
987654
2.b. TLB miss
3.a. TLB update
5. Page Table Update
3.c. Page Fault
3.b. Invalid Page Request
Segmentation Fault
2.a. TLB hit
CPU
1. virtual address
123456
TLB
MMU
4. Page swap
Secondary Storage


Virtual Address
Page Directory Index (10 bits)
Page Table Index (10 bits)
Byte Index (12 bits)
Control Register
Page Directory (One per process)
PDE
1024 Page Directory Entries
Page Table
PTE
1024 Page Table Entries
4 KB Page
Desired Byte



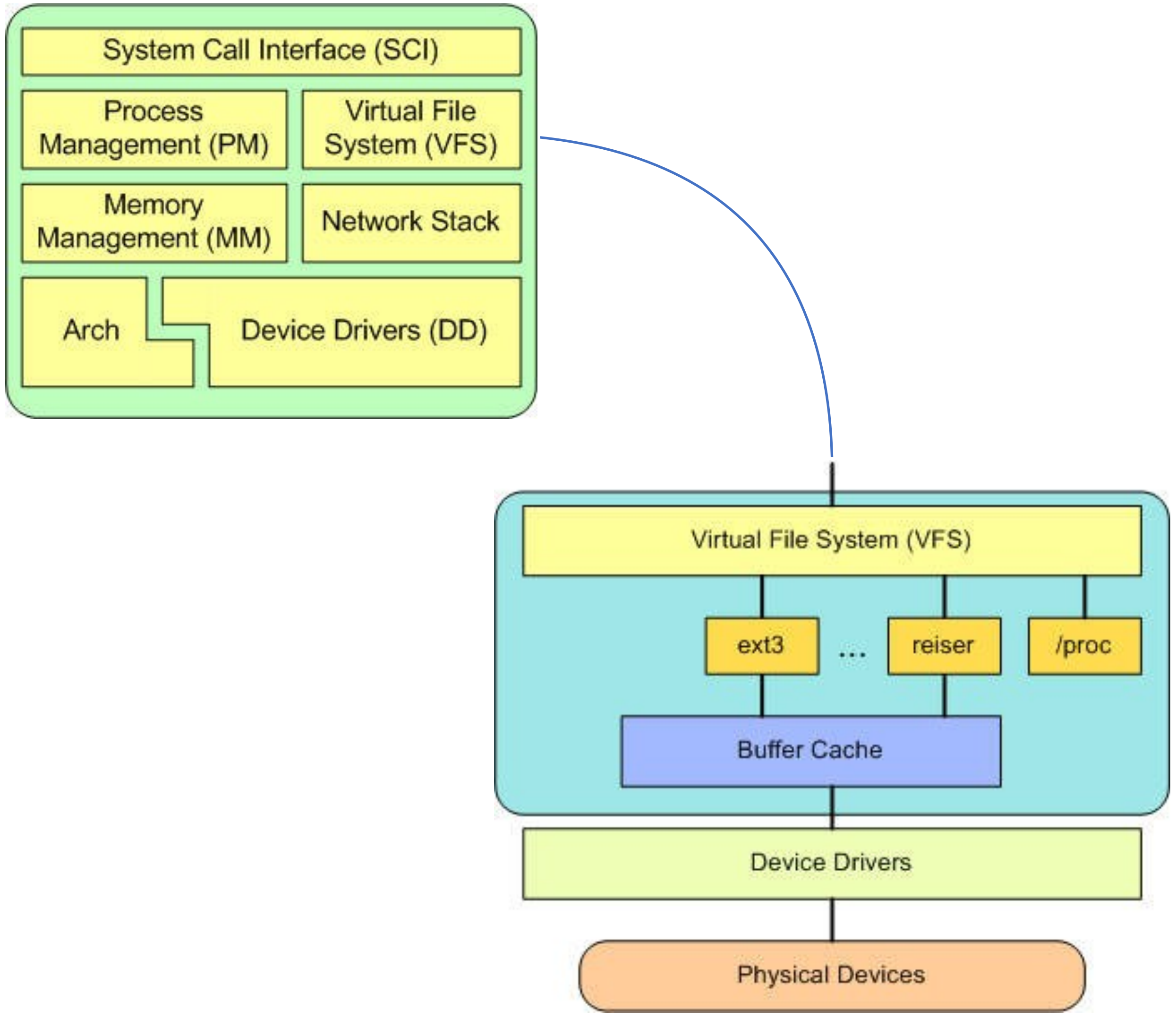
Design of SLAB

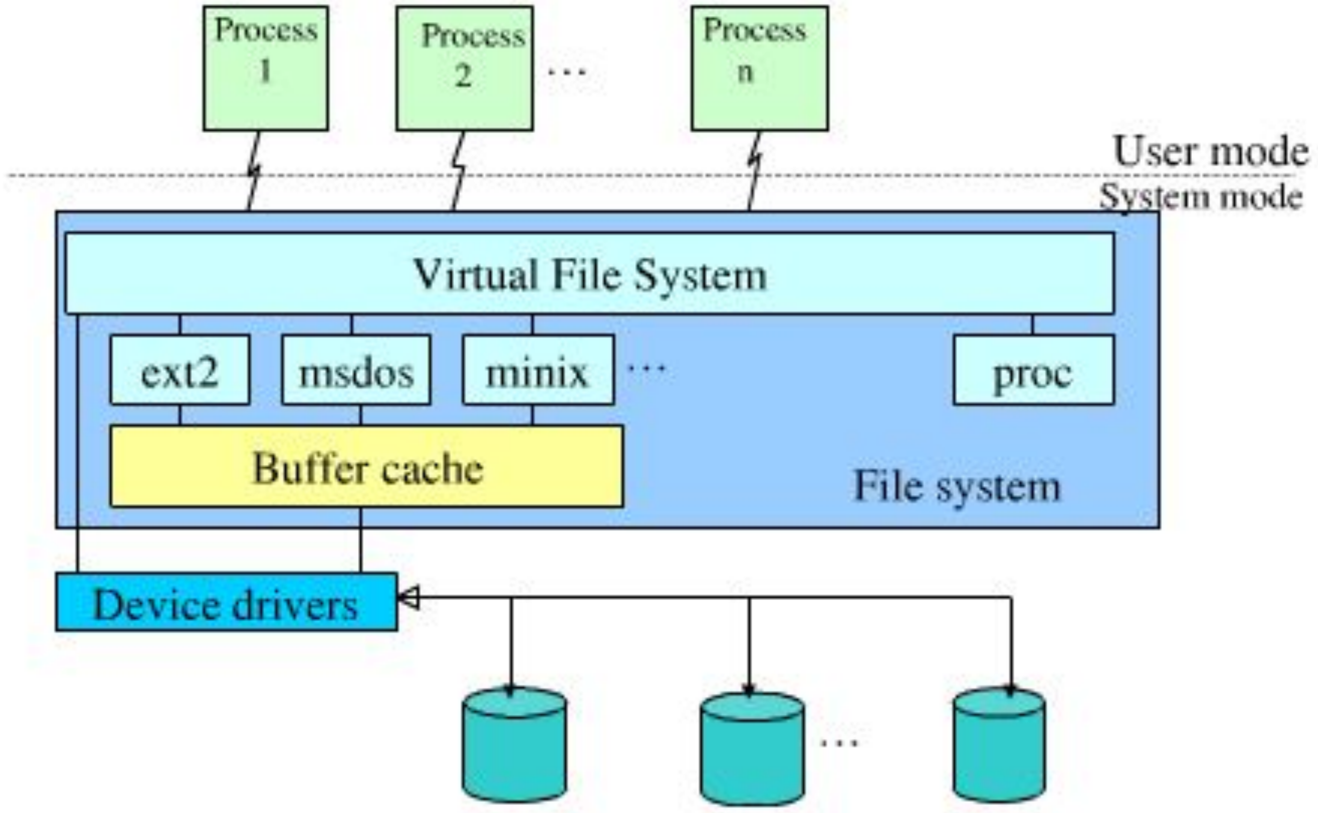
OPEN SOURCE SUMMIT
China 2019

Structural Decomposition Of Linux FileSystem

The Virtual File System

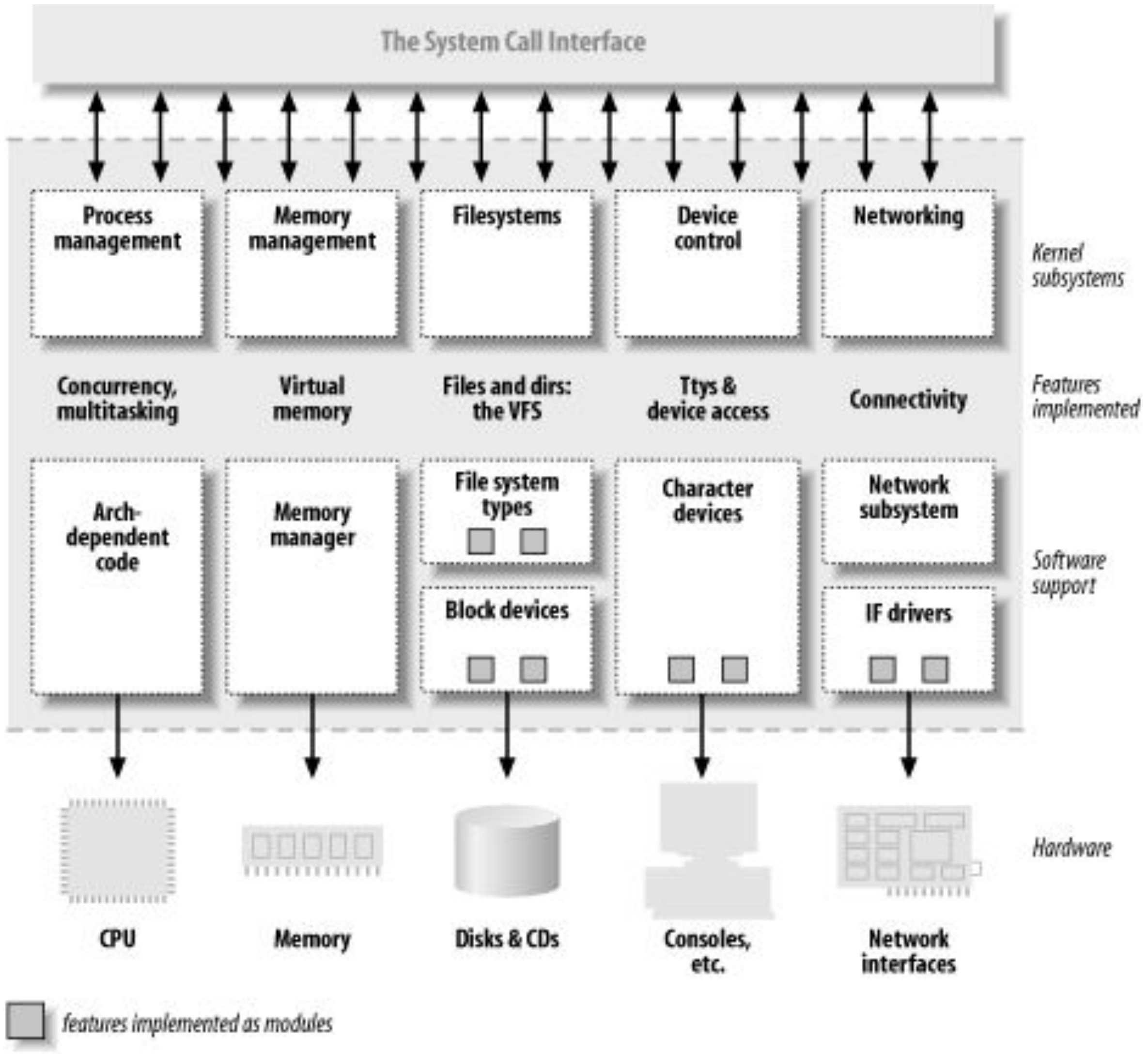
The virtual file system (VFS) is an interesting aspect of the Linux kernel because it provides a common interface abstraction for file systems. The VFS provides a switching layer between the SCI and the file systems supported by the kernel.
At the top of the VFS is a common API abstraction of functions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




Process 1
Process 2
...
Process n
User mode
System mode
Virtual File System
ext2
msdos
minix
...
proc
Buffer cache
File system
Device drivers
...

OPEN SOURCE SUMMIT
China 2019

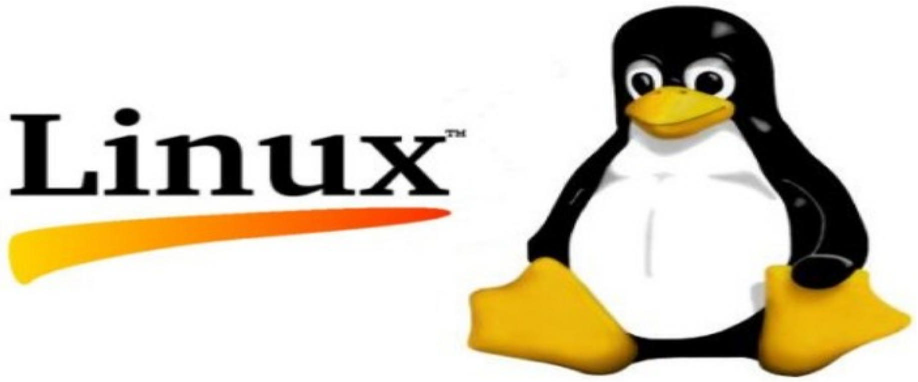
Structural Decomposition Of Linux Device Drivers


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Understanding Architecture Dependent Code

Architecture Dependent Code

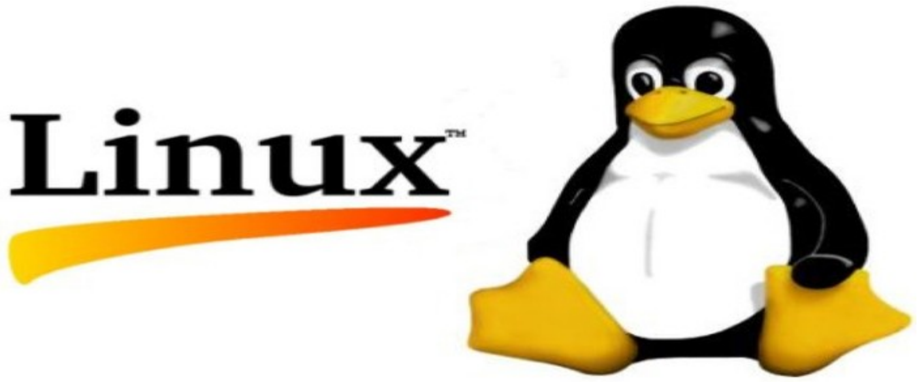
While much of Linux is independent of the architecture on which it runs, there are elements that must consider the architecture for normal operation and for efficiency. The ./linux/arch subdirectory defines the architecture-dependent portion of the kernel source contained in a number of subdirectories 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.



Architecture Dependent Code

1. *The architecture-dependent code*; it is under the architecture-independent code, forms what is usually referred to as a Board Support Package or BSP – this contains a small program called the bootloader that places the Operating System and device drivers into memory.

The architectural perspective of the Linux kernel consists of: System call interface, Process Management, the Virtual File system, Memory Management, Network Stack, Architecture and the Device Drivers.



Questions

OPEN SOURCE SUMMIT

China 2019