Linux概念架构的理解

摘要

Linux kernel成功的两个原因：（1）架构设计支持大量的志愿开发者加入到开发过程中；（2）每个子系统，尤其是那些需要改进的，都支持很好的扩展性。正式这两个原因使得Linux kernel可以不断进化。

一、Linux内核在整个计算机系统中的位置

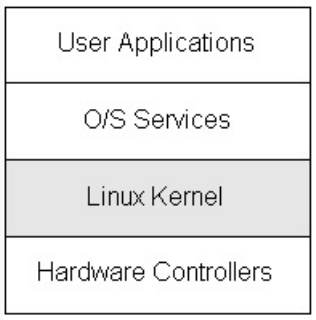


Fig 1 - 计算机系统分层结构

分层结构的原则：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.

这种子系统之间的依赖性只能是从上到下，也就是图中top的子系统依赖bottom的子系统，反之则不行。

二、内核的作用

1.
2. **虚拟化(抽象)**，将计算机硬件抽象为一台虚拟机，供用户进程(process)使用；进程运行时完全不需要知道硬件是如何工作的，只要调用Linux kernel提供的虚拟接口(virtual interface)即可。
3.
4. **多任务处理**，实际上是多个任务在并行使用计算机硬件资源，内核的任务是仲裁对资源的使用，制造每个进程都以为自己是独占系统的错觉。
5.

PS：进程上下文切换就是要换掉程序状态字、换掉页表基地址寄存器的内容、换掉current指向的task_struct实例、换掉PC——>也就换掉了进程打开的文件(通过task_struct的files可以找到)、换掉了进程内存的执行空间(通过task_struct的mem可以找到)；

三、Linux内核的整体架构

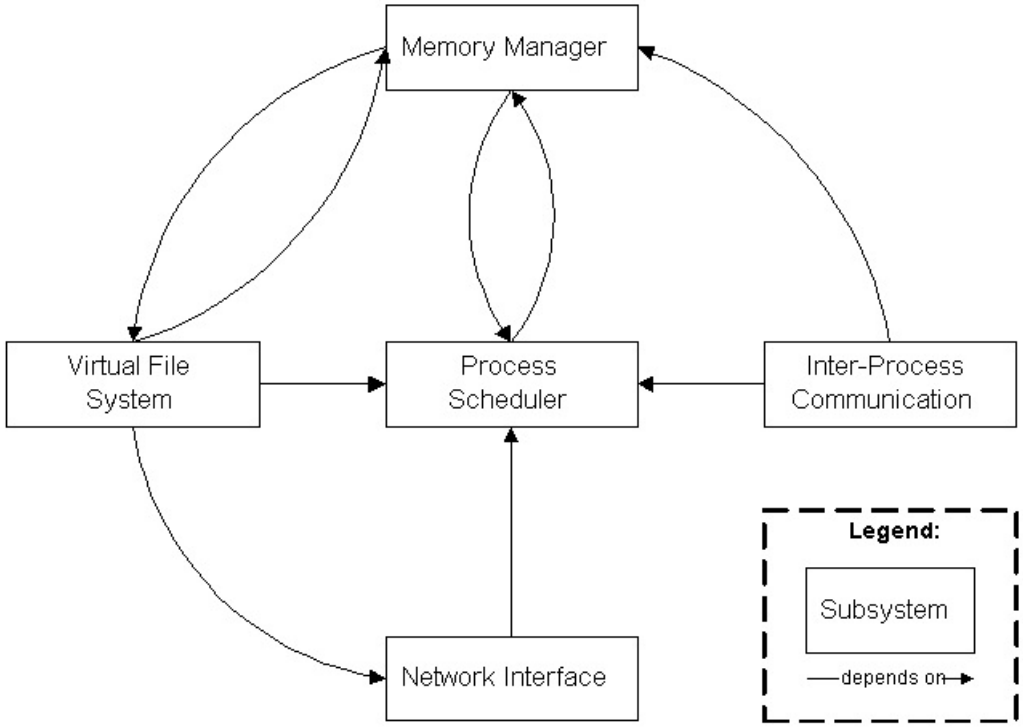


Figure 2.2: *Kernel Subsystem Overview*

Linux内核的整体架构

中心系统是Process Scheduler（SCHED）：所有其余的子系统都依赖于Process Scheduler，因为其余子系统都需要阻塞和恢复进程。当一个进程需要等待一个硬件动作完成时，相应子系统会阻塞这个进程；当这个硬件动作完成时，子系统会将这个进程恢复：这个阻塞和恢复动作都要依赖于Processor Scheduler完成。

上图中的每一个依赖箭头都有原因：

-
- Process Scheduler依赖Memory manager：进程恢复执行时，需要依靠Memory Manager分配供它运行的内存。
-
- IPC子系统依赖于Memory manager：共享内存机制是进程间通信的一种方法，运行两个进程利用同一块共享的内存空间进行信息传递。
-
- VFS依赖于Network Interface：支持NFS网络文件系统；
-
- VFS依赖于Memory Manager：支持ramdisk 设备
-
- memory manager依赖于VFS，因为要支持swapping，可以将暂时不运行的进程换出到磁盘上的swap分区，进入挂起状态。
-

四、高度模块化设计的系统，利于分工合作。

1.
2. 只有极少数的程序员需要横跨多个模块开展工作，这种情况确实会发生，仅发生在当前系统需要依赖另一个子系统时；
3.
4. 硬件设备驱动（hardware device drivers）、文件系统模块（logical filesystem modules）、网络设备驱动（network device drivers）和网络协议模块（network protocol modules）这四个模块的可扩展性最高。
5.

五、系统中的数据结构

1.
2. Task List

   Process Scheduler 针对每个进程维护一个数据结构**task_struct**；所有的进程用链表管理，形成**task list**；process scheduler还维护一个**current指针**指向当前正在占用CPU的进程。

3.
4. Memory Map

   Memory Manager存储每个进程的虚拟地址到物理地址的映射；并且也提供了如何换出特定的页，或者是如何进行缺页处理。这些信息存放在数据结构**mm_struct**中。每个进程都有一个mm_struct结构，在进程的task_struct结构中有一个指针mm指向次进程的mm_struct结构。

   在mm_struct中有一个指针pgd，指向该进程的页目录表（即存放页目录首地址）——>当该进程被调度时，此指针被换成物理地址，写入控制寄存器CR3(x86体系结构下的页基址寄存器)
5.
6. I-nodes

   VFS通过inodes节点表示磁盘上的文件镜像，inodes用于记录文件的物理属性。每个进程都有一个**files_struct**结构，用于表示该进程打开的文件，在task_struct中有个files指针。使用inodes节点可以实现文件共享。文件共享有两种方式：（1）通过同一个系统打开文件**file**指向同一个inodes节点，这种情况发生于父子进程间；（2）通过不同系统打开文件指向同一个inode节点，举例有硬链接；或者是两个不相关的指针打开同一个文件。
7.
8. Data Connection

   内核中所有的数据结构的根都在Process Scheduler维护的task list链表中。系统中每个进程的的数据结构task_struct中有一个指针mm指向它的内存映射信息；也有一个指针files指向它打开的文件（用户打开文件表）；还有一个指针指向该进程打开的网络套接字。
9.

六、子系统架构

1. Process Scheduler 架构

（1）目标

process scheduler是Linux kernel中最重要的子系统。系统通过它来控制对CPU的访问——不仅仅是用户进程对CPU的访问，也包括其余子系统对CPU的访问。

（2）模块

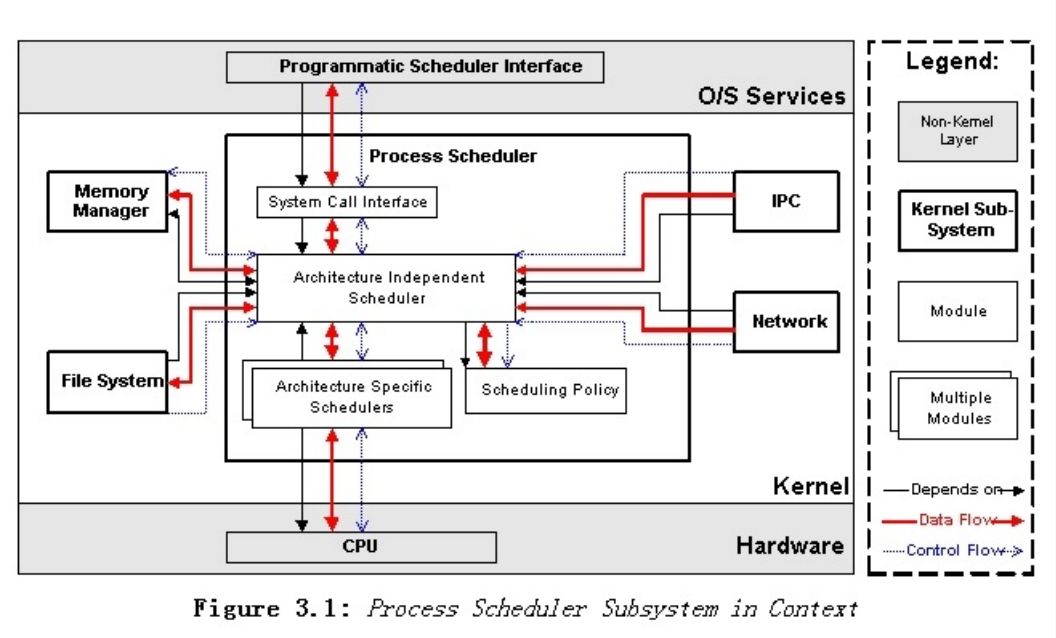


Figure 3.1: *Process Scheduler Subsystem in Context*

进程调度器

调度策略模块(scheduling policy module)：决定哪个进程获得对CPU的访问权；调度策略应该让所有进程尽可能公平得共享CPU。

-
- 体系结构相关模块(architecture-specific module)设计一组统一的抽象接口来屏蔽特定体系接口芯片的硬件细节。这个模块与CPU交互以阻塞和恢复进程。这些操作包括获取每个进程需要保存的寄存器和状态信息、执行汇编代码来完成阻塞或者恢复操

作。

- 体系结构无关模块(architecture-independent module) 与调度策略模块交互将决定下一个执行的进程，然后调用体系结构相关的代码去恢复那个进程的执行。不仅如此，这个模块还会调用memory manager的接口来确保被阻塞的进程的内存映射信息被正确得保存起来。
- 系统调用接口模块(system call interface) 允许用户进程访问Linux Kernel明确暴露给用户进程的资源。通过一组定义合适的基本上不变的接口（POSIX标准），将用户应用程序和Linux内核解耦，使得用户进程不会受到内核变化的影响。

（3）. 数据表示

调度器维护一个数据结构——task list，其中的元素时每个活动的进程task_struct实例；这个数据结构不仅仅包含用来阻塞和恢复进程的信息，也包含额外的计数和状态信息。这个数据结构在整个kernel层都可以公共访问。

（4）. 依赖关系、数据流、控制流

正如前面提到过的，调度器需要调用memory manager提供的功能，去为需要恢复执行的进程选择合适的物理地址，正因为如此，所以Process Scheuler子系统依赖于内存管理子系统。当其他内核子系统需要等待硬件请求完成时，它们都依赖于进程调度子系统进行进程的阻塞和恢复。这种依赖性通过函数调用和访问共享的task list数据结构来体现。所有的内核子系统都要读或者写代表当前正在运行进程的数据结构，因此形成了贯穿整个系统的双向数据流。

除了内核层的数据流和控制流，OS服务层还给用户进程提供注册定时器的接口。这形成了由调度器对用户进程的控制流。通常唤醒睡眠进程的用例不在正常的控制流范围，因为用户进程无法预知何时被唤醒。最后，调度器与CPU交互来阻塞和恢复进程，这又形成它们之间的数据流和控制流——CPU负责打断当前正在运行的进程，并允许内核调度其他的进程运行。

2. Memory Manager 架构

（1）目标

内存管理模块负责控制进程如何访问物理内存资源。通过硬件内存管理系统（MMU）管理进程虚拟内存和机器物理内存之间的映射。每一个进程都有自己独立的虚拟内存空间，所以两个进程可能有相同的虚拟地址，但是它们实际上在不同的物理内存区域运行。MMU提供内存保护，让两个进程的物理内存空间不互相干扰。内存管理模块还支持swap——将暂时不用的内存页换出到磁盘上的swap分区，这种技术让进程的虚拟地址空间大于物理内存的大小。虚拟地址空间的大小由机器字长决定。

（2）模块

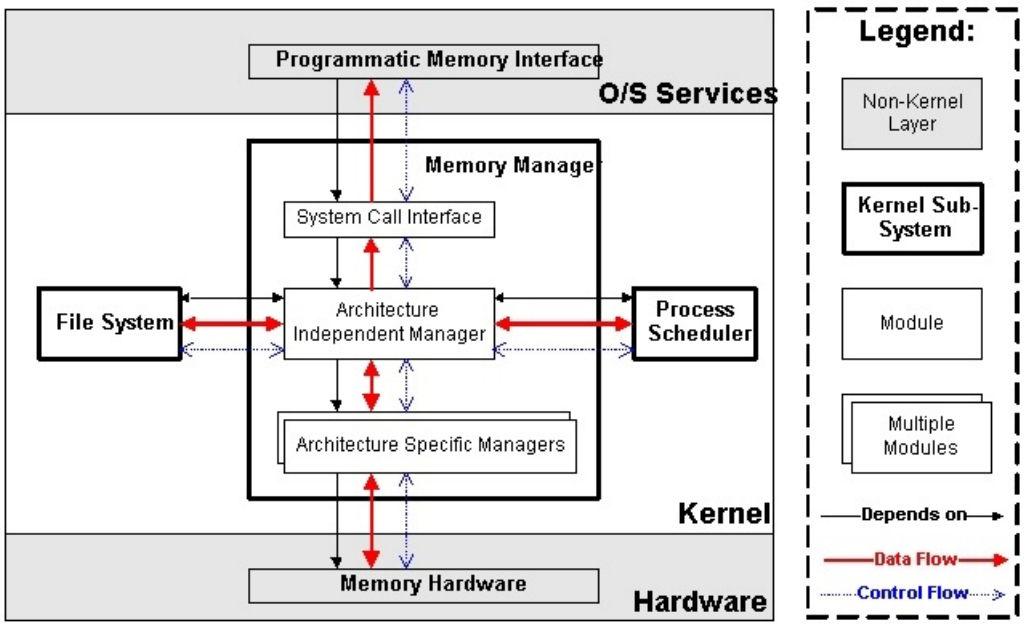


Figure 3.2: *Memory Manager subsystem in context*

内存管理子系统

-
-

  架构相关模块(architecture specific module)提供访问物理内存的虚拟接口；

-

- 架构无关模块(architecture independent module)负责每个进程的地址映射以及虚拟内存交换。当发生缺页错误时，由该模块负责决定哪个内存页应该被换出内存——因为这个内存页换出选择算法几乎不需要改动，所以这里没有建立一个独立的策略模块。

-
- 系统调用接口(system call interface) 为用户进程提供严格的访问接口（malloc和free；mmap和ummap）。这个模块允许用进程分配和释放内存、执行内存映射文件操作。

-

（3）数据表示

内存管理存放每个进程的虚拟内存到物理内存的映射信息。这种映射信息存放在mm_struct结构实例中，这个实例的指针又存放在每个进程的task_struct中。除了存放映射信息，数据块中还应该存放关于内存管理器如何获取和存储页的信息。例如：可执行代码能够将可执行镜像作为备份存储；但是动态申请的数据则必须备份到系统页中。（这个没看懂，请高手解惑？）
最后，内存管理模块还应该存放访问和技术信息，以保证系统的安全。

（4）依赖关系、数据流和控制流

内存管理器控制物理内存，当page fault发生时，接受硬件的通知（缺页中断）—— 这意味着在内存管理模块和内存管理硬件之间存在双向的数据流和控制流。内存管理也依赖文件系统来支持swapping和内存映射I/O——这种需求意味着内存管理器需要调用对文件系统提供的函数接口(procedure calls)，往磁盘中存放内存页和从磁盘中取内存页。因为文件系统请求非常慢，所以在等待内存页被换入之前，内存管理器要让进程需要进入休眠——这种需求让内存管理器调用process scheduler的接口。由于每个进程的内存映射存放在进程调度器的数据结构中，所以在内存管理器和进程调度器之间也有双向的数据流和控制流。用户进程可以建立新的进程地址空间，并且能够感知缺页错误——这里需要来自内存管理器的控制流。一般来说没有用户进程到内存管理器的数据流，但是用户进程却可以通过select系统调用，从内存管理器获取一些信息。

3. Virtual File System 架构

（1）目标

虚拟文件系统为存储在硬件设备上数据提供统一的访问接口。可以兼容不同的文件系统（ext2,ext4,ntf等等）。计算机中几乎所有的硬件设备都被表示为一个通用的设备驱动接口。逻辑文件系统促进与其他操作系统标准的兼容性，并且允许开发者以不同的策略实现文件系统。虚拟文件系统更进一步，允许系统管理员在任何设备上挂载任何逻辑文件系统。虚拟文件系统封装物理设备和逻辑文件系统的细节，并且允许用户进程使用统一的接口访问文件。

除了传统的文件系统目标，VFS也负责装载新的可执行文件。这个任务由逻辑文件系统模块完成，使得Linux可以支持多种可执行文件。

（2）模块

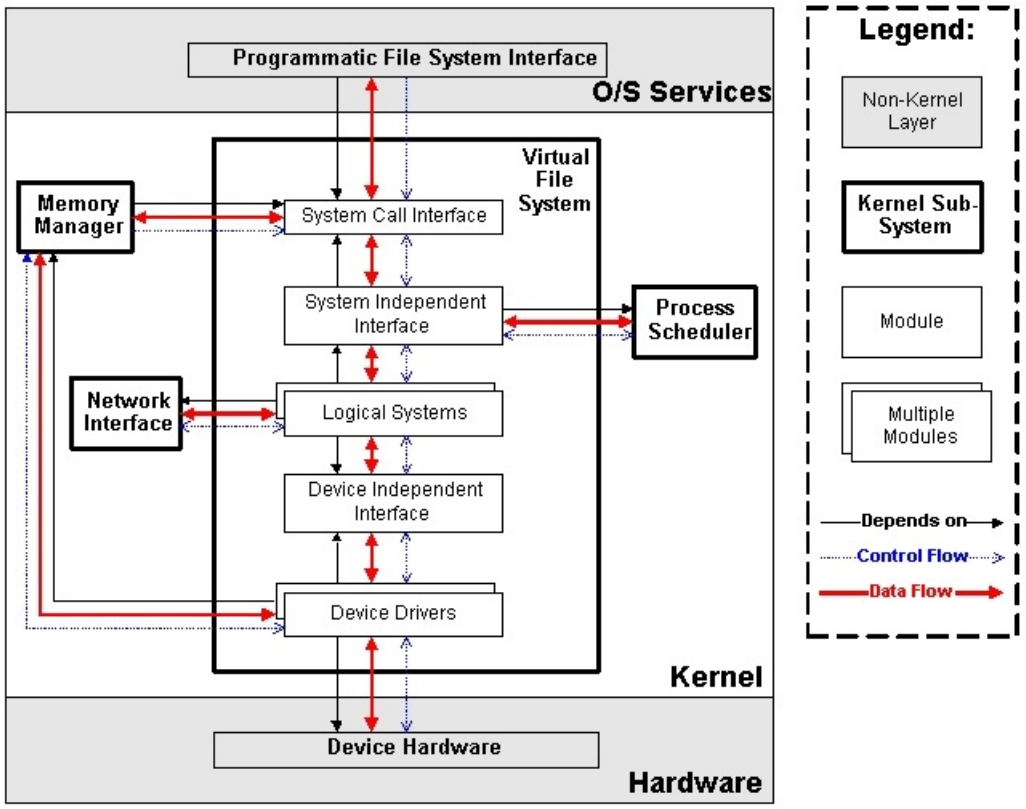


Figure 3.3: *Virtual File System in Context*

虚拟文件系统模块

-
- 设备驱动模块（device driver module）
-
- 设备独立接口模块（Device Independent Interface）：提供所有设备的同一视图
-
- 逻辑文件系统（logical file system）：针对每种支持的文件系统
-
- 系统独立接口（system independent interface）提供硬件资源和逻辑文件系统都无关的接口，这个模块通过块设备节点或者字符设备节点提供所有的资源。
-
- 系统调用模块（system call interface）提供用户进程对文件系统的统一控制访问。虚拟文件系统为用户进程屏蔽了所有特殊的特性。
-

（3）数据表示

所有文件使用i-nodes表示。每个inode都记录一个文件在硬件设备上的位置信息。不仅如此，inode还存放着指向逻辑文件系统模块和设备驱动的的函数指针，这些指针能够执行具体的读写操作。通过按照这种形式（就是面向对象中的虚函数的思想）存放函数指针，具体的逻辑文件系统和设备驱动可以向内核注册自己而不需要内核依赖具体的模块特性。

（4）依赖关系、数据流和控制流

一个特殊的设备驱动是ramdisk，这个设备在主存中开辟一片区域，并把它当成持久性存储设备使用。这个设备驱动使用内存管理模块完成任务，所以在VFS与对内存管理模块存在依赖关系（图中的依赖关系反了，应该是VFS依赖于内存管理模块）、数据流和控制流。

逻辑文件系统支持网络文件系统。这个文件系统像访问本地文件一样，从另一台机器上访问文件。为了实现这个功能，一种逻辑文件系统通过网络子系统完成它的任务——这引入了VFS对网络子系统的一个依赖关系以及它们之间的控制流和数据流。

正如前面提到的，内存管理器使用VFS完成内存swap功能和内存映射I/O。另外，当VFS等待硬件请求完成时，VFS需要使用进程调度器阻塞进程；当请求完成时，VFS需要通过进程调度器唤醒进程。最后，系统调用接口允许用户进程调用来存取数据。不像前面的子系统，VFS没有提供给用户注册不明确调用的机制，所以没有从VFS到用户进程的控制流。

4. Network Interface 架构

（1）目标

网络子系统让Linux系统能够通过网络与其他系统相连。这个子系统支持很多硬件设备，也支持很多网络协议。网络子系统将硬件和

协议的实现细节都屏蔽掉，并抽象出简单易用的接口供用户进程和其他子系统使用——用户进程和其余子系统不需要知道硬件设备和协议的细节。

（2）模块

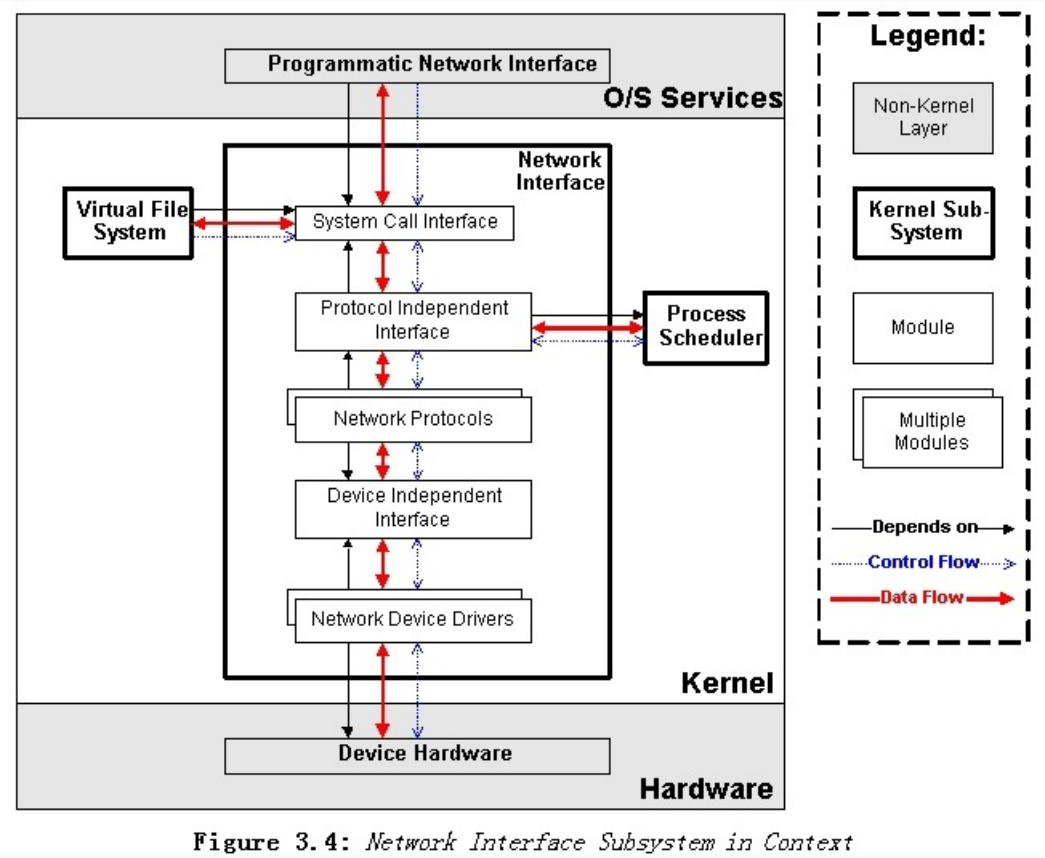


Figure 3.4: *Network Interface Subsystem in Context*

网络协议层模块图

-
- 网络设备驱动模块（network device drivers）
-
- 设备独立接口模块（device independent interface module）提供所有硬件设备的一致访问接口，使得高层子系统不需要知道硬件的细节信息。
-
- 网络协议模块（network protocol modules）负责实现每一个网络传输协议，例如：TCP，UDP，IP，HTTP，ARP等等~
-
- 协议无关模块（protocol independent interface）提供独立于具体协议和具体硬件设备的一致性接口。这使得其余内核子系统无需依赖特定的协议或者设备就能访问网络。
-
- 系统调用接口模块（system calls interface）规定了用户进程可以访问的网络编程API
-

（3）数据表示

每个网络对象都被表示为一个套接字（socket）。套接字与进程关联的方法和i-nodes节点相同。通过两个task_struct指向同一个套接字，套接字可以被多个进程共享。

（4）数据流，控制流和依赖关系

当网络子系统需要等待硬件请求完成时，它需要通过进程调度系统将进程阻塞和唤醒——这形成了网络子系统和进程调度子系统之间的控制流和数据流。不仅如此，虚拟文件系统通过网络子系统实现网络文件系统（NFS）——这形成了VFS和网络子系统指甲的数据流和控制流。

七、结论

1、Linux内核是整个Linux系统中的一层。内核从概念上由五个主要的子系统构成：进程调度器模块、内存管理模块、虚拟文件系统、网络接口模块和进程间通信模块。这些模块之间通过函数调用和共享数据结构进行数据交互。、

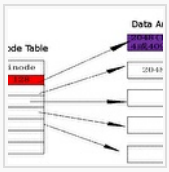
2、Linux内核架构促进了他的成功，这种架构使得大量的志愿开发人员可以合适得分工合作，并且使得各个特定的模块便于扩展。

-

- **可扩展性一**：Linux架构通过一项数据抽象技术使得这些子系统成为可扩展的——每个具体的硬件设备驱动都实现为单独的模块，该模块支持内核提供的统一的接口。通过这种方式，个人开发者只需要和其他内核开发者做最少的交互，就可以为Linux内核添加新的设备驱动。
-
- **可扩展性二**：Linux内核支持多种不同的体系结构。在每个子系统中，都将体系结构相关的代码分割出来，形成单独的模块。通过这种方法，一些厂家在推出他们自己的芯片时，他们的内核开发小组只需要重新实现内核中机器相关的代码，就可以讲内核移植到新的芯片上运行。
-

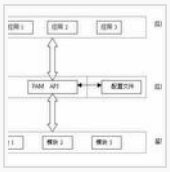
参考文章：

1.
2. http://oss.org.cn/ossdocs/linux/kernel/a1/index.html
3.
4. http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html
5.
6. http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html
7.
8. http://www.fceia.unr.edu.ar/ingsoft/monroe00.pdf
9.
10. 内核源码：http://lxr.oss.org.cn/
11.

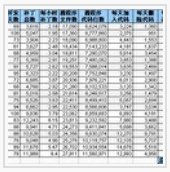


linux文件系统—inode及相关概念

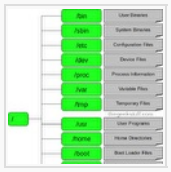
Linux下简单限制网卡的带宽



Linux可插拔认证模块的基本概念与架构



谁写了Linux



Linux目录结构（Linux文件系统结构）

更多 1
上一篇：怎样在1秒内启动Linux
下一篇：Linux性能分析的前60000毫秒

用户评论

0

我有话说...

使用社交帐号登录

或以游客身份发布

昵称 发布

最新评论

还没有评论

更多热评文章

Linux基金会将提供Linux入门的免费公开课

入门级学习：Linux学习方向和方法浅谈

听高焕堂讲架构：提高架构质量的观点

微观架构及宏观架构