# 第1章 前 言

## 1.1 目标

本部分的目标是描述Linux内核的具体系统结构。具体系统结构指的是系统创建之后的实际系统结构。我们希望开发具体的系统结构，以提供现有 Linux内核的高级文档资料。

## 1.2 Linux介绍

Linus B.Torvalds于1991年编写出第一个Linux内核。由于它一直是作为自由软件发布的，所以Linux变得很流行。因为源代码随手可得。用户可以随意改变内核，以使它适应自己的需求。然而，在编写新的系统程序之前，了解 Linux内核的发展过程以及当前它的工作原理是很重要的。

基于Linux内核源代码的具体系统结构可以为 Linux内核高手和开发人员提供一个可靠的和及时的参考。自从 1991年以来，大量的志愿者多次修改过 Linux，他们在因特网上通过Usenet新闻组相通信。过去， Torvalds是主要的内核开发人员。现在 Linux Torvalds已经不再是Linux内核工程小组的成员了。如果能提供准确的和及时的具体系统结构，我们有理由相信Linux会得到进一步修改和进一步发展。

Linux是一个Unix兼容的系统。大部分通用的 Unix工具和程序现在都可以在 Linux下运行。最初设计Linux时是为了它能在Intel 80386微处理器上执行。最初的版本因为使用了 Intel所特有的中断处理例程，所以不能移植到其他的平台上。当把 Linux移植到其他硬件平台（如 DEC Alpha和Sun SPARC）上时，大部分依赖于平台的代码被移入平台相关的模块中，这些模块支持通用接口。

Linux的用户群是巨大的。在1994年，Ed Chi估计Linux大约有40,000个用户（[Chi 1994]）。Linux文档工程小组（LDP）正在开发有用的和可靠的 Linux内核文档，既提供给 Linux用户，也提供给Linux开发人员使用。就我们所知，LDP并不利用逆推机制得到最新的具体系统结构。目前有大量的书和文档资料介绍 Linux内核方面的知识[CS746G Bibliography]。然而，还没有什么文档资料很详细地介绍 Linux的概念和具体系统结构。有些出版物（如 [Beck 1996]和[Rusling 1997]）介绍了Linux内核的工作原理，然而，这些书并没有透彻地分析子系统以及子系统之间的相互依赖性。

## 1.3 软件系统结构的背景知识

最近以来，在工业和学术团体中，对软件系统结构的研究非常流行。软件系统结构的研究带动了大型软件系统的研究。最近的研究表明软件系统结构是很重要的，因为它增强了系统支持者之间的通信。软件系统结构可以用于帮助开发人员作出一些早期的设计决定。此外，它还可以用作系统的一个可传送的抽象表示（ [Bass 1998]）。

软件系统结构与软件可维护性的研究有关。维护现有的（或者传统的）系统常常是非常

麻烦的。这些现有系统的状态既可能是设计非常好的，文档编制得非常好的；也可能是设计非常差的，文档编制得非常不理想的。在许多情况下，原来的一部分或者全部系统结构者和开发人员不会再参加现有系统的开发工作，而缺乏系统结构实践知识将大大地增加软件维护任务的复杂性和困难程度。为了对现有系统的功能进行变动、扩展、修改或者删除，就必须理解系统的实现原理。这个问题就需要研究从现有系统中抽取系统结构信息和设计信息的相关技术。从源代码抽取高级模型的过程常常称为逆推工程。

逆推工程的方法主要分为两种[Bass 1998]：

1. 技术方法：抽取方法是基于现有的产品抽取有关系统的信息。具体来说，抽取的对象包括源代码、注释、用户文档、可执行模块以及系统描述。

2. 人类知识和推理：这些方法的焦点是人怎样理解软件。一般来说，工作人员常使用下列策略：

- 自顶向下策略：从最高级别的抽象开始，依次进行各个子部分的理解。
- 自底向上策略：先理解最低级别的部件，并理解这些部件是怎样在一起工作来完成系统的目标的。
- 基于模型的策略：理解系统工作的概念模型，并试着深入理解所选定的区域。
- 随机应变策略：综合使用以上的这些方法。

在本部分中，我们同时使用技术方法和人类知识方法来描述 Linux内核的具体系统结构。随机应变的策略与[Tzerpos 1996]中所描述的混合方法实际上是同一个策略。我们没有使用Linux内核开发人员的实践知识，而是使用了现代操作系统的领域相关知识（例如，来自任务1的概念系统结构），用来反复精化Linux内核的具体系统结构。

## 1.4 方法与途径

在本部分中，我们使用随机应变策略来开发 Linux内核的具体系统结构。我们修改了在[Tzerpos 1996]中所描述的方法，并用它来判断 Linux内核的结构。所采取的步骤如下（但不一定按照这种顺序）：

- 定义概念系统结构。因为我们无法直接获得开发人员的实践知识，所以使用自己的现代操作系统的领域知识来创建 Linux内核的概念系统结构。这个工作是在任务 1中完成的（[Bowman 1998], [Siddiqi 1998]和[Tanuan 1998]）。
- 从源代码中提取事实。我们使用 Portable Bookshelf公司的C Fact Extractor (cfx)和Fact Base Generator (fbgen)（在[holt 1997]中介绍过），从源代码中提取出依赖性事实。
- 集成到子系统。我们使用 Fact Manipulator（例如，grok和grok脚本），把得到的事实集成到子系统中。集成工作有一部分是由工具完成的（使用文件名和目录），另外有一部分是使用Linux内核的概念模型完成的。
- 概览所生成的软件结构。我们使用 Landscape Layouter、Adjuster、Editor和Viewer（[Holt 1997]）来对抽取的设计信息进行可视化。基于这些图表，用户可以清晰地看出子系统之间的依赖性。 Landscape图表可以肯定用户对具体系统结构的理解。有些时候抽取的系统结构与概念系统结构并不吻合，这时我们需要手工检查源代码和文档。
- 使用概念系统结构优化集成工作。我们使用 Linux内核的概念系统结构来检查部件的集成，以及这些部件之间的依赖关系。

- 使用概念系统结构优化布局。延续前一个步骤，我们使用 Visio画图工具手工画出Linux内核结构的布局图。

根据目的和视角的不同， Linux内核有许多视图。在本部分中，我们使用软件结构（[Mancoridis Slides]）来描述具体的系统结构。使用软件结构可以完成下面的工作：

- 指定Linux内核划分为5个主要的子系统。
- 描述了部件（如子系统和模块）的接口。
- 描述了部件之间的依赖关系。

我们描述资源之间的依赖关系，这里所指的资源可以是子系统、模块、过程或者变量。依赖关系是个非常广泛的概念，通常我们不区分函数调用、变量引用和类型使用。

软件结构与Linux内核的运行时结构关系不大。然而，我们相信如果能够把软件结构和详细的规范结合起来，将可以给潜在的 Linux开发人员提供足够的信息，使他们无需读完所有的源代码就能修改或扩展内核。我们主要关心的不是 Linux内核的进程视图[Kruchten 1995]，这是因为我们把Linux内核当作是一个执行进程。

## 1.5 适用本书的读者

我们假设本书读者在计算机科学和操作系统方面具有足够的背景知识，可以透彻地理解本部分中关于Linux内核的主要部件以及部件之间的交互的讨论。我们并不要求读者对 Linux操作系统有太多的了解。

## 1.6 本部分的章节安排

本部分的剩下章节是这样安排的：

- 第2章讨论了整个系统的系统结构。它描述了系统的系统结构，展示了它的五个主要子系统，以及子系统之间的相互依赖关系。
- 第3章介绍了主要子系统（进程调度程序、内存管理程序、虚拟文件系统、进程间通信以及网络接口）的系统结构。我们将使用图表来帮助介绍子系统，以展示上下文中的子系统，并用线段来展示依赖关系。第 3章还解释了系统的抽象，以及从 Linux内核源代码抽取出来的设计信息。
- 第4章阐述了在本部分中我们所遇到的问题，并给出我们的发现，得出我们的结论。

# 第2章 系统结构

Linux内核本身并没有什么用，它只是整个系统中的一个层（[Bowman 1998]）。

在内核层中，Linux是由5个主要的子系统所组成的：进程调度程序（sched）、内存管理程序（mm）、虚拟文件系统（vfs）、网络接口（net）以及进程间通信（ipc）。对于一个创建好的系统结构，它的划分方法与概念系统结构的划分方法类似 [Siddiqi 1998]、[Tanuan 1998]和[Bowman 1998]。如果读者知道概念系统结构是从创建以后的系统结构抽取出来的，那么读者这不难理解上述的对应关系了。这里所采用的分解方法并不完全按照源代码的目录结构，因为我们相信目录结构并不完全匹配子系统的分组。然而，我们的集成是与该目录结构非常相近的。

在把抽取的设计细节可视化以后，读者可以认清一个区别，即子系统的依赖性和概念系统结构的依赖性是有很大不同的。概念系统结构显示出很少的系统间依赖关系，如图 5-2-1a所示（摘自[Bowman 1998]）。

尽管概念系统结构的依赖性相当少，具体系统结构显示出 Linux内核的5个主要的子系统之间存在着很强的依赖性。图 5-2-1-b显示子系统之间的连接，它与一个完全图的区别是只缺两条边（[PBS:kernel.html]详细介绍了哪个模块是跨子系统交互的）。这种相互依赖关系与概念系统结构有着很大的不同。这表明，任何基于互联属性的逆推技术（如在 [Müller 1993]中描述的Rigi系统）在从这样的系统中抽取任何相关结构时，都将会遭到失败。这就证明了Tzerpos（[Tzerpos 1996]）所介绍的混合方法的有效性。

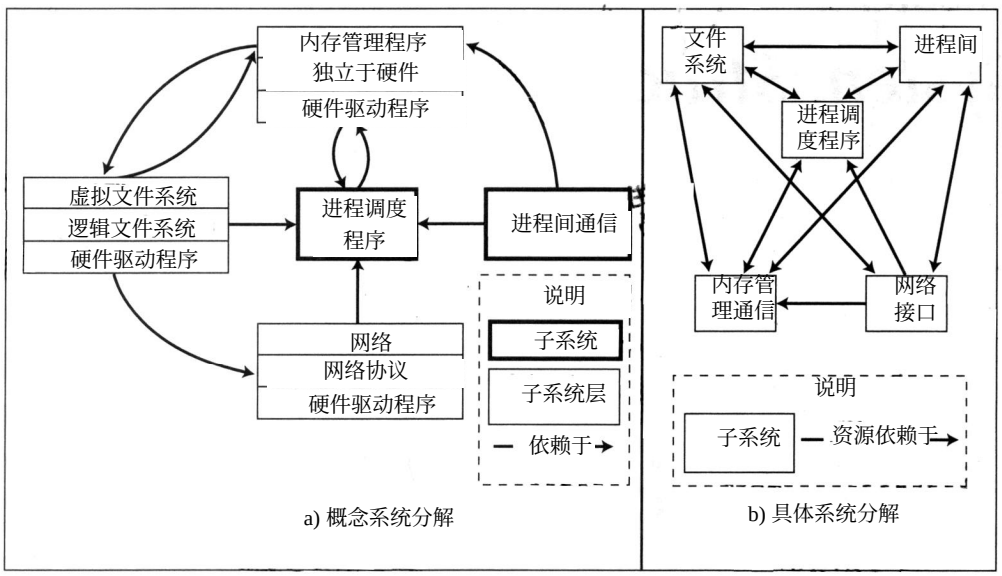


图5-2-1 概念系统的分解和具体系统的分解

系统级别上的差异是子系统级别上差异的延伸。子系统结构大体上对应着概念结构。然而，我们发现具体系统结构中的许多依赖性在概念系统结构中并不存在（在 [Murphy 1995]中称之为散度）。下面一节将会讨论这些附加的依赖关系的原因，在下一节我们将会详细介绍每个主要的子系统的设计。

# 第3章　子系统结构

## 3.1　进程调度程序

### 3.1.1　目标

进程调度程序是Linux操作系统的核心。进程调度程序需要完成以下任务：

- 允许进程创建它们自己的新拷贝。
- 判定哪个进程可以访问CPU，并影响运行进程之间的传输。
- 接收中断，并把它们转交给适当的内核子系统。
- 向用户进程发送信号。
- 管理定时器硬件。
- 当进程结束执行时清除进程资源。

进程调度程序还为动态装入的模块提供支持；动态装入模块是指在内核开始执行之后才能装入的内核功能。虚拟文件系统和网络接口将会用到这个可装入的模块功能。

### 3.1.2　外部接口

进程调度程序提供了两个接口：首先，它提供了一个可供用户进程调用的受限系统调用接口；其次，它为内核系统的其他部分提供了一个丰富的接口。

进程只能通过拷贝现有进程的方法创建其他的进程。在系统引导的时候， Linux系统只有一个运行进程：init。然后该进程再产生其他的进程，而其他进程又可以利用自身的拷贝产生新进程，方法是使用fork ( )系统调用。fork ( )调用会产生新的子进程，而子进程实际上是父进程的拷贝。在系统关闭的时候，某个用户进程（隐式地或者显式地）调用 _exit系统调用。

系统提供了几个过程来处理可装入模块。 create_module ( )系统调用将分配足够的内存,以便装入模块，该调用将初始化 module结构（下面将会介绍该结构），填入分配的模块的名称、大小、始地址以及初始状态。系统调用 init_module ( )将从磁盘装入模块并激活它。最后delete_module ( )将会卸装一个运行中的模块。

可以用系统调用setitimer ( )和getitimer ( )来完成定时器的管理，前者设置一个定时器，而后者则获取定时器的值。

在重要的信号函数之中，最重要的函数要算是 signal ( )了。这个函数允许用户进程把函数处理程序和特定的信号联系起来。

### 3.1.3　子系统描述

进程调度程序子系统主要负责用户进程的装入、执行以及正确的结束。在执行用户进程时，一般在两个不同的地方调用这个调度算法。首先，有一些系统调用可以直接调用调度程

序，比如sleep ( )。其次，在每一个系统调用之后，以及在每一个慢的系统中断之后（稍后再作介绍），可以调用调度算法。

信号可以认为是一种IPC机制，所以将在介绍进程间通信的一节中进行讨论。

中断允许硬件与操作系统进行通信。 Linux下的中断分为快速中断和慢中断。慢中断是典型的中断，当系统正在处理慢中断时，其他的中断是合法的，一旦处理完了慢中断， Linux就可以正常工作了，例如可以调用调度算法。定时器中断是慢中断的典型例子。相比之下，快中断所完成的任务要简单一些，例如处理键盘输入等等。在处理快速中断时，其他的中断将被屏蔽，除非快速中断处理程序显式地使能了其他的中断。

Linux OS使用的定时器每10ms发生一次定时器中断，这样，根据前面所讲的调度程序描述，应该至少每10ms发生一次任务调度。

### 3.1.4 数据结构

结构task_struct代表Linux任务，其中有一个域表示的是进程的状态，它的值可以是如下一些：

- 运行。
- 从系统调用返回。
- 处理中断过程。
- 处理系统调用
- 就绪
- 等待

此外，该结构中还有一个域，它表示进程优先级，另外还有一个域存放着时钟滴答（以10ms为周期）的数量，它表示进程可以持续执行多长时间而不用重新调度。该结构中还有一个域存放着最近一次出错系统调用的错误编号。

为了跟踪所有的执行进程，系统维护一个双链表（通过两个指向 task_struct的域）。因为每一个进程与其他一些进程相关，所以需要提供以下域来描述一个进程：原始父母、父母、最小子进程、弟弟进程，最后是兄长进程。

mm_struct是一个嵌套结构，它包含着进程的内存管理信息（例如代码段的始地址和结束地址）。

进程ID信息也存放在task_struct中。该结构中存放的信息包括进程 ID和组ID。同时还提供了组ID的一个数组，所以进程可以与多个组联系在一起。

文件相关的进程数据存放在 fs_struct子结构中。该结构中存放着一个指针，指向对应着处理器的根目录以及当前工作目录的索引节点。

由进程所打开的所有文件将通过 task_struct的子结构files_struct来跟踪。

最后，还有一些域提供计时信息；例如，进程在用户模式下花费的时间总量。

所有的执行进程在进程表中都有对应的一项。进程表被实现为指向任务结构的指针的一个数组。进程表中的第一个项目是特殊的 init进程，它是Linux系统所执行的第一个进程。

最后，还实现了一个 module结构来表示已经装入的模块。该结构中包含着一些域，可用来实现一个模块结构的列表：一个域指向模块符号表，另一个域存放着模块的名称。 Module结构中还存放着模块的大小（以页面为单位），以及指向模块始地址的指针。

### 3.1.5 子系统结构

图5-3-1显示了进程调度程序子系统。它以集成的方式来表示进程的调度和管理（例如：装入和卸装），以及定时器管理和模块管理功能。

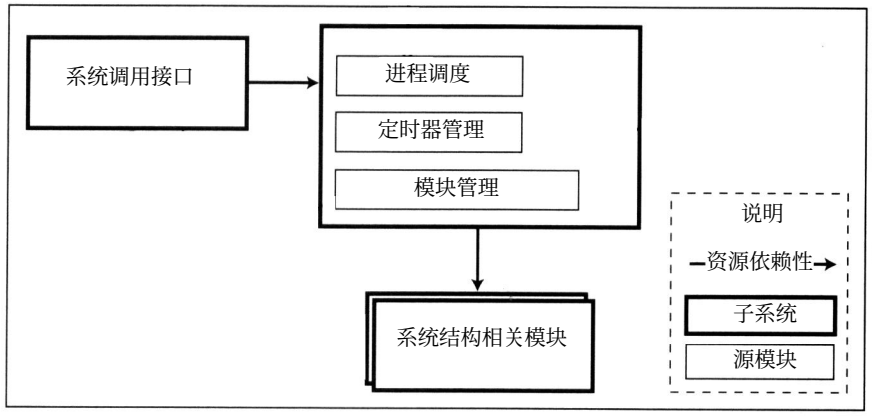


图5-3-1 进程调度程序结构

### 3.1.6 子系统依赖性

图5-3-2显示出进程调度程序是怎样依赖于其他内核子系统的。在调度进程时，进程调度程序需要内存管理程序来设置内存映射。此外，因为底半操作中（在 3.3 节中介绍）需要用到等待队列，所以进程调度程序依赖于IPC子系统。最后，进程调度程序依赖于文件系统从永久性设备中装入可装入的模块。所有的子系统都依赖于进程调度程序，因为它们需要在完成硬件操作的时候中断用户进程。如果读者希望了解更多关于子系统模块之间特定依赖性的知识，请参见 [PBS:kernel.html]。

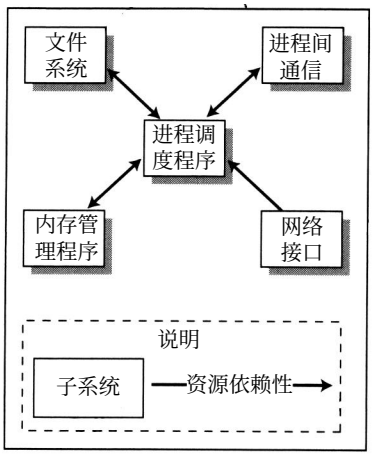


图5-3-2 进程调度程序依赖性

## 3.2 内存管理程序

### 3.2.1 目标

正如在[Rusling 1997]第13页到第30页中所介绍的那样，内存管理程序向它的客户提供以下一些功能：

- 大地址空间 —— 用户程序可以使用超过物理上实际所有的内存数量。
- 保护 —— 进程的内存是私有的，不能被其他进程所读取和修改；而且，内存管理程序可以防止进程覆盖代码和只读数据。
- 内存映射 —— 客户可以把一个文件映射到虚拟内存区域，并把该文件当作内存来访问。
- 对物理内存的公平访问 —— 内存管理程序确保所有的进程都能公平地访问计算机的内存资源，这样就可以确保理想的系统性能。

- 共享内存 —— 内存管理程序允许进程共享它们内存的一部分。例如，可执行代码通常可以在进程间共享。

### 3.2.2 外部接口

内存管理程序向它的功能提供了两个接口：用户进程所使用的系统调用接口，以及其他内核子系统用来完成任务的接口。

- 系统调用接口
- malloc ( )/free ( ) —— 分配或者释放一块供进程使用的内存区域。
- mmap ( )/munmap ( )/msync ( )/mremap ( ) —— 把文件映射到虚拟内存区域。
- mprotect —— 改变对虚拟内存区域的保护。
- mlock ( )/mlockall ( )/munlock ( )/munlockall ( ) —— 超级用户例程，防止内存被交换。
- swapon ( )/swapoff ( ) —— 超级用户例程，可以为系统增加或删除交换文件。
- 内核内部的接口
- kmalloc ( )/kfree ( ) —— 分配或者释放由内核的数据结构所使用的内存。
- verify_area ( ) —— 检查一下用户内存的某块区域是否以所需的许可权限映射。
- get_free_page ( )/free_page ( ) —— 分配和释放物理内存页面。

除了以上这些接口以外，内存管理程序允许在内核中使用它所有的数据结构以及大部分的过程。许多内核模块访问这些数据结构以及子系统的实现细节，它们通过这样的方式与内存管理程序相交互。

### 3.2.3 子系统描述

因为Linux支持许多硬件平台，所以内存管理程序中有一个与平台相关的部分，它把所有硬件平台的细节抽象成一个通用的接口。对硬件内存管理程序进行的所有访问都是通过这个抽象接口实现的。

内存管理程序使用硬件内存管理程序把虚拟地址（由用户进程所使用）映射到物理内存地址。当用户进程访问内存地址时，硬件内存管理程序把这个虚拟内存地址翻译成物理地址，然后使用物理地址来执行这次访问，因为存在这种映射关系，用户进程无需关心特定的虚拟内存地址是与哪个物理地址相联系的。这就允许内存管理程序子系统把进程的内存映射到物理内存的各个地方。此外，如果两个进程的虚拟内存地址空间区域映射到同一个物理地址空间，实际上这种映射方式允许这两个进程共享该物理内存。

此外，当某进程内存未使用时，内存管理程序会把它交换到分页文件中。这就允许系统可以执行那些需要使用较多内存的进程，即使它们所需的内存超过系统上实际所有的内存。内存管理程序包括一个守护进程（kswapd）。Linux使用的术语“守护进程”指的是内核线程。守护进程也是由进程调度程序来调度的，方法与调度用户进程的方法相同，但守护进程可以直接访问内核数据结构。这样，相对进程而言，守护进程的概念更接近于线程。

kswapd守护进程周期性地检查是否有任何物理内存页面最近没有使用到。如果有，则这些页面将被移出物理内存，如果需要则把它们存储在磁盘上。内存管理子系统特别注意减少所需的磁盘活动的数量。如果可以用另一种方法来达到目的，内存管理程序将避免把页面写以磁盘上。

如果用户进程所访问的内存地址当前没有映射到物理内存位置，则硬件内存管理程序将会检测到这一问题。它将把这次页面错误通知给 Linux内核，而解决这个问题是内存管理程序子系统的责任。这里存在两种可能性：或者该页面当前已被移出内存，这时需要把它再交换进来，或者用户进程是在对映射内存之外的内存地址作非法访问。硬件内存管理程序也能检测到对内存地址的非法调用，例如写可执行代码或者执行中的数据。这些调用也会招致页面错误，并被报告给内存管理程序子系统。如果内存管理程序检测到非法的内存访问，它用一个信号通知用户进程，如果进程不处理这个信号，它将被终止执行。

### 3.2.4 数据结构

下面的数据结构在系统结构上是相关的：

1) vm_area —— 内存管理程序为每个进程存储一个对应的数据结构，里面记录着哪块虚拟内存区域被映射到哪个物理页面上。这个数据结构中还存放着一个函数指针的集合，允许它在进程的某块虚拟内存区域上执行操作。例如，进程的可执行代码区域并不需要交换到系统分页文件中，因为它可以使用可执行文件来作后援存储。当进程的虚拟内存区域被映射时（例如在装入可执行文件时），虚拟地址空间中每一块连续的区域都将设置一个 vm_area_struct。因为在查看vm_area_struct以检查页面错误时，速度是非常关键的，所以该结构存放在 AVL树中。

2) mem_map —— 内存管理程序为系统上物理内存的每个页面都维护一个数据结构。该数据结构中存放着表示页面状态（例如，该页面当前是否正在使用）的标志。所有的页面数据结构都可以在一个向量（mem_map）中找到，该向量是在内核引导的时候进行初始化的，当页面状态变化时，该数据结构中的属性也相应地更新。

3) free_area —— free_area向量用于存储未分配的物理内存页面，当页面被分配时，需要从free_area中删除该页面，而当页面被释放时，它又返回到 free_area中。当从free_area分配页面时，需要使用Buddy系统[knowlton 1965; Tanenbaum 1992]。

### 3.2.5 子系统结构

内存管理程序子系统是由许多源代码模块组成的，按照它们的职责可以划分成如下的几个组（如图5-3-3所示）：

- 系统调用接口 —— 这组模块通过一个定义良好的接口（在前面介绍过），把内存管理程序的服务提供给用户进程。
- 内存映射文件（mmap） —— 这组模块负责受支持的内存映射文件 I/O。
- swapfile访问（swap） —— 这组模块控制内存交换。这些模块引起页调入和页调出的操作。
- 核心内存管理（core） —— 这些模块负责核心内存管理程序功能，其他内核子系统将需要使用该功能。
- 系统结构相关的模块 —— 这些模块为所有支持的硬件平台提供一个通用的接口，这些模块执行命令来改变硬件MMU的虚拟内存映射，并在发生页面错误时提供一种通用方法，用来通知内存管理程序子系统的其他部分。

内存管理程序结构的一个有趣之处在于使用了 kswapd，它是一个用于判定应该把哪个内

存页面交换出去的守护进程。 kswapd可以作为一个内核线程来执行，它周期性地检查使用中的物理页面，看看是否可以把哪个页面交换出去。这个守护进程是与内存管理程序子系统的其他部分并发执行的。

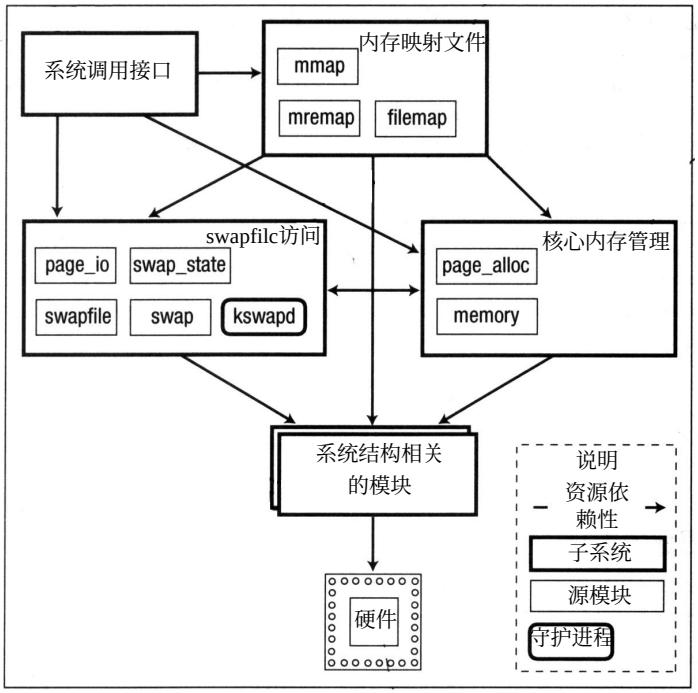


图5-3-3 内存管理程序结构

### 3.2.6 子系统依赖性

内存管理程序直接被每个 sched、fs、ipc和net所使用（通过数据结构和实现函数）。这种依赖性是很难用简洁的语言来描述的。如果读者希望了解更多有关子系统依赖性的详细知识，可以参考[PBS:mm.html]。图5-3-4展示了内存管理程序和其他子系统之间的高级依赖关系。为了简洁明了起见，图中没有画出内部依赖关系。

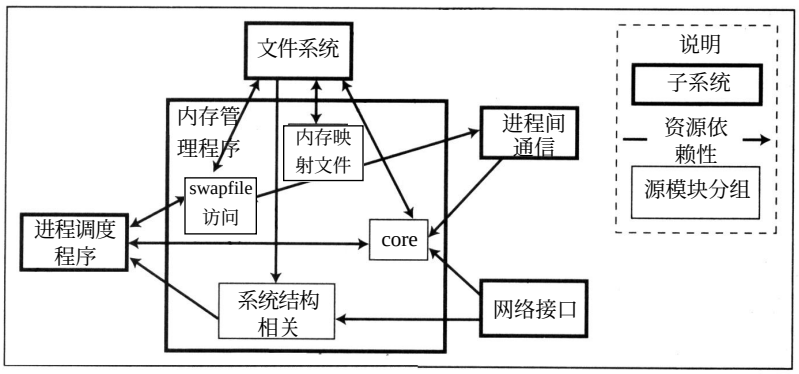


图5-3-4 内存管理程序依赖性

## 3.3 虚拟文件系统

### 3.3.1 目标

Linux在设计时就考虑到支持许多不同的物理设备。甚至就一种特定类型的设备而言，例如硬盘驱动器，在不同的硬件厂商之间也会存在许多接口上的差异。除了 Linux所支持的物理设备以外， Linux还支持大量的逻辑文件系统。正因为它能支持许多逻辑文件系统，所以Linux可以轻松地与其他操作系统进行互操作。 Linux文件系统支持下列目标：

- 多个硬件设备 —— 提供对许多不同的硬件设备的访问。
- 多个逻辑文件系统 —— 支持许多不同的逻辑文件系统。
- 多个可执行格式 —— 支持许多不同的可执行文件格式（例如 a.out、ELF、java）。
- 均一性 —— 为所有的逻辑文件系统以及所有的硬件设备提供了一个通用接口。
- 性能 —— 提供对文件的高速访问。
- 安全 —— 不会丢失或毁坏数据。
- 保密性 —— 限制用户访问文件的许可权限，限制分配给用户的总的文件大小。

### 3.3.2 外部接口

文件系统提供了两个级别的接口：用户进程所使用的系统调用接口，以及其他内核子系统所使用的内部接口。系统调用接口主要负责处理文件和目录。文件操作主要包括由 POSIX兼容系统所提供的一般的 open/close/read/write/seek/tell等操作，目录操作包括 POSIX系统的readdir/creat/unlink/chmod/stat等操作。

文件子系统用于支持其他内核子系统的接口要丰富一些。文件系统提供了一些数据结构和实现函数，以供其他内核子系统进行直接操作。特别是，它为内核的其他部分提供了两个接口 —— 索引节点和文件。其他内核子系统也会使用文件子系统的另一些实现细节，但用得不太普通。

索引节点接口

- create ( )：在目录中创建一个文件。
- lookup ( )：使用文件名称在目录中查找文件。
- link ( )/symlink ( )/unlink ( )/readlink ( )/follow_link ( )：管理文件系统连接。
- mkdir ( )/rmdir ( )：创建或删除子目录。
- mknod ( )：创建目录、特殊文件或者常规文件。
- readpage ( )/writepage ( )：从后援存储中读物理内存页面，或者把物理内存页面写入到后援存储中。
- truncate ( )：把文件的长度设置为0。
- permission ( )：检查一下用户进程是否具有执行某操作的权限。
- smap ( )：把逻辑文件块映射到物理设备扇区。
- bmap ( )：把逻辑文件块映射到物理设备块。
- rename ( )：重新命名文件或者目录。

除了以上这些用户调用索引节点的方法以外，还提供了 namei ( )函数，它允许其他内核子

系统可以找到与文件或目录相联系的索引节点。

文件接口

- open ( )/release ( )：打开或关闭文件。
- read ( )/write ( )：读或写文件。
- select ( )：一直等待，直到文件处于一个特殊的状态（可读或者可写）。
- lseek ( )：如果此功能受支持的话，则移动到文件中特定的偏移量处。
- mmap ( )：把一个文件区域映射到用户进程的虚拟内存。
- fsync ( )/fasync ( )：把任意内存缓冲区与物理设备同步。
- readdir：读某个目录文件所指向的文件。
- ioctl：设置文件属性。
- check_media_change：检查一下可移动媒体（如软盘）是否已被拿走。
- revalidate：确认所有高速缓冲的信息都是合法的。

### 3.3.3 子系统描述

文件子系统需要支持许多不同的逻辑文件系统和许多不同的硬件设备。它是通过拥有两个概念层而做到这一点的，这两个概念层很容易扩展。设备驱动程序层用一个通用的接口来表示所有的物理设备。虚拟文件系统层（ VFS）则用通用接口来表示所有的逻辑文件系统。Linux内核的概念系统结构（[Bowman 1998]、[Siddiqi 1998]展示了这种分解在概念上是怎样安排的。

### 3.3.4 设备驱动程序

设备驱动程序层负责向所有的物理设备提供一个通用接口。 Linux内核有三种类型的设备驱动程序：字符设备驱动程序、块设备驱动程序和网络设备驱动程序。其中与文件子系统相关的两种类型是字符设备和块设备。字符设备必须以串行的顺序依次访问，它的典型例子是磁带驱动器、调制解调器和鼠标。块设备可以用任意顺序进行访问，但是只能从多个块大小中读或者写入到多个块大小中。

所有的设备驱动程序都支持前面所介绍的文件操作接口。因此，每个设备都可以当作是文件系统中的一个文件来进行访问（这里所说的文件是指设备特殊文件）。因为内核的大部分都是通过这个文件接口来处理设备的，通过实现设备相关代码来支持这个抽象的文件系统，则加入新设备驱动程序的操作会变得简单一些。因为存在大量不同的硬件设备，所以越容易编写新的设备驱动程序越好，这一点是很重要的。

Linux内核在访问块设备时，使用一个缓冲区高速缓存来提高性能。所有对块设备的访问都是通过缓冲区高速缓存子系统发生的。因为缓冲区高速缓存可以减少对硬件设备的读写操作次数，所以它大大提高了系统的性能。每个硬件设备都有一个请求队列；当缓冲区高速缓存不能在内存缓冲区中满足用户请求时，它把请求加入到该设备的请求队列中，并开始睡眠，直到该请求被满足为止。缓冲区高速缓存使用一个独立的内核线程 kflushd，把缓冲区页面写到设备上，并把它们从高速缓存中删除。

当设备驱动程序需要满足请求时，它首先通过对设备的控制与状态寄存器（ CSR）进行操作，完成该硬件设备操作的初始化工作。有三种通用机制可以把数据从主计算机上移到分

层设备上：轮询、直接内存访问（DMA）以及中断。在使用轮询的情况下，设备驱动程序周期性地检查分层设备的CSR，看看当前请求是否已经完成了。如果已经完成，则驱动程序初始化下一个请求，再继续进行上述操作。对低速硬件设备如软盘驱动器和调制解调器而言，轮询是一种适合的机制，传输的另一种机制是DMA。在这种情况下，设备驱动程序将初始化计算机的主内存和分层设备之间的DMA传输。该传输和主CPU并发工作，在操作继续进行的过程中，允许CPU处理其他任务。当DMA操作完成以后，CPU接收到一个中断。在Linux内核中，中断处理是相当普遍的，它比其他两种方法都要复杂得多。

当硬件设备想要报告某些条件的变化时（例如鼠标键被按下，键盘键被按下），或者报告某操作的完成时，它向CPU发送一个中断，如果使能了中断，则CPU停止执行当前指令，开始执行Linux内核的中断处理代码。内核找到应该调用的合适的中断处理程序（每个设备驱动程序都会注册一些处理程序，用于处理该设备所产生的中断）。当正在处理某中断时，CPU在特殊的上下文中执行，其他的中断将被推迟，直到该中断处理完毕。因为这一限制，中断处理程序的效率需要相当高，这样才不会丢失其他的中断。有时候在时间限度内中断处理程序不能完成所需的全部工作，在这种情况下，中断处理程序在底半处理程序中调度余下的工作。底半处理程序是一段代码，它是下一次完成系统调用时调度程度将要执行的代码。通过把不太重要的工作交给底半处理程序去完成，设备驱动程序可以减少中断延迟，提高并发性。

总而言之，设备驱动程序隐藏了对分层设备的CSR进行操作的细节，以及每个设备的数据传输机制。对块设备而言，缓冲区高速缓存试图在内存缓存区中满足文件系统请求，这样就提高了系统的性能。

### 3.3.5 逻辑文件系统

尽管可以通过设备特殊文件来访问物理设备，通过逻辑文件系统来访问块设备要更普遍一些。逻辑文件系统可以挂装在虚拟文件系统的挂装点上。这意味着相联系的块设备包含着文件和结构信息，这些信息允许逻辑文件系统访问该设备。在任意的时刻，一个物理设备只能支持一个逻辑文件系统。然而，该设备可以被重新格式化，以便支持另一个逻辑文件系统。在进行写操作时，Linux支持15个逻辑文件系统，这提高了Linux与其他操作系统的互操作性。

当文件系统被作为子目录挂装时，在该设备上的所有目录和文件都将被视为挂装点的子目录。虚拟文件系统的用户不需要考虑哪种逻辑文件系统实现了目录树的哪个部分，也无需考虑哪个物理设备包含着这些逻辑文件系统。这种抽象性在选择物理设备和逻辑文件系统时，提供了很大的方便，而这种方便性是Linux操作系统成功的重要因素之一。

为了支持虚拟文件系统，Linux使用了索引节点的概念。Linux使用索引节点来表示块设备上的文件。索引节点中包含了一些操作，根据文件所驻留的逻辑系统和物理系统的不同，这些操作的实现方法也会有所变化。从这层意义上讲，索引节点是虚拟的。索引节点接口使所有的文件在其他Linux子系统看起来都是一样的。索引节点可以用作一个存储位置，存储磁盘上与某个打开的文件相关的所有信息。索引节点存储相关缓冲，以块为单位的文件总长度，以及文件偏移量和设备块之间的映射。

### 3.3.6 模块

虚拟文件系统的大部分功能都是以动态装入模块的形式（参见3.1节）提供的。这种动态

配置的特征使得Linux用户可以编译一个尽可能小的内核，如果在单个操作过程中需要的话，还可以允许内核装入所需的设备驱动程序和文件系统模块。例如， Linux系统可能会有一个打印机连接到它的并行口。如果打印机驱动程序永远链接在内核中，则当没有打印机时内存将会浪费。通过使打印机驱动程序成为一个可装入的模块， Linux允许用户在需要使用打印机时装入驱动程序。

### 3.3.7 数据结构

下面这些数据结构在系统结构上是与文件子系统相关的：

- super_block：每个逻辑文件系统都有一个相联系的超级块，用于把它提供给 Linux内核的其他部分，该超级块包含着整个挂装文件系统的有关信息 —— 正在使用哪些块、块的大小有多大等等。超级块与索引节点的相似点在于：它们均可作为逻辑文件系统的虚拟接口。
- inode：索引节点是一个内存中的数据结构，它表示内核需要知道的有关磁盘上某文件的所有信息。单个索引节点可以被打开该文件的所有进程使用。索引节点存储着内核需要与某个文件相关联的所有信息。计帐、缓冲以及内存映射信息都存储在索引节点中。一些逻辑文件系统在磁盘上也会有一个索引节点结构，可以永久性地保存这一信息，但这与内核的其他部分所使用的索引节点数据结构是不同的。
- fite：文件结构代表由特定进程打开的文件。所有打开的文件都存储在一个双链表中（由first_file所指向），在POSIX形式的过程（open、read、write）中所使用的文件描述符是这个链表中某个打开文件的索引。

### 3.3.8 子系统结构

（原文缺）

### 3.3.9 子系统依赖性

图5-3-5、图5-3-6显示了文件系统是怎样依赖于其他内核子系统的。再说一遍，文件系统依赖于所有其他的内核子系统，而所有其他的内核子系统也依赖于文件子系统。特别是，网络子系统之所以依赖于文件系统，是因为网络套接字是作为文件描述符提供给用户的。内存管理程序依赖于文件系统以支持交换。 IPC子系统依赖于文件系统以实现管道和 FIFO。进装调度程序依赖于文件系统来装入可装入的模块。

文件系统使用网络接口来支持 NFS；它使用内存管理程序来实现缓冲区高速缓存以及ramdisk设备；它使用IPC子系统来帮助支持可装入模块；当正在进行硬件请求处理时，它还使用进程调度程序把用户进程置为睡眠状态。如果想要了解依赖性方面的更详细的知识，请参阅[PBS:fs.html]。

## 3.4 进程间通信

### 3.4.1 目标

之所以提供Linux IPC机制，是为了给并发执行的进程提供一种方法，是它们可以共享资

源，与其他进程同步并且交换数据。在相同系统上执行的进程之间， Linux通过共享资源、内核数据结构以及等待队列实现了各种形式的 IPC机制。

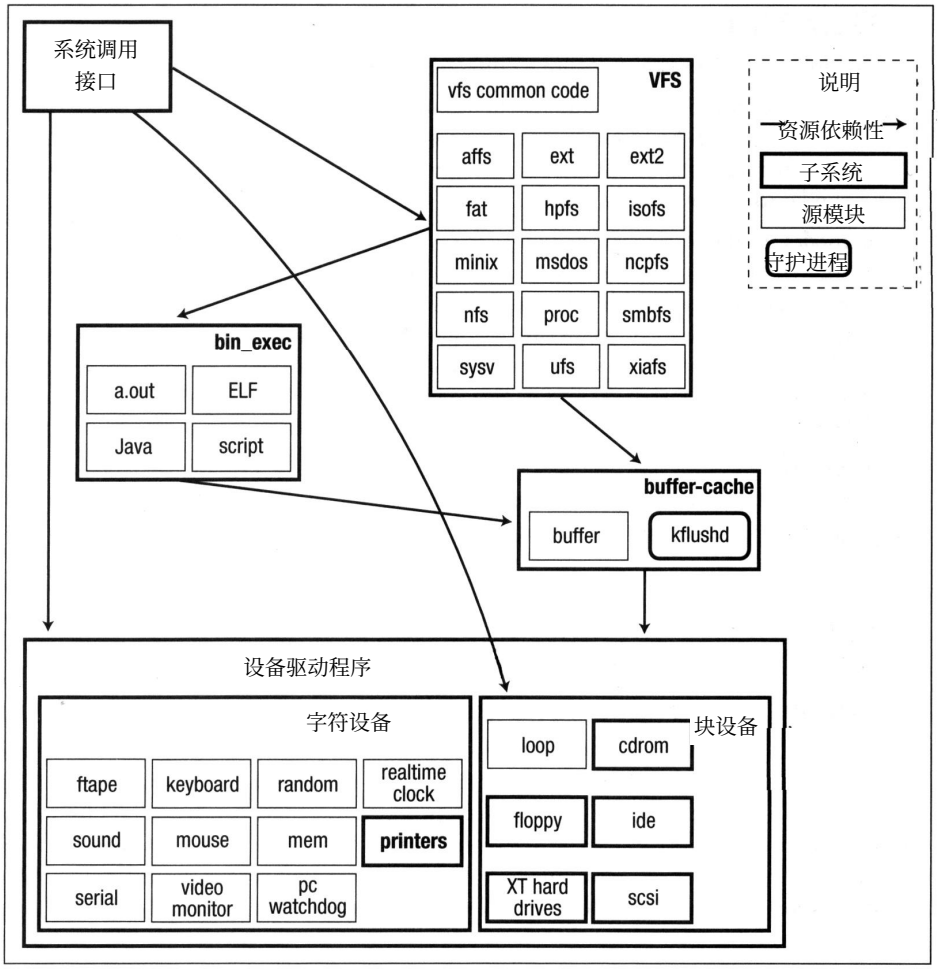


图5-3-5 文件子系统结构

Linux提供了下列形式的IPC机制：

- 信号 —— 可能是最古老的 Unix IPC形式。信号是发往某进程的异步消息。
- 等待队列 —— 当进程正在等待某操作的完成时，等待队列提供了一种机制，可以把进程置为睡眠状态。进程调度程序使用这一机制来实现 3.3.3节所介绍的底半处理操作。
- 文件锁 —— 它提供了一种机制，允许进程声明文件的一个区域，或者整个文件本身，把它们声明为对所有进程都是只读的，除了拥有文件锁的进程以外。

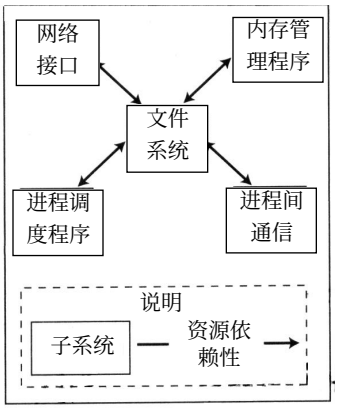


图5-3-6 文件子系统依赖性

- 管道和命名管道 —— 允许在两个进程之间进行面向联接的单向的数据传输，方法可以是显式地建立管道连接，或者通过驻留在文件系统中的命名管道进行通讯。
- 系统V IPC
- 信号量 —— 传统信号量模型的一种实现，该模型还允许创建信号量数组。
- 消息队列 —— 一种无联接的数据传输模型。消息是字节的序列，并带有相应的类型。消息可以写入到消息队列中，并且可以通过从消息队列中读取来获得消息，当然也可以限制读入哪种类型的消息。
- 共享内存 —— 通过使用这种机制，几个进程可以访问物理内存的同一块区域。
- Unix域套接字 —— 另一种面向连接的数据传输机制，它提供了与 INET套接字相同的通信模型。INET套接字将在下一章进行介绍。

### 3.4.2 外部接口

信号是内核或者另一个进程发送给某个进程的通知。信号是使用 send_sig ( )函数发送的。信号编号以及目标进程都必须作为参数提供给该函数。进程可以通过使用 signal ( )函数来注册处理信号。

文件锁直接由Linux文件系统支持。为了锁定整个文件，可以使用系统调用 open ( )，或者使用系统调用sys_fcntl ( )。锁定文件中的几个区域也可以通过 sys_fcntl ( )系统调用来完成。

管道可以使用pipe ( )系统调用来创建。然后可以使用文件系统的 read ( )和write ( )调用在管道上传输数据。命名管道使用 open ( )系统调用来打开。

系统V IPC机制有一个通用接口，它就是 ipc ( )系统调用。使用该系统调用的参数可以指定各种IPC操作。

Unix域套接字功能也封装在单个系统调用 socketcall ( )中。

上面所提到的这些系统调用的文档都编制得很好，提倡读者去查询相应的 man帮助页面。

IPC子系统向其他内核子系统提供等待调用。因为用户进程不能使用等待队列，所以它们没有系统调用接口。等待队列可以用于实现信号量、管道以及底半操作处理程序（参见 3.3.3节）。过程add_wait_queue（）把一个任务插入到等待队列中。过程 remove_wait_queue（）则把任务从等待队列中删除。

### 3.4.3 子系统描述

下面简要描述一个3.4.1节中所列出的每个IPC机制的低级功能。

信号用于把事件通知给进程，根据特定信号的语义不同，信号可以改变接收进程的状态。内核可以向任意执行中的进程发送信号。只有在拥有相应的 PID或GID的情况下，用户进程才能向进程或者进程组发送信号。对静止的进程来说，信号并不立即处理。相反，在下一次调度程序设置进程使其在用户模式中运行以前，它检查是否有个信号被发送给该进程，如果是，则调度程序调用do_signal ( )函数，该函数可以正确地处理信号。

等待队列只是指向任务结构的指针的链表，该链表对应着等待某内核事件的进程。内核事件的例子有DMA传输的结束等。进程可以把自己加入到等待队列中，方法是调用 sleep_on ( )或者interruptable_sleep_on ( )函数。与之相对应，函数wake_up ( )和wake_up_interruptable ( )可以把进程从等待队列中删除。中断例程也使用等待队列来避免条件竞争。

Linux使用户进程可以防止其他进程访问某文件。这种排它性可以基于整个文件，也可以基于文件的一块区域。文件锁可以用于实现这种排它性。文件系统实现包括在数据结构中适当的数据域，该域允许内核去判断该锁锁定的是文件，还是文件中的区域。如果是前者，则试图去锁一个已经锁定的文件将会出错；而如果是后者，则试图去锁一个已经锁定的区域也将出错。在两种情况下，请求进程均不允许访问文件，这是因为内核还没有确认该锁。

管道和命名管道具有相似的实现，这是因为它们的功能也几乎是相同的。虽然它们的创建过程是不同的。然而，无论是创建管道还是命名管道，都将会返回一个文件描述符，该文件描述符就指向该管道。在创建以后，把一个内存页面与打开的管道联系在一起。这个内存被当作环形缓存区来处理，在该缓冲区中，写操作是以原子操作的方式来完成的，当缓冲区已满后，写进程将阻塞。如果读请求所要读的数据量超出了现有的数据量，则读进程将阻塞。这样一来，每个管道相应该有一个等待队列与之相连。在读和写的过程中，进程将被增加到队列中，或者从队列中删除。

信号量是使用等待队列来实现的，它遵循的是典型的信号量模型。每个信号量都有一个与之相联系的值。在信号量上实现了两个操作 up ( )和down ( )。当信号量的值为零时，在信号量上执行减一操作的进程将在等待队列上阻塞。信号量数组只是信号量的一个连续的集合。每个进程还保留着它执行过的信号量操作的列表，这样一来，如果进程过早地退出，这些操作可以撤消。

消息队列是一个线性列表，进程可以从消息队列读字节的一个序列，或者把字节序列写到消息队列中。消息的接收顺序与当时写入它们的顺序是一致的。有两个等待队列与消息队列相联系，一个用于存放想要把消息写入到一个已满的消息队列的进程，而另一个则用于串行化消息的写操作。消息的实际大小在创建消息队列时设置。

共享内存是IPC最快捷的方式。这个机制允许进程共享它们内存的一个区域。共享内存区域的创建工作是由内存管理系统来处理的。通过调用系统调用 sys_shmat ( )，可以把共享页面与用户进程的虚拟内存空间连接在一起。通用调用 sys_shmdt ( )调用，可以把共享页面从进程的用户段删除掉。

Unix域套接字的实现方式与管道相似，因为它们都是基于环形缓冲区来实现的，而环形缓冲区又是基于内存页面的。然而，套接字为每个通信方向都提供了一个独立的缓冲区。

### 3.4.4 数据结构

本节将介绍用于实现上述IPC机制所需的一些重要的数据结构。

信号是通过task_struct结构中的signal域来实现的。每个信号都由这个域中的一个位来表示。这样一来，Linux的某个版本所能支持的信号的数目就不能超过一个字中位的数目。在域block中，存放着进程所阻塞的信号。

等待队列只有一个数据结构与之相联系，即 wait_queue结构。这些结构包含着一个指针，指向相联系的task_struct，并被链接到一个列表中。

文件锁有一个相联系的 file_lock结构。这个结构中存放的信息有：一个指向拥有进程的task_struct结构的指针、被锁定文件的文件描述符、等待取消文件锁的进程的等待队列，以及被锁定的文件区域。每个打开的文件的 file_lock结构均被链接到一个列表中。

管道都是由文件系统索引节点来表示的，不管它是无名管道还是命名管道。该索引节点

在pipe_inode_info结构中存放着附加的一些管道相关的信息。这个结构中存放着一个等待队列，用于存储在读或写操作上阻塞的进程，它还存放着指向内存页面的指针（该页面被用作管道的环形缓冲区）、在管道中数据的数量，以及当前正在读写管道的进程的数量。

所有的系统V IPC对象都是在内核中创建的，每个对象都有与之相联系的访问许可权限。这些访问许可权限存放在ipc_perm结构中。信号量是用sem结构来表示的，该结构中存放着信号量的值，以及最后一次在信号量上执行操作的进程的 pid。信号量数组是用semid_ds结构来表示的，它存放着访问许可权限、最近一次执行信号量操作的时间、指向数组中第一个信号量的指针，以及执行信号量操作时阻塞的进程的队列。结构 sem_undo可用于创建某进程所执行的信号量操作的一个列表，这样在该进程被杀掉时，可以撤消这些操作。

消息队列是基于结构msquid_ds的，该结构中存放着管理和控制信息。这个结构中存储着下列域：

- 访问许可权限。
- 用于实现消息队列的链接字段（例如，指向 msquid_ds的指针）。
- 最后一次发送、接收和修改的时间。
- 进程阻塞的队列，和前一节所介绍的那样。
- 当前在队列中的字节数量。
- 消息的数量。
- 队列的大小（以字节为单位）。
- 最后一个发送者的进程编号。
- 最后一个接收者的进程编号。

消息本身是用msg结构存储在内核中的。该结构存放一个链接域，用来实现消息的一个链表，它还存放着消息的类型、消息数据的地址，以及消息的长度。

共享内存的实现是基于 shmid_ds结构的。该结构与msquid_ds结构相似，也是存放一些管理和控制信息。 shmid_ds结构中存放着访问控制许可权限、最后一次连接、断开连接和修改的时间、创建者的pid、最后一次对该共享段执行操作的进程的 pid、共享内存段所连接的进程的数量、组成共享内存段的页面数量，以及页面表项目的一个域。

Unix域套接字是基于socket数据结构的，该结构将在网络接口那一节（ 3.5）中介绍。

### 3.4.5 子系统结构

图5-3-7显示了IPC子系统资源的依赖性，控制从系统调用层向下流入每个模块中。系统 V IPC工具都是在内核源代码的ipc目录中实现的。内核IPC模块调用在kernel目录中实现的IPC工具。文件工具和网络IPC工具的实现与此类似。

系统V IPC模块依赖于内核IPC机制。特别是，信号量是用等待队列实现的。其他所有的IPC工具在实现上都是相互独立的。

### 3.4.6 子系统依赖性

图5-3-8显示了IPC子系统与其他内核子系统之间的资源依赖性。

IPC子系统是因为套接字而依赖于文件系统的。套接字使用文件描述符，并且一旦打开它们，它们都会被赋予一个索引节点。内存管理之所以依赖于 IPC，是因为页面交换过程调用

IPC子系统来实现共享内存的交换。 IPC之所以依赖于内存管理，主要是因为缓冲区的分配以及共享内存的实现。

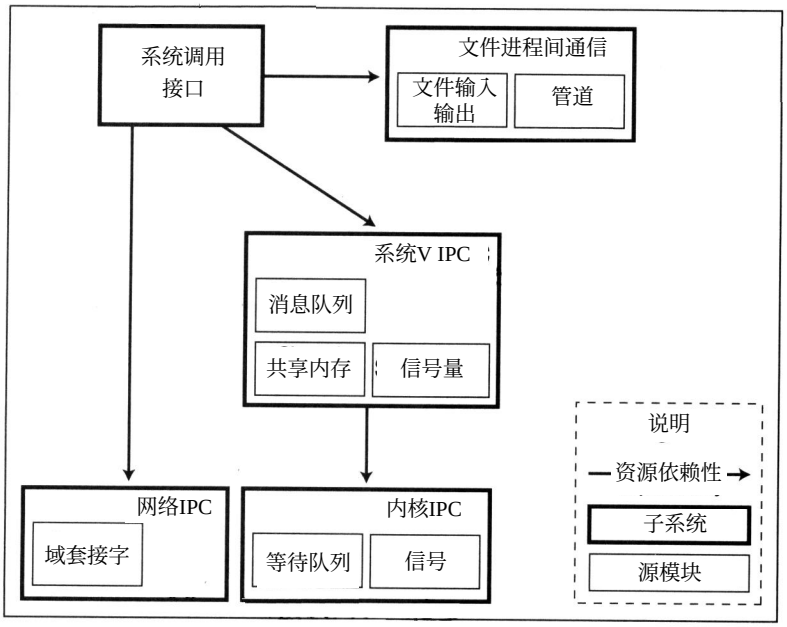


图5-3-7 IPC子系统的结构

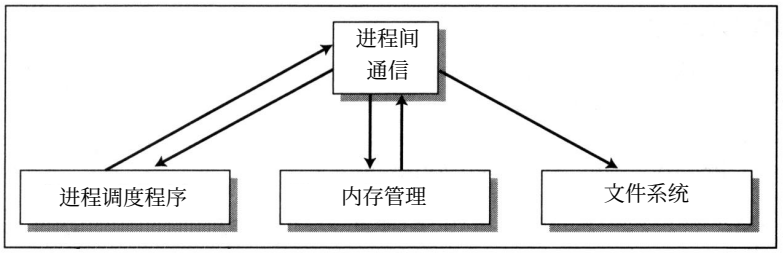


图5-3-8 IPC子系统的依赖性

一些IPC机制赖使用定时器。定时器是在进程调度程序子系统中实现的。进程调度主要依靠信号。因为这两个原因， IPC和进程调度程度模块必须相互依赖。如果读者想要了解 IPC子系统模块和其他内核子系统之间依赖性的更多知识，可以参考 [PBS:ipc.html]。

## 3.5 网络接口

### 3.5.1 目标

Linux网络系统提供了计算机之间的网络连接，以及套接字通信模型。 Linux总共提供了两种类型的套接字实现：BSD套接字和INET套接字。BSD套接字是使用INET套接字实现的。

Linux网络系统提供了两种传输协议，它们的通信模型和服务质量是不同的。这两种传输协议是不可靠的，基于消息的 UDP协议；以及可靠的流式 TCP协议。它们都是在 IP网络协议的基础上实现的。 INET套接字则同时基于传输协议和 IP协议来实现。

最后，IP协议是位于基本设备驱动程序的上一层。设备驱动程序可以为三种不同类型的连接而提供：串行联接（SLIP）、并行联接（PLIP）以及以太网联接。在IP和以太网驱动程序之间，存在一个地址解析协议。这个地址解析协议的任务是在逻辑 IP地址和物理以太网地址之间进行翻译。

### 3.5.2 外部接口

其他子系统和用户是通过套接字接口来使用网络服务的。套接字可以通过socketcall ( )系统调用来创建和操作，通过在套接字文件描述符上使用read ( )和write ( )调用，可以发送和接收数据。

网络子系统不提供其他的网络机制和功能。

### 3.5.3 子系统描述

BSD套接字模型是提供给用户进程使用的。该模型是面向联接的、流式的，并且是缓存的通信服务。BSD套接字是在INET套接字模型上实现的。

BSD套接字模型处理的任务与 VFS相似，它还为套接字联接管理一个通用的数据结构。BSD套接字模型的目标是：通过把通信细节抽象成通用接口来提供更大的可移植性。 BSD接口在现代操作系统（如Unix和Microsoft Windows）中广泛使用。INET套接字模型为基于IP的UDP和TCP协议管理实际通信端点。

网络I/O是以读写套接字开始的。它将调用一个 read/write系统调用，而该调用由虚拟文件系统的部件进行处理（对网络子系统层的 read/write调用是对称的，所以从现在起，我们只考虑写操作）。那个部件判定BSD套接字sock_write ( )是用于实现实际的文件系统写调用的，所以它将调用sock_write ( )。该过程处理管理细节，并把控制传送给 inet_write ( )函数。而该函数反过来又调用传输层写调用（如 tcp_write ( )）。

传输层写协议负责把到达的数据分成传输报文。这些过程把控制传送给ip_build_header ( )过程，它创建一个IP协议报头，以便插入到欲发送的报文中，然后再调用tcp_build_header ( )来创建TCP协议报头。一旦完成了以上工作，就可以使用底层的设备驱动程序来进行实际数据发送了。

网络子系统提供两种不同的传输服务，每一种的通信模型和服务质量都是不同的。 UDP提供了一个无联接的、不可靠的数据传送服务。它负责从 IP层接收报文，并找到应该把报文数据发送到哪个目标套接字。如果没有提供目标套接字，则将会报错。否则，如果有足够的缓冲区内存，报文数据将加入到套接字接收的报文列表中。在读操作上睡眠的所有套接字均获得通知，并被唤醒。

TCP传输协议提供了一个更为复杂的方案。除了处理发送进程和接收进程之间的数据传输以外，TCP协议还执行复杂的联接管理。TCP把数据以流的形式向上传送，而不是以报文序列的形式发送，发送目标是套接字层，并保证提供一个可靠的传输服务。

IP协议提供报文传输服务。给定一个报文，以及报文的目标， IP通信层将负责把该报文传送到正确的主机上。对于一个向外流出的数据流， IP负责以下工作：

- 把该流分成IP报文。
- 把IP报文传送到目标地址。
- 产生一个报文头，以供硬件设备驱动程序使用。
- 选择发送到合适的网络设备上。

对于一个到达的报文流，IP必须执行以下操作：

- 检查报文头，以确认其合法性。
- 把目标地址和本地地址相比较，如果报文没有到达正确的目标，则把它向前移动。
- 解析IP报文。
- 把报文向上发送到TCP或者UDP层，以便进一步处理。

ARP（地址解析协议）负责转换 IP地址和实际硬件地址。 ARP支持各种硬件设备，如以太网、 FDDI等等。因为套接字是处理 IP地址的，而底层硬件设备不能直接使用 IP地址，所以ARP的功能是非常必要的。因为使用的是一个中性的编址方案，所以同一个通信协议可以在各种硬件设备间实现。

网络子系统为串行联接、并行联接以及以太网联接提供了自己的设备驱动程序。为了隐藏通信媒体之间差异，不让网络子系统的上层知道，这里提供了一个各种硬件设备的抽象接口。

### 3.5.4 数据结构

图5-3-9显示了网络子系统的结构。

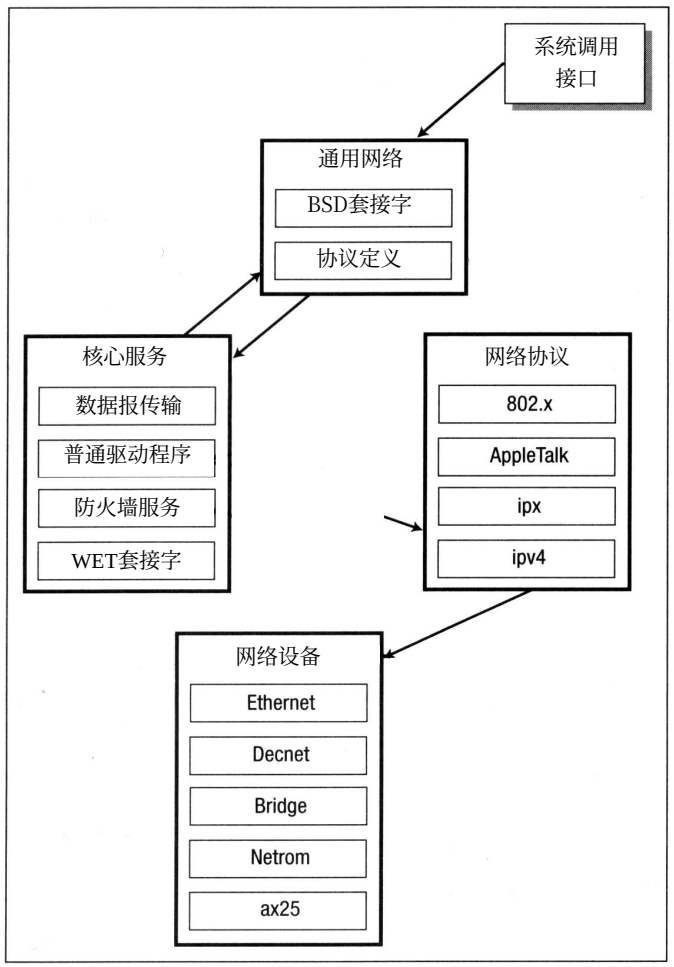


图5-3-9 网络子系统的结构

BSD套接字实现是用 socket结构来表示的。它包含着一个域，该域标识了套接字的类型

(流式的或者数据报的)，该域还标识套接字的状态（联接还是未联接)，该结构中还包含一个存放着标志的域，该标志修改了套接字的操作。除此之外，它还包含一个指针，指向一个结构，该结构用于描述可以在该套接字上执行的操作。另外 socket结构还包含一个指针，指向相联系的INET套接字实现，以及对索引节点的引用。每个 BSD套接字都是与一个索引节点相联系的。

结构sk_buff可以用于管理单个通信报文。该缓冲区指向它所属的套接字，其中包含上一次它被传输的时间，以及一个链接域，这样与给定套接字相联系的所有报文都可以链接在一起（在一个双链表中)。在该缓冲区中存放着源地址和目的地址、头部信息以及报文数据。该结构封装了建网系统所使用的所有报文（例如 TCP报文、UDP报文、IP报文等等)。

sock结构调用INET套接字相关的信息。该结构的成员包括套接字申请读写内存的次数，TCP协议所需的序列号、可以设置用来改变套接字行为的标志、缓冲区管理域（例如为了保存给定套接字所接收到的所有报文的列表)，以及阻塞读和阻塞写的等待队列。除此以外，该结构还提供了一个指针，指向保存一个函数指针列表的结构，而这些函数指针又是用来处理协议相关过程的。最后， sock结构中还包含proto结构。proto结构更大更复杂，但是最重要的是，它向TCP和UDP协议提供了一个抽象接口。还提供了源地址和目的地址，以及更多的 TCP相关数据字段。TCP使用定时器是为了处理时间消耗完这一事件。这样， sock结构中包含着与定时器操作相适应的数据域，以及一些函数指针，它们可以用作定时器响铃的回调处理。

最后，device设备被用于定义网络设备驱动程序。这与用来表示文件系统设备驱动程序的结构是同一个结构。

### 3.5.5 子系统结构

通用网络包含着那些向用户进程提供最高级接口的模块。这些主要是 BSD套接字接口，另外通用网络还包括网络层所支持的协议的定义。这里所包含的协议包括 MAC协议802.x、ip、ipx以及Apple Talk。

核心服务与高级实现工具相对应，这些工具包括 INET套接字、支持防火墙、通用网络设备驱动程序接口以及数据报和 TCP传输服务的工具。

系统调用接口与 BSD套接字接口相交互。 BSD套接字层为套接字通信提供了一个通用的抽象，这种抽象是使用 INET套接字来实现的。这就是通用网络模块与核心服务模块之间存在依赖性的原因。

该协议模块包含着一些代码，用于取得用户数据，并用特定协议所需的格式来格式化这些数据。协议层最终会把数据发送到合适的硬件设备，因此导致网络协议模块和网络设备模块之间存在依赖性。

网络设备模块包含设备类型相关的一些高级过程。实际的设备驱动程序与常规设备驱动程序一起驻留在目录drivers/net中。

### 3.5.6 子系统依赖性

图5-3-10显示了网络子系统与其他子系统间的依赖性。

网络子系统因为缓冲区而依赖于内存管理系统。文件系统用于给套接字提供一个索引节点接口。文件系统使用网络系统来实现 NFS。网络子系统需要使用 kerneld守护进程，所以它

依赖于IPC。网络子系统使用进程调度程序来完成计时器功能，并在执行网络传输时中断进程。如果读者想要了解更多有关网络子系统模块与其他内核子系统之间依赖性的知识，可以参考[PBS:net.html]。

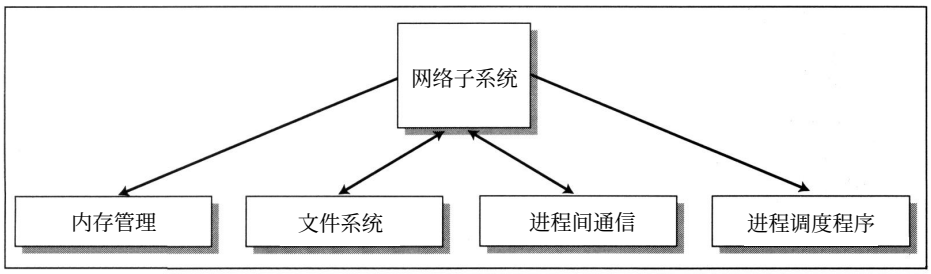


图5-3-10 网络子系统依赖性

# 第4章 结 论

总而言之，我们概览了一遍现有的文档，抽取出源文件事实，把模块集成到子系统中，并使用现代操作系统的领域知识优化了这种集成工作。我们发现，具体系统结构与我们以前所形成的概念系统结构并不匹配。最初抽取的结构显示，由于抽取工具的限制，我们遗失了依赖性。在人工确认了这些依赖关系以后，我们发现所有的失误都是由于这些工具的限制所造成的。另一方面，最后得到的具体系统结构展示了几个在概念系统结构中我们没有考虑到的依赖性。这些依赖性是由于一些设计决定所造成的。例如让 IPC子系统执行一些内存管理交换函数。这些预想之外的依赖性在 [Murphy 1995]中称为散度，它显示出概念系统结构的表示方式还不太准确，或者具体系统结构没有使用到合适的集成机制。然而，我们相信这些散度是Linux开发人员忽略系统结构方面的考虑而造成的。

我们浏览了Linux内核的可用文档。当前有着大量的系统文档，如 [Rulsing 1997]，甚至还有一些随源代码一起提供的受限文档。然而，这种文档因为层次太高，不适合作为详细的具体系统结构。

我们使用cfx工具从Linux内核代码中抽取详细的设计信息。cfx抽取工具产生了100,000个事实，这些事实使用grok可以组合成1,600个源模块中的15,000个依赖关系。在考察了事实抽取工具的输出以后，我们注意到结果存在两个问题。第一，有些依赖关系没有被检测出来。特别是在隐式调用的情况下更容易出现这个问题。在隐式调用的情况下，子系统使用另一个子系统注册它的功能，因为对注册子系统的访问是通过函数指针完成的，所以cfx工具不能检查出这种依赖关系。第二，cfx工具会报告一些不存在的依赖性。这种情况的一个例子发生在IPC子系统中，在IPC子系统中有三个源模块具有相同名称的函数（findkey)。因为cfx工具使用的是名称相等假设，它会报告说这三个模块相互依赖，而人工检查的结果却显示出这三个模块并不互相依赖。

除了由cfx工具所引入的一些假象以外，我们遇到另外一些问题，它们是由于 grok脚本假设每个源模块（*.c）都有一个相关的头文件（ *.h）而引起的。在Linux内核中这个假设一般是对的，但是有些头文件在许多源模块中都实现了。例如， mm.h在内存管理程序子系统的多个源模块中都有实现。因为对函数所作的调用都被抽象成对声明函数的头文件的调用，在把源模块分到子系统中时，我们就会遇到困难。所有的源模块都好象依赖于 mm.h，而不是依赖于实现所使用资源的特定源模块。可以用修改 grok脚本的方法来纠正这个问题，也许某些能够识别出头文件（如mm.h）的特殊本质的综合方法可能会更有用。

因为存在这些问题（丢失依赖性、伪造依赖性或者依赖性定位不准），我们需要人工检查抽取事实中与我们的期望不同的地方。不幸的是，因为源代码数量很大，而我们自己 Linux的经验又相对有限，所以我们所抽取的事实不能认为是完全正确的。

## 将来的工作

应该调整PBS工具，以便处理Linux源结构，我们所提供的概念系统结构和具体系统结构应该优化，方法是向Linux开发人员讨教，在优化之后，可以使用反射模型 [Murphy 1995]把这两个模型进行比较。

# 附录A 术语定义

BSD

Berkeley系统发布（Berkeley System Distribution）的缩写，这个Unix版本是基于AT&T系统V Unix，它是由加利福利亚大学Berkeley分校的计算机系统研究小组开发的。

**设备特殊文件（device special file）**

在该文件系统中的每个文件都表示一个物理设备。对此文件进行读写操作将导致对该物理设备的直接访问。这些文件很少直接使用，除了字符模式的设备以外。

fs

Linux内核的文件系统子系统。

**中断延迟（interrupt latency）**

中断延迟是在把中断报告给 CPU到中断被处理的那一段时间。如果中断延迟太大，则高速分层设备将会出错，因为在发生下一次中断之前无法处理它们的中断（通过从 CSR读数据）。后续的中断将覆盖CSR中的数据。

**进程间通信（IPC）**

IPC是进程间通信（Interprocess communication）的缩写。Linux支持信号、管道以及系统V IPC机制。这些机制使用的名称是沿用它们第一次出现在 Unix版本中的名称。

**内核线程（守护进程，daemon）**

内核线程是在内核模式中运行的进程。这些进程准确地说是内核的一部分，因为它们可以访问内核的所有数据结构和函数。但是它们被当作独立的进程来对待，可以中断和恢复执行。内核线程没有虚拟内存，但是可以访问其他内核所访问的相同的物理内存。

**逻辑文件系统（logical file system）**

该系统把数据块表示成文件和目录，这些文件和目录存储逻辑文件系统所特有的属性，但可以包括许可权限、访问和修改时间，以及其他一些计帐信息。

mm

Linux内核的内存管理程序子系统

**挂装点（mount point）**

它是虚拟文件系统中的目录，是挂装逻辑文件系统所在的地方。在挂装的逻辑文件系统上的所有文件都好象是挂装点的子目录。根文件系统是在 ’ / ’ 下挂装的。

Net

Linux内核的网络接口子系统

**逆推工程（reverse engineering）**

从源代码中抽取高级模型的过程。

sched

Linux内核的进程调度程序子系统。

**软件系统结构（software architecture）**

它是系统的结构，由软件部件、这些部件的外部可视属性以及它们之间的关系所组成。

**系统V（system V）**

一个Unix版本，由AT&T的贝尔实验室开发。

Unix

一种操作系统，最早是在1970年由贝尔实验室开发的。

# 附录B 参考文献

Bass 1998

Bass、Len、Clements、Paul、Kazman及Rick所著：Software Architecture in Practice, ISBN 0-201-19930-0, Addison Wesley, 1998

Beck 1996

Beck等所著：Linux Kernel Internals, Addison Wesley, 1996

Bowman 1998

Bowman, I.:“Conceptual Architecture of the Linux Kernel”,
http://www.grad.math.uwaterloo.ca/~itbowman/CS 746G/al/,1998

Chi 1994

Chi, E.:“Introduction and History of Linux”,
http://lenti.med.umn.edu/~chi/technolog.html,1994

CS746G Bibliography

http://plg.uwaterloo.ca/~holt/cs/746/98/linuxbib.html

Holt 1997

Holt, R.:“Portable Bookshelf Tools”,
http://www.turing.toronto.edu/homes/holt/pbs/tools.html

Knowlton 1965

Knowlton.K.C.:“A Fast Storage Allocator”, Communications of the ACM, vol.8, 第 623页～第625页, Oct.1965

Kruchten 1995

Kruchten, P.B.:The 4+1 Views Model of Architecture, IEEE Software, Nov 95, 第42页～第50页

LDP

Linux Documentation Project:http://sunsite.unc.edu/mdw/linux.html

Mancoridis Slides

Mancoridis, S.:MCS680 Slides, Drexel University

Muller 1993

Muller、Hausi A.、Mehmet、O.A.、Tilley、S.R.以及Uhl、J.S.所著：“A Reverse Engineering Approach to Subsystem Identification”, Software Maintenance and Practice, vol.5, 181~204, 1993

Murphy 1995

Murphy, G.C、Notkin、D.以及Sullivan K.所著：“Software Reflexion Models:Bridging the Gap between Source and High-Level Models”, Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (FSE’95)

PBS
我们所抽取的Linux描述：http://plg.uwaterloo.ca/~itbowman/pbs/.
Rusling 1997
Rusling,D.A:The Linux Kernel, ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/,1997
Siddiqi 1998
Siddiqi, S.:“A Conceptual Architecture for the Linux Kernel”, http://se.math.uwaterloo.ca/~s4siddiq/CS746G/LA.html, 1998
Tanenbaum 1992
Tanenbaum, A.S.:Modern Operating Systems, Englewood Cliffs, NJ:Prentice-Hall, 1992
Tanuan 1998
Tanuan, M.:“An Introduction to the Linux Operating System Architecture”, http://www.grad.math.uwaterloo.ca/~mctanuan/cs746g/LinuxCA.html, 1998
Tzerpos 1996
Tzerpos, V.和Holt,R.所著：“A Hybrid Process for Recovering Software Architecture”, http://www.turing.toronto.edu/homes/vtzer/papers/hybrid.ps, 1996