分类号 _____ 密级 _____

UDC[注1] _____

学 位 论 文

Linux 内核中一种高精度定时器的设计与实现

（题名和副题名）

周 鹏

（作者姓名）

指导教师姓名 周明天 教 授

电子科技大学 成 都

（职务、职称、学位、单位名称及地址）

申请专业学位级别 硕士 专业名称 计算机软件与理论

论文提交日期 2006.5 论文答辩日期 2006.6

学位授予单位和日期 电子科技大学

答辩委员会主席 朱 滴 新

评阅人 罗克露 许 毅

2006 年 6 月 2 日

注 1：注明《国际十进分类法 UDC》的类号。

摘要

以各类多媒体服务为代表的新的电信服务快速增长的需求，使得各电信运营商之间的竞争越来越激烈，为帮助电信营运商在激烈的竞争中脱颖而出，Intel、IBM等业界巨头提出了“基于标准的模块化网络平台” 概念。与传统的专有化网络通信平台不同，基于标准的模块化的网络平台要求从下层硬件、操作系统，到中间件、上层应用软件的接口都基于开放的标准；符合开放标准平台的各部分以模块的形式组织在一起。

标准的制定使得网络平台的每个“模块”允许多个供应商的参与竞争，竞争机制导致成本降低；同时，模块化的方式大大提高了构建电信应用平台的速度。“基于标准的模块化网络通信平台”不仅可以大大降低平台的整体拥有成本，而且可以缩短应用系统进入市场的时间。

作为“基于标准的模块化网络平台”核心的电信级 Linux(Carrier Grade Linux, CGL)是由开源组织 OSDL(Open Source Development Lab)发起的、专门针对电信级服务的 Linux。电信级 Linux 在标准 Linux 的基础上，增加了一组为适应电信运营环境而设计的特性。某些电信应用对实时性有较高要求，普通 Linux 在实时性方面离电信平台的要求还存在一定的差距。为增强系统的软实时能力，OSDL 要求电信级 Linux 提供一种精度在 0.1 毫秒以上高精度定时器（high-resolution timer）。本文首先介绍 Linux 内核 2.6.10 中时钟与定时器的情况，然后详细阐述这种符合 POSIX1003.1b API 标准的高精度定时器在 CGL 内核中的设计与实现，最后总结该定时器的性能并得出结论。

关键词： Linux内核，时钟，定时器，Carrier Grade Linux

Abstract

The increasing growth of new services makes the competition among tele-operating corperations become more and more fierce. To support them survive in the competition, the industry leaders, such as Intel and IBM, initiate the concept of "standard-based modularized network platform" which differentiates from the proprietary platform. The new network platform requires the hardwares, operating systems, middlewares and the interfaces of application totally based on open standards. All parts of the platform are organized by modules.

Standardization enables every module in the platform could be supported by various providers and the competition among providers results in the cost effect, and also, the modularization greatly speeds up the construction of services supported by telecom. So the standard-based modularized network platform not only reduces the cost of operator but also shortens the time to market for the operator.

As the core of standard-based modularized network platform, CGL(Carrier Grade Linux), which focuses on telecom services, is initialized by open source organization OSDL(Open Source Development Lab). Based on generic release of Linux, CGL adds a series of features to meet the telecom grade needs. High-resolution timer is one of enhancements for soft real-time performance in generic Linux whose interfaces conform to the POSIX1003.1b API. This thesis firstly introduces the clock and timer in standard kernel 2.6.10, Secondly describes design and implementation of high-resolution timer, at last, summarizes the performance of high-resolution timer and draws a conclusion.

Keyword: Linux kernel, clock, timer, Carrier Grade Linux

缩略语

CGL	Carrier Grade Linux	电信级 Linux
OSDL	Open Source Development Lab	开放源代码开发实验室
SMP	Symmetric Multiple Processor	对称多处理器
UP	Unique Processor	单处理器
RAS	Reliability, Availability, Serviceability	可靠性、可用性、可服务性
LSB	Linux Standard Base	Linux 标准基础
RTC	Real Time Clock	实时时钟
TSC	Time Stamp Counter	时间戳计数器
PIT	Programmable Interval Timer	可编程间隔定时器
APIC	Advanced Programmable Interrupt Controller	高级可编程中断控制器
POSIX	Portable Operating System Interface for Computer Environment 计算机环境的可移植操作系统界面	

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：周鹏　日期：2006 年 6 月 2 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：周鹏　导师签名：

日期：2006 年 6 月 2 日

第一章 引言

1.1 课题背景与意义

传统的数据通信网络为满足电信级服务在可靠性、可用性、可服务性（RAS，Reliability, Availability, Serviceability）以及响应时间等各方面的特殊要求，将其服务建立在专有平台之上。这样的私有系统通常由特殊的硬件、操作系统、中间件、应用程序及接口组成。私有系统不仅在体系结构设计上缺乏自由性和灵活性，同时也造成系统的维护、扩展费用居高不下。今天，以各类多媒体服务为代表的新的电信服务需求快速增长，电信运营商之间的竞争愈发激烈。为了在竞争中脱颖而出，各电信运营商纷纷开始寻求更高效、更经济、更快速的整体解决方案。

对电信运营商而言，他们面临的主要挑战是：

1) 如何降低部署新的电信应用的投入成本；

2) 如何快速地部署新的电信应用，缩短进入市场的时间。

如何解决这两个问题将在很大程度上决定电信运营商的竞争力。传统的私有系统已不再是各运营商的首选。当前，电信级平台正在逐步由私有系统向基于业界统一标准的模块化网络平台过渡。

由众多业界巨头如IBM, HP, CA, Intel以及NEC发起创立的非盈利性组织开放源代码开发实验室（OSDL, Open Source Development Lab）于2003年提出了电信级Linux(CGL, Carrier Grade Linux)项目[1]。电信级Linux就是业界各巨头所倡导的基于标准的模块化网络平台的核心，其初衷就是为电信服务建立一个开放标准的操作系统平台，为标准的模块化网络通信平台提供有力的保障。

毫无疑问，一个具有电信级特性的Linux内核是这个基于标准的模块化网络平台的精华。虽然Linux系统以其精妙的设计、开放源代码等特点而备受业界开发人员青睐——尤其是在嵌入式和服务器系统领域，但作为标准的模块化网络平台的核心，面对电信级服务极高的RAS（Reliability, Availability, Serviceability）要求，Linux系统仍存在诸多有待加强的地方。

开放源代码实验室电信级Linux工作组为增强Linux在电信服务方面的能力做了大量工作，在开源社区不断推动相关项目的发展。作为电信级Linux工作组成员之一，Intel公司一直致力于电信级Linux规范的制定及项目开发、测试。

作者于 2004 年 8 月至 2005 年 7 月在 Intel 中国软件中心(上海)作实习生，有幸参与了电信级 Linux 的开发和测试。本课题来自作者当时在 Intel 中国软件中心(上海)的实习项目。

1.2 国内外研究现状

今天，电信级 Linux 工作组已经成长为包括全球 30 多个一流厂商的国际化团队，这些来自不同领域（包括硬件平台提供商，Linux 发行版本提供商，网络设备提供商等）的业界巨头对电信级 Linux 投入了大量资金和人力，同 Linux 开源社区中众多的开发人员一道积极推动、促进电信级 Linux 的快速健康发展。

为了更清楚地制定电信级 Linux 规范，工作小组把电信级规范划分成 7 个独立分支：可用性（Availability），集群（Clusters），服务性（Serviceability），性能（Performance），标准（Standards），硬件（Hardware）和 安全性（Security）。每个分支下包含众多与该主题相关的子项目；来自业界大公司及开源社区的开发人员密切配合，相互促进，推动着这些子项目的开发。

截至 2006 年初，开放源代码开发实验室电信级 Linux 工作组已经制定出 3 个版本的规范来定义电信级服务所要求的特性。开源社区里的众多项目有力地支持了电信级 Linux 的发展。

与此同时，各 Linux 发行版本提供商，如 SuSE Linux, RedHat Linux, Monta Vista Linux, Redflag Linux, 越来越积极地参与到电信级 Linux 的开发与实现中；按照电信级 Linux 规范需求，各操作系统提供商不断在它们的企业级版本中集成各种相关软件；并且通过在开放源代码开发实验室的注册认证证明发行版本符合电信级 Linux 规范的要求，能够为基于标准的模块化网络平台提供高效的软件平台保障，满足不断增长的电信需要。

1.3 课题研究内容

1.3.1 Linux 操作系统及内核

开放源代码是 Linux 系统最主要的特点，通过对 Linux 内核源代码的研究，理解 Linux 系统的体系结构，掌握内核开发的基本步骤和方法。

1.3.2 基于标准的模块化网络平台与电信级 Linux(CGL)

基于标准的模块化网络平台是下一代网络平台；电信级 Linux 在标准 Linux 发行版的基础上加入了一系列针对电信级服务的特性。

研究基于标准的模块化网络平台体系结构，理解“标准”和“模块化”对于开放网络平台的意义，同时清楚认识电信级 Linux 在网络平台中的核心地位。

1.3.3 CGL 中一种高精度定时器及符合 POSIX 规范的实时函数接口

为弥补 Linux 作为分时操作系统在实时性方面的不足，提高 Linux 的软实时能力，电信级 Linux 规范 3.0 中提出了一系列改进措施，其中明确提出为满足某些实时流媒体响应时间的需求，Linux 内核应该提供一种精度在 100 微秒以上的高精度定时器，同时提供一套符合 POSIX 规范的实时函数接口。

Linux 内核 2.6 中的内核定时器精度是 1 毫秒，尚不能满足电信级服务的需要。本课题设计实现了一种基于 x86 体系结构和 Linux2.6 内核的新型定时器和实时函数接口，其精度达到电信级 Linux 3.0 规范的要求。

1.4 论文的组织

论文第二章简要介绍 Linux 的基本情况和内核特点，列举当前主流的企业级 Linux 发行版本。

从第三章开始，引入电信级 Linux 的概念，详细介绍了电信级 Linux 的体系结构和特性以及目前电信级 Linux 的开发状况；同时阐述了高精度定时器在电信级服务中的应用背景。

第四章重点分析传统 Linux 2.6 内核中的时间子系统和定时器结构，在此基础上介绍了几种流行的时钟粒度细化方案，同时详细描述一种新型的内核定时器的设计与实现方法。

第五章给出基于 Linux 2.6 内核的高精度定时器测试结果并分析结论。

第六章总结全文并提出下一步工作方向。

第二章 Linux 系统及内核概述

2.1 Linux 系统

Linux 是类 Unix(Unix-like)操作系统大家族中的一名成员。从 20 世纪 90 年代末开始，Linux 这位相对较新的成员突然变得非常流行，并且跻身于那些有名的商用 Unix 操作系统之列，这些 Unix 系统包括 AT&T 公司发布的 SVR4(System V Release 4)，加州大学伯克利分校发布的 4.4 BSD，IBM 公司的 AIX，HP 公司的 HP—UNIX，Sun 公司的 Solaris 等。

1991 年，Linus Torvalds 开发出最初的 Linux，它是一个适用于 x86 微处理器的操作系统。现在，Linus 依然不遗余力地改进 Linux，使它保持与各种硬件平台同步发展，并协调世界上各地的上百名开发人员工作。目前，Linux 已经可以在很多平台上运行，包括 HP 公司的 Alpha，IBM 公司的 PowerPC 等。

Linux 最吸引人的一个优点在于它不是商业操作系统：它的源代码在 GNU 公共许可证下是开放的，任何人都可以获得源代码并研究它。从技术的角度来说，Linux 是一个真正的 Unix 内核，但它不是一个完全的 Unix 操作系统，这是因为它不包含全部的 Unix 应用程序，诸如文件系统实用程序、窗口系统、文本编辑程序等等。不过，因为以上大部分应用程序都可以在 GNU 许可证下免费获得，因此，可以把它们安装在任何一个 Linux 支持的文件系统中。

2.2 Linux 内核

Linux 用途广泛，包含的东西也名目繁多。Linux 系统的基础是内核、C 库、编译器和系统的基本工具，如登陆程序和 shell。Linux 系统也支持现代的 X Window 系统，这样就可以使用完整的图形用户桌面环境，如 GNOME。可以在 Linux 上使用的商业和自由软件数以千计。这里需要特别说明的是，本文中使用"Linux"这个词时，通常是指 Linux 内核。在容易引起混淆的地方，论文会具体说明 Linux 到底是指整个系统还是内核，一般情况下，Linux 这个词主要是指内核。
是指整个系统还是内核，一般情况下，Linux 这个词主要是指内核。

2.2.1 Linux 内核功能划分

内核是操作系统的内在核心。操作系统的其他部分必须依靠内核提供服务；通常一个内核由负责响应中断的中断服务程序，负责管理多个进程分享处理器时间的调度程序，负责管理进程地址空间的内存管理程序和网络、进程间通信等系统服务程序共同组成。对于提供保护机制的现代操作系统来说，内核独立于普通应用程序，它一般处于系统态，拥有受保护的内核空间和访问硬件设备的所有权限。这种系统态和被保护起来的内存空间统称为内核空间。根据内核完成任务的不同，可将内核功能分成如下几部分[2]：（见图 2-1， Linux 内核功能划分）

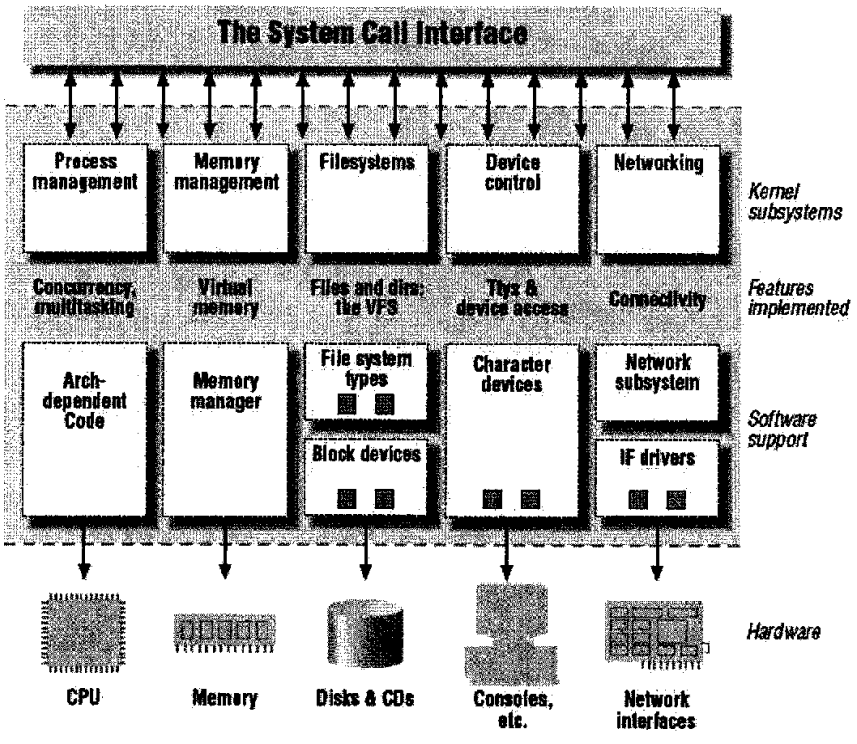


图 2-1 Linux 内核功能划分

1) 进程管理

进程管理功能负责创建和销毁进程，并处理它们和外部世界直接的连接（输入输出）。不同进程之间的通信——通过信号，管道或进程间通信原语是整个系统的基本功能，因此也由内核处理。除此之外，控制进程如何共享 CPU 的调度器也是进程管理的一部分。概括来说，内核进程管理活动就是在单个或多个 CPU 上实现多个进程的抽象。

2) 内存管理

内存是计算机主要资源之一，用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源之上为每个进程都创建一个虚拟地址空间。内核的不同部分在和内存管理子系统交互时使用一组函数调用，包括简单的 malloc/free 函数对以及其他一些复杂的函数。

3) 文件系统

Unix 系统在很大程度上依赖于文件系统的概念，Unix 中的每个对象几乎都可以当作文件来看待。内核在没有结构的硬件上构造结构化的文件系统，而文件抽象在整个系统中广泛使用。另外，Linux 支持多种文件系统类型，也就是在物理介质上组织数据的不同方式。例如，磁盘可以格式化为符合 Linux 标准的 ext3 文件系统，也可以格式化为常用的 FAT 文件系统或其他种类。

4) 设备控制

几乎每一个系统操作最终都会映射到物理设备上。除了处理器，内存以及其他很有限的几个对象外，所有设备控制操作都由与被控制设备相关的代码来完成，这段代码就叫做驱动程序。内核必须为系统中的每件外设嵌入相应的驱动程序，这包括硬盘驱动器、键盘等。

5) 网络功能

网络功能也必须由操作系统来管理，因为大部分网络操作和具体进程无关：数据包的传入是异步事件。在某个进程处理这些数据包之前必须收集、标识和分发这些数据包。系统负责在应用程序和网络接口之间传递数据包，并根据网络活动控制程序的执行。同时，所有的路由和地址解析问题也都由内核处理。

相应地，应用程序在用户空间执行。它们只能看到允许它们使用的部分系统资源，并且不能使用某些特定的系统功能，不能直接访问硬件。应用程序通过系统调用和内核进行通信。应用程序通常调用库函数——比如说 C 库函数——然后再由库函数通过系统调用让内核完成各种不同的任务。许多库函数提供的功能并没有单独的系统调用可以代替，在那些较为复杂的函数中，调用内核的操作通常只是整个工作的一个步骤而已。比如 printf()函数，它提供了数据的缓存和格式化操作，也只是在执行的末期通过 write()系统调用把处理后的最终数据写在终端上而已。不过，也有一些库函数和系统调用就是一一对应的关系，比如 open()库函数除了调用 open()系统调用，其他什么也不做。甚至还有一些 C 库函数，像 strcpy()，根本不需要调用系统级操作。当一个应用程序请求执行一条系统调用，我们称内核正在代其执行，在这种情况下，应用程序被称为通过系统调用在内核空间中运

行，而内核被称为运行在进程上下文中。这种交互关系——应用程序通过系统调用陷入内核——是应用程序完成其工作的基本行为方式，如图 2-2 所示。

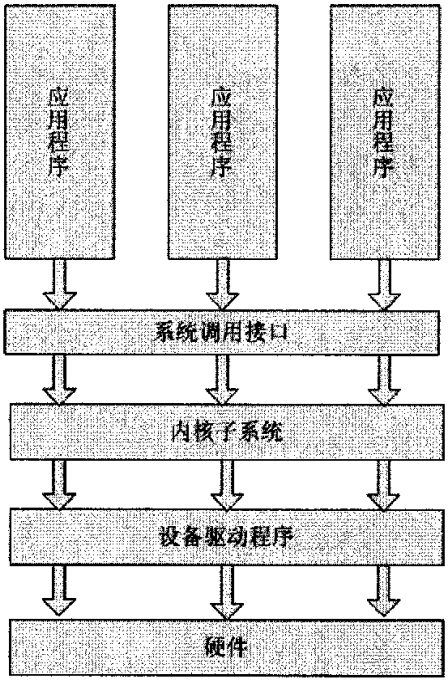


图 2-2 应用程序、内核和硬件的关系

2.2.2 Linux 内核和传统 Unix 内核比较

由于所有类 Unix 内核都是同宗同源，并且提供相同的 API，现代的 Unix 内核存在许多设计上的相似之处。Unix 内核几乎毫无例外的都是一个不可分割的静态可执行文件，即它们必须以完整、单独的可执行块的形式在一个单独的地址空间中运行。Unix 内核几乎都是需要硬件系统提供页机制管理内存。这种页机制可以加强内存空间的保护并保证每个进程都可以运行于不同的虚拟地址空间上。

当 Linus 和其他内核开发者设计 Linux 内核时，他们并没有完全彻底地与 Unix 诀别。他们充分地认识到，不能忽视 Unix 的底蕴（特别是 Unix 的 API）。而由于 Linux 并没有基于某种特别的 Unix，Linus 和他的伙伴们对每个特定的问题都可以选择已知最理想的解决方案——在有些时候，当然也有新的创新方案。以下是 Linux 内核和 Unix 各种变体的内核特点所作的分析比较[3]：

1) Linux 支持动态加载内核模块。尽管 Linux 内核也是整体式结构，可是允许在需要的时候动态地卸除和加载部分内核代码，尤其是设备驱动代码。

2) Linux 支持对称多处理（SMP， Symmetric Multiple Processing）机制，尽管许多 Unix 的变体也支持对称多处理机制，但传统的 Unix 并不支持这种机制。
3) Linux 内核可以抢占（Preemptive）。与传统的 Unix 不同，Linux 内核允许在内核中优先级高的任务优先执行。在其他各种 Unix 产品中，只是 Solaris 和 IRIX 支持抢占。
4) Linux 内核并不区分线程和其他一般进程。对于内核来说，所有的进程都一样。
5) Linux 体现了“自由”这个词的精髓。现在的 Linux 特性集就是 Linux 开放源代码模型自由发展的结果。如果一个特性没有任何价值或者创意很差，没有任何人会被迫去实现它。相反，在 Linux 的发展过程中已经形成了一种值得称赞的务实态度：任何改变都要针对现实中确实存在的问题，经过完善的设计并有正确简洁的实现。于是，许多其他现代 Unix 系统包含的特性，如内核换页机制，都被毫不迟疑的引入到 Linux 中。

2.2.3 Linux 内核版本

Linux 内核有两种：稳定的内核和处于开发中的内核。稳定的内核具有工业级的强度，可以广泛地应用和部署。新推出的稳定内核大部分都只是修改一些以前的 bug 或者加入一些新的设备驱动程序。相反地，处于开发中的内核里有很多东西变化得都很快。而且由于开发者不断实验新的解决方案，内核常常发生很大的变化。

Linux 通过一个简单的命名机制来区分稳定的和处于开发中的内核[4]。这种机制使用三个“.”分隔的数字来代表不同的内核。第一个数字是主版本号，第二个数字是从版本号，第三个数字是修订版本号。从版本号可以反映出该内核是一个稳定版本还是一个处于开发中的版本：该数字如果是偶数，那么此内核就是稳定的；如果是奇数，那么它就是开发版。举例来说，版本号 2.6.0 的内核就是一个稳定版。这个内核的主版本号是 2，从版本号是 6，修订版本号是 0。

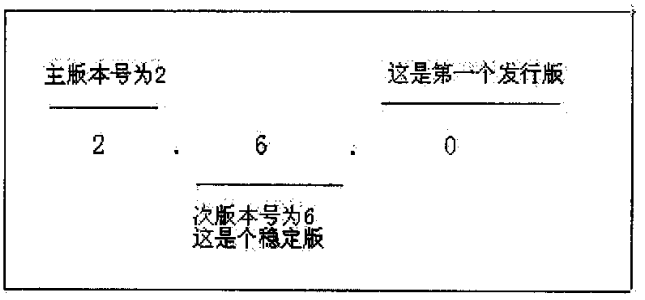


图 2-3 内核版本号的命名规则

处于开发中的内核一般要经历几个阶段。最开始，内核开发者们开始实验新的特性，这时候出现错误和混乱是在所难免的。经过一段时间，系统渐渐成熟，直到一个新的特性被宣布通过审定。这时就不再允许加入新的特性了。而对已有特性所进行的后续工作会继续进行，当这个新内核确实被认为是稳定下来以后，就开始审定代码。这以后，就只允许再向其中加入修改 bug 的代码了。在经过一个短暂的准备期，这个内核会作为一个新的稳定版推出。如 2.5 稳定下来后被作为 2.6 发布。

最新的 Linux 内核代码可以从 http://www.kernel.org 处获得。内核源代码通常安装在目录/usr/src/linux 下。

2.2.4 Linux 内核开发的主要特点

相对于用户空间的应用程序开发，内核开发有很大的不同。内核开发与应用程序开发之间的差异给开发内核带来了特别的挑战，但这并不意味着开发内核一定比开发应用程序难很多。

这种差异使内核看起来成了一个性格迥异的猛兽。一些常用的准则被颠覆了，而又必须建立许多全新的准则。尽管有许多差异一目了然，但还是有一些差异晦暗不明。最重要的差异包括以下几种：

1) 内核编程时不能访问 C 函数库；

与用户空间应用程序不同，内核不能链接使用标准 C 函数库（其他库也不行）。造成这种情况的原因有许多，其中就包括先有鸡还是先有蛋这个驳论。不过主要的原因还是在于速度和大小。对内核来说，完整的 C 库太大了，即使从中抽取一个合适的子集，大小和效率都不能接受。

幸运地是，大部分常用的 C 库函数在内核中都已经得到实现。比如说操作字符串的函数组就位于 lib/string.c 文件中。只要包含头文件<Linux/string.h>，

就可以使用它们。注意，这里所说的头文件是指包含在内核源代码树中的内核头文件。内核代码中不能包含外部头文件，因为它们无法调用外部的库函数。

2) 内核编程时必须使用 GNU C;

和所有 Unix 内核一样，Linux 内核是用 C 语言编写的。但 Linux 内核并不完全符合 ANSI C 标准。实际上，内核开发人员总是要用到 GCC 提供的许多语言扩展部分。

内核开发人员使用的 C 语法覆盖了 ISO C95 标准和 GNU C 扩展标准。这其中的种种变化把 Linux 推向了 GCC 的怀抱。尽管目前出现的一些新编译器如 Intel 的 ICC 已经支持了足够多的 GCC 扩展特性，完全可以用来编译 Linux 内核。下面就例举内核中使用的 GNU C 语言的扩展部分：

a) 内联（inline）函数

GNU 的 C 编译器支持内联函数。内联函数会在它所调用的地方展开。这么做可以消除函数调用和返回带来的开销（寄存器存储和恢复）；而且，由于编译器会把调用函数的代码和函数本身放在一起优化，所以内联函数也有进一步优化代码的可能。不过内联函数是有代价的，代码会变长，这意味着占用更多的内存空间或占用更多的指令缓存。所以，Linux 内核开发人员通常会把那些对时间要求比较高，而本身长度又比较短的函数定义成内联函数。

定义一个内联函数时，需要使用 static 作为关键词，并且用 inline 限定它，比如：

static inline void dog(unsigned long tail_size)

内联函数必须在使用之前就定义好，否则它就没法展开。

b) 内联汇编

GCC 编译器支持在 C 函数中嵌入汇编指令。当然，在内核编程时，只有知道对应的体系结构，才能使用这个功能。Linux 的内核混合使用了 C 和汇编语言。在靠近系统结构的底层或对执行时间要求严格的地方，一般使用的是汇编语言。而内核其他大部分代码是由 C 语言实现的。

c) 分支声明

对于条件选择语句，GCC 内建了一条用于优化的指令，在一个条件经常出现或者该条件很少出现的时候，编译器可以根据这条指令对条件分支选择进行优化。内核把这条指令封装成了宏，比如 likely()和 unlikely():

下面是一个条件选择语句：

```
if (foo) {
    /*...*/;
}
```

如果想要把这个选择标记成绝少发生的分支：

```
/*我们认为 foo 绝大多数情况下都是 false*/
if (unlikely(foo)) {
    /*...*/;
}
```

相反，如果要把一个分支标记成通常为 true 的选择：

```
/*认为 foo 绝大多数情况下都是 true*/
if (likely(foo)) {
    /*...*/;
}
```

3) 内核编程时缺乏像用户空间那样的内存保护机制；

如果一个用户程序试图进行一次非法的内存访问，内核会发现这个错误并结束整个进程。要是内核非法访问了内存，后果就很难控制了。内核中发生的内存错误会导致 oops，这是内核中出现的最常见的一类错误。因此在内核中千万不能去访问非法的内存地址，引用空指针。此外，内核中的内存都不分页。

4) 内核编程时浮点数很难使用；

在用户空间的进程内进行浮点操作时，内核会完成从整数操作到浮点数操作的模式转换。在执行浮点指令时到底会做怎样的操作，根据不同的体系结构，内核会作出不同的选择。

和用户空间进程不同，内核并不能完美地支持浮点操作。在内核中使用浮点数时，除了要人工保护和恢复浮点寄存器，还有其他一些琐碎的事情要做。所以不要轻易在内核中使用浮点数。

5) 内核只有一个很小的定长栈；

用户空间的程序可以从栈上分配大量的空间来存放变量，甚至巨大的结构体或者是包含许多数据项的数组都没问题。因为用户空间的栈本身很大，而且还能动态增长。

内核栈既不大也不能动态增长；它很小，而且长度固定。32 位机器的内核栈

是 8KB，而 64 位机器的内核栈是 16KB。

6) 由于内核支持异步中断、抢占和对称多处理机制，因此必须时刻注意同步和并发；

内核很容易产生竞争条件。和单线程的用户空间程序不同，内核的许多特性都要求能够并发的访问共享数据，这就要求有同步机制保证不出现竞争条件，特别是：

a) Linux 内核支持多处理器并发处理。所以，如果没有适当的保护，在两个或两个以上的处理器上运行的代码很可能会同时访问共享资源。

b) 中断是异步到来的，不论内核当前在执行什么代码。也就是说，如果不加以适当的保护，中断完全有可能在代码访问共享资源时到来，这样，中断处理程序就可能访问同一资源。

c) Linux 内核是可以被抢占的。如果不加以适当的保护，内核中一段正在执行的代码可能会被另一段代码抢占，从而有可能导致几段代码同时访问相同的资源。

常用的解决竞争的办法是自旋锁(spin lock)和信号量(Semaphore)。

7) 要仔细考虑可移植性的重要性。

尽管用户空间的应用程序不太注意移植性问题，然而 Linux 却是一个可移植的操作系统，并且要一直保持这种特点。也就是说，大部分 C 代码应该与体系结构无关，在许多不同体系结构的计算机上都能够编译和执行。

2.2.5 编译内核

在编译内核之前，首先需要对内核进行配置。由于内核提供了数不胜数的功能，支持众多的硬件，因而有许多东西需要配置。每个内核配置选项通常有 2 至 3 个选项，分别是 yes, no 和 module。yes 表示将该功能编译入内核；no 则相反；module 表示以模块的形式实现该功能，模块能够动态的加入或移出内核。

内核提供了各种不同的工具来简化内核配置。最简单的一种是一个字符界面下的命令行工具：

```
make config
```

该工具会遍历所有配置项，要求用户选择，所以该过程往往要消耗很长的时间。

通常使用更方便的配置工具：

make menuconfig

或者，使用基于 X11 的图形工具：

make xconfig

这两种工具将所有配置项分门别类放置，比如“Processor Type and Features”和“Device Driver”分别代表与处理器和设备驱动相关的配置选项。下图 2-4 为“make menuconfig”的界面。

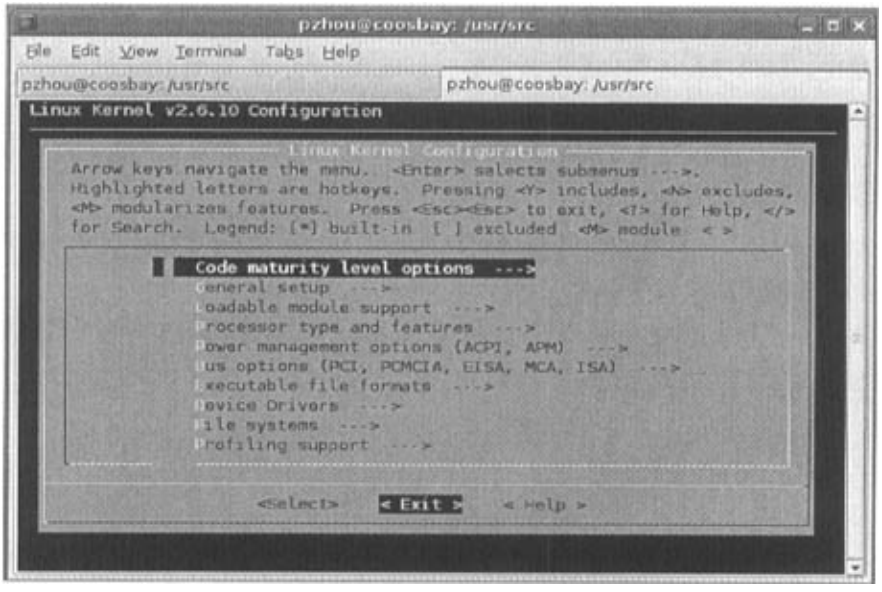


图 2-4 Make menuconfig 主界面

内核所有的配置信息都被存放在内核代码树根目录下的 .config 文件中。有经验的内核开发人员喜欢直接修改该文件。当修改该文件后，或者在用旧的配置文件编译内核之前，都应验证和更新配置文件：

make oldconfig

一旦配置好内核，接下来就应该编译它了：

make

编译时，内核会在内核代码树根目录下创建一个 System.map 文件。这是一份符号对照表，用以将内核符号和它们的起始地址对应起来。在对系统进行调试时，借助于 System.map，可以把内存地址翻译成容易理解的函数名或变量名。

2.3 几种流行的 Linux 发行版本简介

RedHat Linux: 红帽公司是开源社区最大的公司之一，它的企业级 Linux 版本为 RedHat Enterprise Linux，同时红帽公司还在开源社区发起了 Fedora 项目，Fedora Linux 完全免费，旨在推动 Linux 在个人桌面的应用，并为其企业级版本做前瞻性的技术研究。

SuSE Linux: SuSE 原本是一家德国公司，现已被 Novell 收购，在桌面和服务器市场上，SuSE Linux 的上升势头强劲。在欧洲占有大量市场份额。

Debian Linux：Debian Linux 是一个完全由开源社区推动的免费 Linux 发行版本，以稳定的性能和方便的软件包管理工具而闻名。Debian Linux 及其变种 Ubuntu 在个人桌面 Linux 市场上占有大多数份额。

MontaVista Linux: MontaVista 公司在嵌入式 Linux 方面具有较强实力，同时也是电信级 Linux 的积极参与者。

RedFlag Enterprise Linux: 红旗是我国 Linux 厂商的代表，在本地化和中文化方面红旗公司做了大量努力。

第三章 CGL---电信级的 Linux 平台

3.1 为什么需要 CGL

当前全球电信业正面临举步为艰的境地，而以各类多媒体服务为代表的新的电信服务需求的快速增长使得电信运营商之间的竞争更加激烈。对电信运营商而言他们面临的主要挑战是：

1) 如何降低部署新的电信应用的投入成本。

2) 如何快速的部署新的电信应用，缩短进入市场的时间。

如何解决这两个问题将在很大程度上决定电信运营商的竞争力。

3.1.1 传统的电信平台和基于标准的模块化网络平台

传统的电信网络平台往往基于特定目的应用而构建，从底层硬件（各类大、小型机，特殊的相关硬件）、操作系统到上层应用几乎都是绑定在一起的。与现在的 PC 服务器相比，这类平台不仅价格昂贵，而且由于每种平台往往都有各自的一整套操作维护系统，因此其维护代价也远远高于 PC 服务器。随着电信业的快速发展和竞争的日益激烈，新的电信业务不断出现，必然要求电信运营商能够快速地部署新的电信应用以满足不断增长的需求。

与此同时还要考虑整体投入成本的因素。而传统的电信网络平台的封闭、不灵活、供应商单一等特征决定了其在部署新的电信应用时往往技术难度较大，不能很快的部署成功，并且成本较高。

解决这些难题的方案就是 Intel、IBM 等提出的基于标准的模块化网络平台，图 3-1 显示了传统的电信网络平台和基于标准的模块化网络平台的差异。

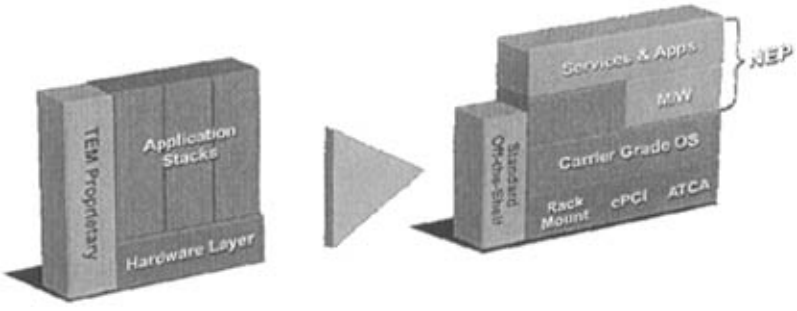


图 3-1 传统电信网络平台和基于标准的模块化网络平台

“标准”和“模块化”两个词体现了“基于标准的模块化网络平台”思想的精髓。

“标准”的意思是从底层的硬件到电信级的操作系统，以及各类中间件都是符合各类开放的标准，而不属于某个供应商所独有。比如底层硬件有 Compact PCI 标准、ATCA 标准，机架式服务器的各类标准等；电信级的操作系统应该满足电信运营需要的标准；中间件满足相关的各类标准，提供标准的应用编程接口，比如服务可用性相关的标准接口。

“模块化”的意思是组建整个平台的过程就象搭积木一样，从硬件、操作系统、中间件到各类应用都是组成系统的模块。而每一个模块都有各种不同的商业产品可供选择，而这些产品符合统一的标准。

标准的制定使得每个“模块”都允许多个供应商的参与竞争，而竞争机制导致成本降低，模块化的方式大大提高了构建电信应用平台的速度。由此可见基于标准的模块化网络平台能够很好地解决电信运营商面临的困难，即不仅可以大大降低平台的整体拥有成本，而且可以加速应用方案进入市场的时间。

3.1.2 电信级操作系统

从图 3-1 我们可以看到，在基于标准的模块化的网络平台中，电信级的操作系统是一个非常关键的组成部分。毫无疑问，一个统一的、开放的、跨平台的、并具备电信应用所要求的各种特性的软件环境将大大减少开发和部署各类电信级应用方案的费用和风险。Linux 作为一个正在快速发展的通用的服务器操作系统，它的跨平台性、开放性、和可扩展性，决定了它是潜在的承载电信运营级应用的最佳平台。Linux 作为电信级应用平台的主要好处是：

1) 良好的性能价格比，极大地节约了成本

和其他所有操作系统（Unix, Windows, Solaris）相比，Linux 操作系统的价格非常低廉，几乎可以忽略不计。而 Linux 的跨平台特性，使得以 Linux 为基础的整体解决方案不仅可以保护已有的电信设备投资，而且在新的设备投资中可以以 PC 服务器替代昂贵的传统的电信设备，因此可以大大降低总的投资成本。这对当前步履艰难的电信业而言无疑是最大的好处。

2) 开放的架构和标准的接口，加快了进入市场的速度

由于传统平台的封闭性，无论是硬件还是软件往往都受限于单个供应商。Linux 是一个开放的架构，提供了各种标准的接口，如 POSIX 标准，使得许多独立软件供应商都能够参与到从 Linux 发布、中间件到应用软件各个层次的开发当中。因此作为一个整体的解决方案，在系统的各个层次都有不同供应商可供选择，不再依赖于某个特定供应商。因此以 Linux 为基础的方案可以大大加快电信服务的部署，从而很快的进入市场。

3) 系统可以根据特定的垂直应用栈进行裁减和优化

Linux 社区倡导的是开放和自由的精神，不仅操作系统本身的源码是开放的，而且有大量的开放源码的项目。因此电信运营商可以根据特定应用的需要自己修改相应的代码，以满足特定的功能和性能需求，而无须等待特定的供应商帮助解决。

当然标准的 Linux 在可用性、可伸缩性、可管理性、实时性等方面和一个完全符合电信运营需求的电信级操作系统之间还存在一定的差距。比如就可用性而言，电信级平台往往要求具备 5 到 6 个 9（99.999%到 99.9999%）的高可用性，这意味着一年中只能有 5 分钟到 30 秒的停机时间。而标准的 Linux 内核从高可用性角度考虑，还有很多可以改进和强化的地方，比如对一些异常情况的处理，作为操作系统重要组成部分的各种驱动程序同样也需要特定的强化。再比如从实时性考虑，电信级操作系统平台要求有微秒级中断响应的时间，而标准的 Linux 内核目前还有一定的距离。因此从标准的 Linux 到满足电信运营需求的电信级操作系统，仍然存在大量的工作要做。然而 Linux 操作系统从诞生开始就有着一个天生的优越条件，那就是 Linux 拥有一个庞大的开发团队——Linux 社区，这一优越条件是 Linux 发展为电信级操作系统的坚实基础，而任何其他操作系统都不具备这样的条件。

基于标准的模块化电信平台的应用，为设备制造商和电信运营商提供了能力，使其能具备良好的成本效率，开发和部署新的服务和解决方案。而电信级 Linux 是基于标准的模块化电信网络平台中电信级操作系统的最佳选择。

3.2 CGL 平台的体系结构

简单地说，电信级的 Linux 就是一个标准的 Linux 的发布加上一组为适应电信运营环境的特殊需求而设计的增强特性。这些增强的特性主要包括以下三部分的内容：

1) 可靠性、可用性和可服务性（RAS）方面的特性
2) 增强的内核
3) 系统资源的监控和管理

所有这些增强特性都将通过开放源码项目的方式进行开发实现，Linux 版本的发行商通过开源社区，将各电信级特性的功能集成到自己的发行版本中，形成电信级的 Linux。

图 3-2 给出了电信级 Linux 平台的体系结构图：

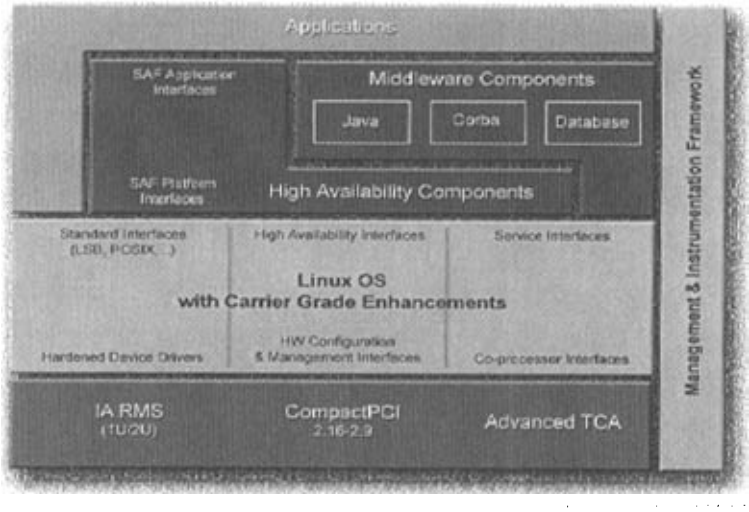


图 3-2 电信级 Linux 平台的体系结构

电信级 Linux 平台的体系结构特点分别如下：

1) 增强的操作系统（Linux OS with Carrier Grade Enhancements）

针对电信运营的需求而增强的 Linux 操作系统。对操作系统内核的增强主要包括可用性和可伸缩性两方面的内容，这样的增强还包括对硬件的访问接口，提供给用户层应用的接口，以及提供给开发和调试工具的接口。

2) 管理和分析框架（Management & Instrumentation Framework）

主要包括管理、调试和分析工具。

3) 高可用的硬件平台（High Availability Hardware Platform）

运营级的硬件平台包括了从无盘的 Blade 服务器到企业级的管理平台（比如大的存储设备）。硬件平台的可靠性对整个系统的可用性有着巨大的影响。硬件平台的任何一个组件（比如电源、风扇、各类适配卡等）都必须保证不会引起单点失效。硬件的失效必须及时标识出来，并且对失效硬件的替换不能中断系统的正常操作。

4) 高可用的组件（High Availability Components）

高可用的组件包括各类管理和监控的机制，比如对各种进程的状态监控，对系统各类资源参数的监控，管理集群的机制。

5) 中间件（Middleware Components）

对于管理、监视和控制电信运营系统的各类电信应用而言，它们往往依赖于数据库、对象管理和 JAVA 运行环境等各类通用目的的服务。这些中间件在移植到 Linux 的过程中依赖于内核提供的符合工业标准的应用编程接口（APIs），比如 POSIX 标准。

6) 应用（Application）

为了符合高可用性的要求，应用程序在维护系统服务和处理系统资源紧张带来的各种异常情况时必须足够的健壮。

由上可知，电信级 Linux 平台的架构鼓励各类中间件提供商向应用开发者提供用户模式下的软件服务。而应用给设备制造商和电信级 Linux 操作系统厂商提供了一个区别于其竞争对手的手段。而 Linux 操作系统本身只是整个解决方案中一个商业组件而已。

3.3 CGL 的特性

电信级 Linux 是在标准的 Linux 基础上，这些增强的工作既包括对 Linux 内核的强化，也包括对支持电信级 Linux 的诸多开放源码软件项目的增强。这些工作主要可以分为以下几类：

1) 可用性方面的强化
2) 安全性方面的强化
3) 操作、管理和维护的增强
4) 可伸缩性的增强
5) 实时性

6) 标准

7) 工具

下面逐一对这些电信级 Linux 的特性进行说明。

1) 可用性增强

可用性对于电信业来说一直是最关键的话题。一个系统的整体可用性往往和组成系统的各个组成部分的可用性密切相关，CGL 关注的是整个电信平台中电信级操作系统的可用性增强。比如，传统的 Linux 内核在出现一些特殊异常（panic）之后往往仅仅给出一个系统信息，而 CGL 将对这些情况进行逐个考量，尽可能保持系统的可用性。CGL 还对各类驱动程序进行增强，因为许多数据表明驱动程序失效引起的系统故障占的比率非常高。此外，CGL 还将增加对冗余硬件的热插拔、热切换支持以及高可用集群的支持。

2) 安全性增强

电信运营商的一些大客户，如一些大公司、政府组织和一些电子商务公司，对电信业务的安全性往往有严格的要求。随着电信服务器从传统的封闭系统向基于 IP 的系统的转化，在安全方面面临的隐患也越来越多。对计算机系统的安全主要考虑对如下三方面的保护：

a) 机密内容的保护：组织未授权用户对敏感信息的访问。

b) 完整性的保护：对数据改动过程中的完整性保护，当然也包括组织对未授权数据的删改。

c) 可用性的保护：这个词含义非常广泛，它可以定义为“任何服务在授权用户需要的时候都是可以访问的并且不会出现不合理的延迟”。在安全领域，它的意思为“保证合法用户能够访问授权的服务而不受攻击者的干扰”。

当然安全性问题还不仅限于这些方面，其他比如审计、访问控制等都是需要考虑的方面。CGL 工作组针对电信业在安全性方面的需求制定了相应的规范。比如：LSM（Linux Security Module），提供了一个通用的内核框架进行访问控制；IPsec 模块给 IPv4 提供保密和完整性机制；IKE（Internet Key Exchange）的支持；PKI CA 的支持等等。这些都将在电信级 Linux 中实现。

3) 操作、管理、维护的增强

一个好的管理平台对电信应用同样重要，比如在日常的软件维护和升级过程中，可以允许相应的电信服务的性能有适当的下降，但是服务不能中断。CGL 除了提供有效的手段对操作系统内核和系统资源进行检测外，还将提供应用程

序。

4) 性能和可伸缩性的增强

电信运营级应用程序对操作系统的可伸缩性和性能有很高的要求。可伸缩性包含两方面的含义：随着系统资源（比如处理器、网卡、磁盘等）增加，操作系统的性能（当然最终以电信应用的性能表现）应该有明显的增加；在系统资源不变的情况下，随着系统负载的增加，系统的性能应该保持稳定，而不至于有很大的滑坡。比如，许多传统的电信应用依赖于强大处理能力、多线程和大量定时器等。传统的 Linux 逻辑上可以支持大量的处理器，但其效率不高并可能带来性能下降；系统是否能够快速有效地生成大量的线程等。电信级 Linux 在许多方面上对系统可伸缩性进行了增强，比如：增强操作系统进程调度器的可伸缩性；对定时器机制的改善；链接绑定机制增强了网络带宽的可伸缩性；通过软 RAID 机制增强系统 IO 能力的可伸缩性。

5) 实时性

电信级操作系统平台要求有微秒级的中断响应的时间[5]。某些电信应用对实时性有一定的要求，比如软交换服务要求具备 80 毫秒的实时能力[6]。普通 Linux 在实时性方面和电信平台的要求之间还存在一定的差距[7]。电信级的 Linux 增强了实时性能力，具体的特性比如：提供高精度的定时器，增强操作系统的软实时能力，中断响应时间可以达到几十个微秒；提供可抢占的内核；提供 RAID 0 支持，在服务请求较多、数据传输较密集的应用中提高请求响应能力；提供应用的快速加载能力，从而提高响应能力。

6) 标准

标准是开放和跨平台的基础，CGL 工作组规定了电信级 Linux 必须满足一系列的标准。这些标准中有一些是作为一个跨平台的 Linux 操作系统所需满足的标准，另一些是作为电信级操作系统所必须满足的标准。主要有以下一些标准：

a) Linux Standard Base (LSB)：即电信级 Linux 必须和标准的 Linux 兼容，标准 Linux 能运行的应用程序在电信级 Linux 上也能运行。

b) POSIX 标准：包括 POSIX 定时器、信号、消息队列、信号量、事件日志等标准接口。POSIX 兼容性的支持大大提高了传统的电信应用向电信级 Linux 平台的移植效率，比如从 Solaris 到电信级 Linux 平台的移植。

c) IPv6 系列标准的支持：比如 IPSECv6、MIPv6 等协议和标准的支持。

这些协议和标准将支持今后的无线电信应用。

d) 高可用性应用编程接口：比如服务可用性论坛 (SAF)制定的一些标准，包括监视和控制电信平台各类硬件资源的硬件平台接口，以及监视和控制高可用电信应用的应用程序。电信级 Linux 将逐步满足所有这些标准。

7) 工具

电信级 Linux 将提供多种工具加强对操作系统内核和电信应用的调试支持，以便在系统或者应用出现异常时能够很快的定位、排除故障，恢复系统的可用性。这些工具包括：内核调试器，提供在启动或系统运行时的内核级中断调试，并且支持对称多处理器环境；支持内核配置，比如不同内核事件（中断和系统调用）的跟踪；增强了应用程序调试工具 GDB 的能力，支持调试多线程应用程序，并扩展了调试特性。

3.4 CGL 的发布机制与现状

3.4.1 发布机制

如上所述，电信级 Linux 操作系统是在标准的 Linux 操作系统基础上的增强。对于操作系统内核而言，增强的 CGL 特性将以各种补丁的方式整合到标准的 Linux 内核中，使之成为能够胜任电信运营需要的操作系统。各个 Linux 厂商如 RedHat、SUSE、MontaVista 等都可以发布自己的电信级的 Linux 版本。

此外，电信级的 Linux 还应该具备向后兼容性。即任何能够在标准 Linux 上运行的应用程序在不经改动的情况下都能够在电信级 Linux 平台上运行。当然，这些应用可能不能从电信级 Linux 平台提供的 CGL 特性中完全受益。

3.4.2 现状

目前在电信级 Linux 方面投入较大的公司是 RedHat、SuSE 和 MontaVista。

RedHat 的企业级发行版本 RedHat Enterprise Linux，根据用户选择的服务档次和技术支持的等级不同，RedHat Enterprise Linux 又分为 Advanced Server 和 Enterprise Server。目前的版本是 RedHat Enterprise Linux 4。

SuSE Linux 的企业级发行版本是 SuSE Linux Enterprise Server，当前的版本是 SuSE Linux Enterprise Server 9，相比 RedHat，SuSE Linux Enterprise Server 在电信级运用方面做了更多的工作，所支持的电信级特性也更多。

MontaVista 的部分股份是由 Intel 掌握，由于 Intel 在电信级 Linux 方面的积极态度，MontaVista 也在电信级 Linux 领域做了大量工作；其发行版本 MontaVista Enterprise Linux 集成了绝大多数电信级 Linux 特性，是一个比较完善的电信级 Linux 版本。目前在电信市场有较广泛的应用。

3.4.3 应用实例

电信级 Linux 的关键应用主要有以下三类：

1) 网关

网关是两个不同技术或管理域的桥接，如短信息服务网关。比如，媒体网关完成 TDM 电路交换网络与 IP 包交换网络的语音信息流转换。每一个网关在大量的接口上实时维持着众多的连接，并且要保证不丢帧，不丢包。

2) 信令服务器

信令服务器负责处理呼叫控制，会话控制和无线资源控制，如软交换服务器。它负责网络上的呼叫路由和维持呼叫状态。它接受发起呼叫的用户代理的呼叫请求并进行路由。信令服务器要求少于 80 毫秒的软实时能力[8]以及数以万计的并发连接的处理能力。鉴于快速交换和大容量连接管理的需要，每一个信令服务器是内容交换并对内存敏感。

3) 管理服务器

管理服务器处理传统的网络管理操作、业务管理和客户管理，如网管服务。它们提供了一些业务，如无线网络中的本地定位注册器、访问定位注册器和客户信息（比如客户是否获得认证等客户个人属性）。与信令服务器和网关这些应用相比，管理服务器对响应时间要求并不迫切，命令的数量相对也是比较少的。

3.5 结论

基于标准的模块化电信平台的应用，为设备制造商和电信运营商提供了能力，使其具备成本效率，即以更快的速度和更低的费用开发和部署新的服务和解决方案。Linux 的开放、跨平台特性是其成为电信级操作系统的一个很好的发展基础。通过对普通 Linux 在可靠性、性能、可伸缩性、实时性等方面的增强，电信级 Linux

及其服务环境日趋成熟，正逐步成为工业标准的电信级操作系统。事实上，国内很多电信设备制造商和运营商也逐步意识到以 Linux 为核心的模块化电信网络平台的优势，开始向这一方向转变。

第四章 基于Linux 2.6内核的一种高精度定时器的设计与实现

时间管理在内核中占有非常重要的地位。相对于事件驱动而言，内核中大量的函数都是基于时间驱动。很多计算机内部活动都是由定时测量（timing measurement）驱动的，对于用户来说，这常常是不可见的。例如，当你停止使用计算机的控制台一段时间后，显示器屏幕会自动关闭，这归功于定时器，它允许内核跟踪用户按键或者移动鼠标后到现在过了多少时间。如果你收到一个来自系统的警告信息，希望你删除一组不用的文件，这就是由于有一个程序能识别长时间未被访问的所有用户文件。为了进行这些操作，程序必须能从每个文件中检索到文件的最后访问时间，即时间戳（timestamp）。因此，这样的时间戳必须由内核自动的设置。更重要的是，定时机制与一些不可见的内核活动（如检查超时）一起来驱动使进程切换。

Linux内核必须完成两种主要的定时测量：

1) 保存当前的时间和日期，以便能通过time()、ftime()和gettimeofday()等系统调用把它们返回给用户程序，也可以由内核本身把当前时间作为文件和网络包的时间戳。
2) 维持定时器(timer)，这种机制能够提醒内核或用户程序某一时间间隔已经过去了，内核或用户程序收到提醒后去做之前指定的事情。

应该注意相对时间和绝对时间之间的差别。如果某个事件在5秒后被调度执行，那么系统所需要的不是绝对时间，而是相对时间（比如，相对现在起5秒后）；相反，如果要求管理当前日期和当前时间，则内核不但要计算流逝的时间而且还要计算绝对时间，所以这两种时间概念对内核时间管理来说都很重要。

另外的一个关注的焦点是内核动态定时器——一种用来推迟执行程序的工具。内核可以动态的创建和销毁动态定时器。

定时测量是由基于固定频率振荡器和计数器的几个硬件电路完成的。本章前部分描述定时基础的硬件设备，并给出Linux计时体系的总体概貌；接下来描述内核与时间相关的主要任务：实现CPU分时，更新系统时间和资源使用统计数以及维护软件定时器；再对内核中需要的不同长度的延迟进行描述和比较；最后提出符合电信级规范要求的新的内核定时器的设计思想和实现。

4.1 Linux 2.6 内核中的时间子系统

4.1.1 x86 上的硬时钟

在 x86 体系结构上，内核必须显式地与四种时钟交互：实时时钟（Real Time Clock, RTC），时间戳计数器（Time Stamp Counter, TSC）、可编程间隔定时器（Programmable Interval Timer, PIT）及 CPU 上的本地 APIC 定时器。前两种硬件设备允许内核跟踪当前时间；PIT 设备和 CPU 本地 APIC 定时器由内核编程指定，能以固定的可预见的频率发出中断。这样的周期性中断对于实现内核和用户程序使用的定时器都是至关重要。

4.1.1.1 实时时钟（RTC）

所有 x86 体系结构的机器都包含一个实时时钟(RTC, real time clock)，它独立于 CPU 和其他所有芯片。实时时钟是用来持久存放系统时间的设备。即使在系统关闭以后，实时时钟也可以依靠主板上的微型电池提供的电力保持系统计时。在 x86 体系结构中，实时时钟通常和 CMOS RAM 集成在一块芯片上（Motorola 145818 或其他相似的芯片上），而且实时时钟的运行和 BIOS 的保存设置都是依靠主板上同一电池供电。

实时时钟能在 IRQ8 上发出周期性的中断，频率在 2HZ—8192HZ 之间。可以对 RTC 进行编程以便按照一定的频率激活 IRQ 8 line。 通过对只读的字符设备 /dev/rtc 编程可以完成对实时时钟的操作。

Linux 内核通常只用实时时钟来获取日期和时间，通过 I/O 端口 0x70 和 0x71 可以访问实时时钟。当系统启动时，内核通过读取实时时钟来初始化墙上时间(wall time)，即当前实际时间。该时间存放在变量 xtime 中。

内核提供了宏 CMOS_READ 和 CMOS_WRITE 用于读取 CMOS 相关的设备，并用宏 RTC_PORT(x）简化了 RTC 的端口号：

include/asm-i386/mc146818rtc.h:

```
#define RTC_PORT(x)        (0x70 + (x))
#define CMOS_READ(addr)            \
        ({ outb_p((addr),RTC_PORT(0)); inb_p(RTC_PORT(1)); })
#define CMOS_WRITE(val, addr)      \
        ({outb_p((addr),RTC_PORT(0)); outb_p((val),RTC_PORT(1)); })
```

读取 RTC 时间参数可以用以下方式完成，

include/asm-i386/mach-default/mach_time.h:

```
sec = CMOS_READ(RTC_SECONDS);              /*读取当前时刻秒*/
min = CMOS_READ(RTC_MINUTES);              /*读取当前时刻分钟*/
hour = CMOS_READ(RTC_HOURS);               /*读取当前时刻小时*/
day = CMOS_READ(RTC_DAY_OF_MONTH);         /*读取当前时刻日期*/
mon = CMOS_READ(RTC_MONTH);                /*读取当前时刻月*/
year = CMOS_READ(RTC_YEAR);                /*读取当前时刻年*/
```

4.1.1.2 时间戳计数器(TSC)

如果需要度量非常短的时间，或者需要极高的时间精度，就可以使用与特定平台相关的资源，这种做法将时间精度的重要性凌驾于代码可移植性之上。

现代处理器中，由于缓存、指令调度、分支预测等技术的应用，在大部分的CPU 设计中，指令时序本质上是不可预测的，这导致依赖于指令周期的经验型性能描述方法不再适用。为解决这一问题，CPU 制造商引入了一种通过计算时钟周期来度量时间差的简便、可靠的办法，绝大多数现代处理器都包含一个随时钟周期不断递增的计算寄存器，这是时钟计数器完成高精度计时任务的唯一可靠途径。

基于不同的体系结构，该寄存器可能是 64 位也可能是 32 位。甚至在某些平台上，该寄存器根本就不存在，如果 CPU 缺少这样的寄存器而内核或应用程序又需要高精度的计时任务，则可能会由硬件设计者通过外部设备来实现。

幸运地是，从 pentium 开始的所有 x86 处理器都包含一个 64 位的时间戳计数器（TSC，timestamp counter）。这个 64 位的寄存器用于记录 CPU 时钟周期，它在每个时钟信号到来时加 1，例如，如果时钟节拍的频率是 400MHZ，那么时间戳计数器每 2.5 纳秒增加一次。内核空间和用户空间都可以读取它。

包含头文件<asm/msr.h>后，就可以 使用如下的宏：

```
rdtsc(low32, high32);     /*分别读取 TSC 高 32 位和低 32 位*/
rdtscl(low32);            /*读取 TSC 低 32 位*/
rdtscll(var64);           /*读取 TSC 整个 64 位*/
```

第一个宏原子性地把 64 位数值读到两个 32 位变量中；第二个宏只把 TSC 寄存器的第 32 位读进一个 32 位变量后；最后一个宏将 64 位值读入一个 long long 型变量。

在大多数常见的 TSC 应用中，读取计数器的低半部分就够了。

内核同时提供一个与体系结构无关的函数用来代替 rdtsc(),它是 get_cycles(),其

原型如下：

```
#include <Linux/timex.h>

cycles_t get_cycles(void);
```

各个平台的内核都可以使用这个函数，但在没有时钟周期计数寄存器的平台上它总是返回0。

为了利用时间戳计数器获取高精度的时间测量，内核必须在初始化系统的时候确定处理器时钟信号的频率。编译内核时并不声明时钟频率，所以同一内核映像可以在任何时钟频率的CPU上运行。

计算CPU实际频率的任务在系统初始化阶段完成。calibrate_tsc()函数承担着该任务，它通过计算在一个相对长时间间隔里所发生的时钟信号个数而得到CPU实际频率。calibrate_tsc()执行较长时间不会引起问题，因为只有在系统初始化期间才调用这个函数。

关于时间戳计数器，还有一点值得一提：在对称多处理系统（SMP）中，它们不会在多个处理器间保持同步。为了获取一致的时钟周期数，需要在查询该计数器的代码中禁止抢占。

4.1.1.3 可编程间隔定时器（PIT）

除了实时时钟和时间戳计数器，x86还提供了另一种时间测量设备——可编程间隔定时器（PIT， Programmable Interval Timer）。 可编程间隔定时器的作用类似于闹钟，所不同的是，这个设备不是通过振铃，而是通过一个特殊的中断，即时钟中断（timer interrupt）来通知内核又一个时间间隔过去了。与闹钟的另一个区别是，可编程间隔定时器永远以内核规定的频率不停地发出中断。可编程间隔定时器通常是一个使用0x40-0x43 I/O端口的8254 CMOS芯片。

通过对可编程间隔定时器编程，可以周期性地产生时钟中断。时钟中断的周期由内核中HZ的值设定。HZ是一个与体系结构有关的常数，定义在<Linux/param.h>或包含该文件的某特定平台目录下。对真实的硬件，已发布的Linux内核源代码为大多数平台定义的默认HZ值范围为50－1200，而软件仿真器的HZ值是24。在x86体系结构中，2.6内核默认定义HZ为1000（在2.4及其之前的早期内核版本中，x86平台上的HZ为100）。在include/asm-i386/param.h中对HZ定义如下：

```
#define HZ 1000        /*设定内核时钟频率为1000*/
```

作为一般性原则，即使知道对应平台上的确切HZ值，也不应在编程时依赖该

HZ 值。如果想改变系统时钟中断发生的频率，可以通过修改 HZ 值来实现。但是如果修改了头文件中的 HZ 值，则必须使用新的值重新编译内核以及所有模块。

可编程间隔定时器在系统启动过程中由 init_IRQ()进行初始化：

```
outb_p(0x34,PIT_MODE);          /* binary, mode 2, LSB/MSB, ch 0 */
udelay(10);
outb_p(LATCH & 0xff , PIT_CH0);  /* LSB */
udelay(10);
outb(LATCH >> 8 , PIT_CH0);      /* MSB */
```

4.1.1.4 理想的 HZ 值

自 Linux 问世以来，x86 体系结构中的 HZ 就设定为 100Hz，从内核 2.5 开始，中断频率被提高到 1000Hz。由于内核中众多子系统都必须依赖时钟中断，所以改变中断频率必然会对整个系统造成很大的冲击：提高 HZ 值以便在异步任务中获得更细的分辨率，但同时也会引入额外的时钟开销。下面我们将分析系统定时器高频率和低频率各自的优缺点。

提高时钟频率意味着时钟中断产生得更加频繁，所以中断处理程序也会更频繁地执行，这样给系统带来的好处如下：

1) 更高的时钟中断解析度可提高时间驱动事件的解析度。

2) 提高了时间驱动事情的准确度。

HZ＝100 的时钟中断执行粒度是 10 毫秒，即系统中周期事件最快为 10 毫秒运行一次，而不可能有更高的精度（注：这里所说的是计算机意义上的精度，而非科学意义上的精度；科学意义上的精度是统计反复性的量度；在计算机领域，精度是表示一个值的有效位个数。）。当 HZ＝1000 时，解析度就是 1 毫秒——比 HZ＝100 精度提高了 10 倍。

解析度提高的同时也提高了准确度。假定内核在某个随机时刻触发定时器，而这个定时器可能在任何时刻超时，但由于只有在时钟中断到来时内核才有可能去执行它，所以平均误差大约为半个时钟周期。如果时钟周期是 HZ＝100，那么时间平均在设定时刻的+/-5 毫秒内发生，所以平均误差为 5 毫秒。如果 HZ＝1000，那么平均误差可以降低到 0.5 毫秒——准确度提高了 10 倍。

更高的时钟中断频率和更高的准确度还会带来以下好处：

1) 内核定时器能够以更高频率和更高的准确度运行；

2) 依赖定时执行的系统调用如 poll()和 select()能够以更高的精度执行。

3) 对诸如资源消耗和系统运行时间等测量会有更精细的解析度。

4) 提高进程抢占的准确度。

对 poll()和 select()超时精度的提高会给系统性能带来极大的好处。提高精度可以大幅度提高系统性能。频繁使用上述两个系统调用的应用程序往往在等待时钟中断上浪费大量的时间，而事实上，定时器很可能早就超时了。

更高的准确度也使进程抢占受益，同时还会加快调度响应时间。中断处理程序负责减少当前进程的时间片（time slice）计数。HZ＝100 时，当时间片计数减至 0 且设置了 need_resched 标志位的话，内核便立刻重新运行调度程序。假定有一个正在运行的进程，它的时间片只剩下 2 毫秒了，此时如果调度程序要求另一个进程抢占该进程，则抢占行为不会在下一个时间中断到来前发生；也就是说，在这 2 毫秒内不可能进行抢占。在最坏的情况下，新进程可能要比要求的时刻晚 10 毫秒才能执行。更重要的是，由于耽误了抢占，对类似于填充音频缓冲区这样有严格时间要求的任务来说，结果是无法接受的。所以在电信级应用中，HZ 一般是 1000hz，这样即使在最坏的情况下，也能将调度延迟时间降低到 1 毫秒；平均情况下，延迟将降低到 0.5 毫秒。

当然，任何事情都有利有弊。时钟频率提高到 1000hz 也会带来一个大问题：过于频繁的时钟中断将导致沉重的系统负担。因为处理器必须花时间来执行时钟中断处理程序，所以越频繁的时钟中断，中断处理程序占用的处理器时间也越多。

但在现代计算机系统上，尤其是服务器平台上，时钟中断频率 1000hz 不会导致难以接受的负担。

4.1.1.5 jiffies

全局变量 jiffies 用来记录自系统启动以来产生的时钟中断总数。启动时内核将该变量初始化为 0，此后每次时钟中断产生时该变量都会加一。因为一秒内时钟中断的次数等于 HZ，所以 jiffies 一秒内增加的值是 HZ，系统运行时间以秒为单位计算，就等于 jiffies/HZ。

jiffies 定义于文件<linux/jiffies.h>中：

```
extern unsigned long volatile jiffies;
```

下面几个例子说明了内核中时间单位秒和 jiffies 的相互转化：

```
(seconds * HZ); /*将秒转化为 jiffies*/

(jiffies/HZ) ;    /*将 jiffies 转化为以秒为单位的时间*/
```

内核中经常需要用 jiffies 来设置一些将来的时间：

```
/*当前时刻*/
unsigned long time_stamp = jiffies;
/*从现在开始下一个时钟中断时刻*/
unsigned long next_tick = jiffies + 1;
/*从现在开始5秒*/
unsigned long later = jiffies + 5*HZ;
```

从2.6内核开始，内核中还增加了一个jiffies_64变量，这个64位的变量在文件<Linux/jiffies.h>中声明：

```
extern u64 jiffies_64;
```

之所以要增加jiffies_64，因为2.4及其以前的内核中HZ=100，32位的无符号长整型497天才会溢出；2.6内核中HZ=1000，则这个32位变量49.7天就会溢出。如果使用64位的变量，在可以预见的时间里，它都不会溢出。

从32位到64位的转化带来了兼容性的问题。由于现存的内核代码中仍有很多使用jiffies，而且大多数代码也只涉及低32位，所以原来的jiffies变量依然保留。内核将jiffies和新的jiffies_64结合在一起，图4-1反映了jiffies在jiffies_64上的布局。

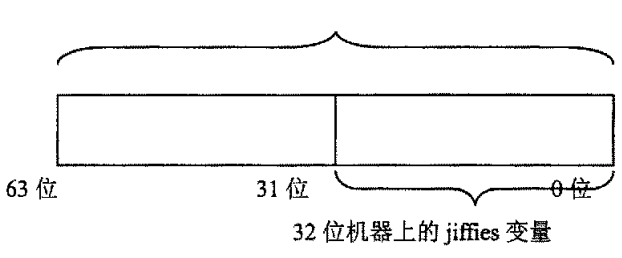


图4-1 jiffies在jiffies_64上的布局

访问jiffies的代码只会读取jiffies_64的低32位。如果需要读取整个64位，需要使用函数get_jiffies_64()，这是因为在32位机器上不能原子地的一次性访问64位中的两个32位数值。所以需要特殊的函数来保证原子性，该函数在读取jiffies时会利用xtime_lock锁对jiffies进行锁定。在64位机器中，jiffies和jiffies_64是一回事，既可以直接读取也可以通过函数get_jiffies_64()读取。

当jiffies变量超过其最大存放范围后就会发生溢出。32位的无符号长整型的最大取值是2^{32} -1，如果jiffies达到了最大值后还要继续增加的话，它的值会回绕（wraparound）到0。jiffies的回绕可能会给内核计时比较带来混乱。幸好内核提

供了四个宏来比较时间大小，这些宏能够正确地处理 jiffies 回绕的问题，在<linux/jiffies.h>中：

```
time_after(unknown, know);          /*时间晚于 know, 返回真*/
time_before(unknown, know);         /*时间早于 know, 返回真*/
time_after_eq(unknown, know);       /*时间不早于 know, 返回真*/
time_before_eq(unknown, know);      /*时间不晚于 know, 返回真*/
```

其中 unknown 是 jiffies, know 是需要对比的超时值。

4.1.1.6 CPU 本地定时器

x86 处理器的本地 APIC（local APIC）还提供了一种定时测量设备：CPU 本地定时器。

CPU 本地定时器是一种能够产生单次（one-shot）中断或周期性(interval)中断的设备，它类似于前面描述的可编程间隔定时器，不过 CPU 本地定时器还具有下面几个特点：

1) 本地 APIC 定时器计数器是 32 位，而可编程间隔定时器是 16 位。因此，可以对本地定时器编程来产生更高频率的中断。
2) 本地 APIC 定时器只把中断发送给本地的处理器，而可编程间隔定时器产生一个全局性中断，系统中任何一个 CPU 都可以对其处理。
3) 本地 APIC 定时器基于总线时钟信号。可以对该定时器每 1，2，4，8，16，32，64 或 128 总线时钟信号进行递减。相反，可编程间隔定时器有其自己内部的时钟振荡器。

4.1.2 Linux 计时系统结构

内核必须执行与定时相关的操作，例如，周期性地：

1) 更新系统启动以来所经过的时间。
2) 更新时间和日期
3) 确定当前进程在每个 CPU 上运行了多长时间，如果已经用完了所分配的时间片，则抢占它。
4) 更新系统资源使用统计数。
5) 检查每个软件定时器是否超时。

Linux 的计时体系结构是一组与时间流相关的内核数据结构和函数。在多处理器系统和单处理器系统之间，计时体系略有不同：

1) 单处理器系统上，所有的计时活动都是由可编程间隔定时器产生的中断触发。
2) 多处理器系统上，所有普通活动（如软件定时器的处理）都是由可编程间隔定时器产生的中断触发，而具体 CPU 的活动，如监控当前进程的运行时间，是由本地 APIC 定时器的中断触发。

实际上，以上两种情况的区别很模糊。因为新近的单处理器系统也拥有本地 APIC 和 I/O APIC，因此内核可以使用 SMP 的计时体系。

Linux 的计时体系结构还依赖于时间戳计数器（TSC）的可用性。内核使用了两个基本的计时函数：一个保持当前的最新时间，另一个计算在当前秒内走过的微秒数。关于如何通过时间戳计数器获得精确的微秒数，见本文 4.1.1.2 节的讨论。

4.1.2.1 时钟中断处理程序

内核计时体系的核心是时钟中断处理程序，以上描述的各种周期性活动均是在时钟中断处理程序中执行。分析时钟中断处理程序具有重要意义。

时钟中断处理程序可以分为两个部分：体系结构相关部分和体系结构无关部分。与体系结构相关的例程作为系统定时器的中断处理程序注册到内核中，以便在产生时钟中断时，它能够及时的运行。绝大多数时钟中断处理程序最低限度都要执行以下工作：

1) 获得 xtime_lock 锁，以便对访问 jiffies_64 和墙上时间 xtime 进行保护。
2) 在需要时应答或重新设置系统时钟。
3) 周期性地使用墙上时间更新实时时钟。
4) 调用体系结构无关的时钟例程 do_timer()。

具体的过程分析如下，在内核启动时，time_init()函数建立 IRQ0 对应的中断门。一旦这一步完成，IRQ0 的 irqaction 描述符的 handler 字段就包含 timer_interrupt()函数的地址。由于 IRQ0 主描述符的 status 域设置了 SA_INTERRUPT 标志，这个函数(timer_interrupt())以关中断开始运行，下图 4-2 为时钟中断处理程序的流程图：

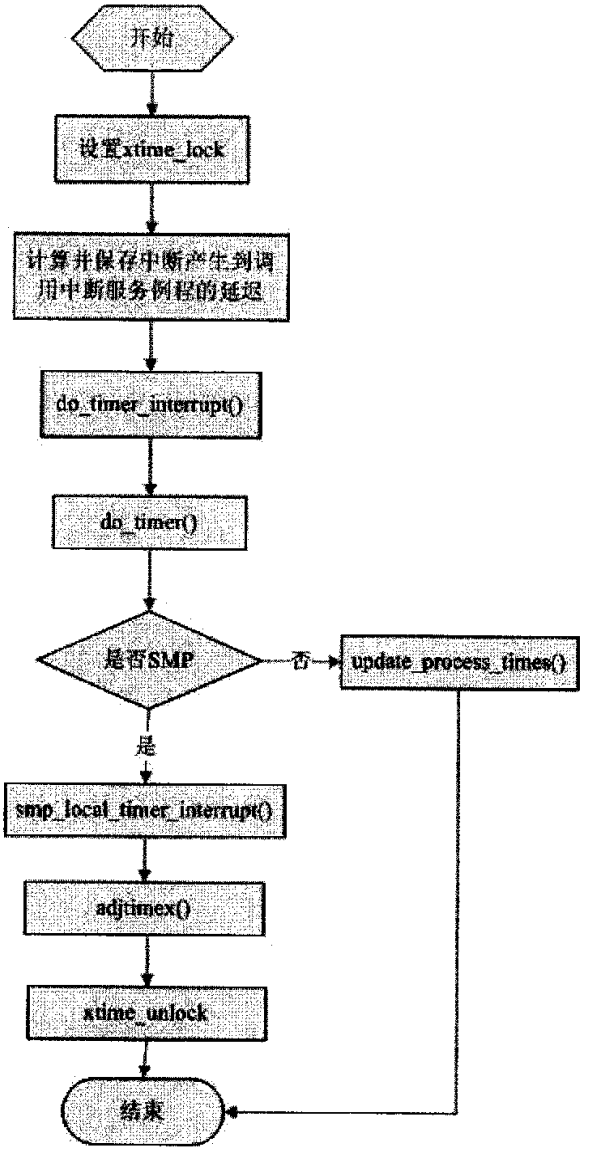


图 4-2 时钟中断处理程序的流程图

具体过程分析如下：

1) 为读取 xtime 变量设置 xtime_lock；

2) 如果 CPU 有 TSC 寄存器，就执行下列步骤：

a) 执行 rdtsc 汇编指令以在 last_tsc_low 变量中保存 TSC 寄存器的最低 32 位有效值。

b) 读 8254 芯片内部振荡器的状态，并计算时钟中断的发生与中断服务

程执行之间的延迟。

c) 在 delay_at_last_interrupt 变量中保存这个时延（以微秒为单位）。

3) 调用 do_timer_interrupt()；

do_timer_interrupt()执行下列操作：

a) 调用与体系结构无关的 do_timer()函数

b) 如果是单处理器，调用 update_process_time(user_mode(regs))；否则调用 CPU 本地定时器处理程序 smp_local_timer_interrupt(regs)；

c) 如果 adjtimex()系统调用被调用，那么它每 600 秒会调用一次 set_rtc_mmss()，也就是说，每 11 分钟调用一次实时时钟。这个特点有助于网络上的系统同步它们的时钟。

中断服务例程主要通过调用与体系结构无关的 do_timer()函数（kernel/timer.c 中）完成，do_timer()的主要任务是更新墙上时钟 xtime 并计算平均负载。具体过程如下：

1) 给 jiffies_64 变量增加 1（该操作即使在 32 位体系结构上也是原子性的，因为前面已经获得了 xtime_lock 锁）。

2) 更新墙上时间，保存在 xtime 中。

3) 计算平均负载。

因为上述工作分别都由单独的函数完成，所以 do_timer()看起来非常简单：

```
void do_timer(struct pt_regs *regs)
{
    jiffies_64++;
    update_times();
}
```

jiffies_64 全局变量中存放自系统启动以来的时钟中断数目。内核初始化期间把它初始化设置成 0，每当时钟中断发生时，jiffies_64 加一。Update_time()的作用就是更新墙上时间和计算系统负载。

从 timer_interrupt() 中可以看到，根据系统是否支持 SMP（对称多处理器），内核会选择不同的方式处理。多处理器的情况下会调用 smp_local_timer_interrupt()，用 local APIC 处理与进程相关的任务；单处理器则直接调用 update_process_times()。其实不论单处理器还是多处理器，内核都会调用 update_process_times()。

update_process_time()根据时钟中断产生的位置，对用户时间或系统时间进行

相应的时间更新。user_mode（）宏查询处理器寄存器 regs 的状态，如果时钟中断发生在用户空间，它返回 1；如果发生在内核空间，则返回 0。

```
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id(), system = user_tick ^ 1;
    update_one_process(p, user_tick, system, cpu);
    run_local_timers();
    scheduler_tick(user_tick, system);
}
```

run_local_timers()将运行所有超时的定时器，其方式是触发一个定时器的软中断：

```
void run_local_timers(void)
{
    raise_softirq(TIMER_SOFTIRQ);
}
```

4.1.3 CPU 的分时

Linux 是分时操作系统，每个进程的运行时间由内核所分配的时间片决定。时钟中断对于可运行（即处于 TASK_RUNNING 状态）进程之间共享 CPU 时间是必不可少的。通常内核给每个进程分配一个有限的时间片，如果时间片用尽时进程还没有完成，则 schedule()函数将重新选择一个新的进程投入运行。

进程描述符的 counter 字段表示进程在 CPU 上运行所剩下的节拍数。时间片总是节拍数的倍数。counter 的值在每个时钟中断处理程序中都由 update_process_times()更新，但单处理器和多处理器系统在调用方法上略有不同：在单处理器系统上直接由可编程间隔定时器中断处理函数调用，即在 do_timer_interrupt()中：

```
#ifndef CONFIG_SMP
/* SMP process accounting uses the local APIC timer */
    update_process_times(user_mode(regs));
#endif
```

在多处理器上则是由CPU本地时钟中断处理程序调用smp_local_timer_interrupt(regs)。

```
inline void smp_local_timer_interrupt(struct pt_regs * regs)
{
    int cpu = smp_processor_id();
    ...
    #ifdef CONFIG_SMP
    update_process_times(user_mode(regs));
    #endif
    ...
}
```

当counter值小于0时，进程描述符的need_resched字段被置为1。在这种情况下，内核会在恢复到用户态之前调用schedule()函数，选择处于TASK_RUNNING状态的其他进程运行。

4.1.4 更新时间和日期

当前实际时间（墙上时间）定义在文件kernel/timer.c中：

```
struct timespec xtime;
```

timespec数据结构定义在文件<linux/time.h>中：

```
struct timespec{
    time_t tv_sec;   /*1970年纪元到现在的秒数*/
    long tv_nsec;   /*上一秒到当前时刻的纳秒数*/
};
```

xtime.tv_sec以秒为单位，存放着自1970年1月1日凌晨0点以来所经过的时间；1970年1月1日成为纪元，多数Unix系统的墙上时间都是基于该纪元。xtime.tv_nsec记录着自上一秒开始经过的纳秒数，其取值范围为$0-10^9$-1。

系统初始化期间，函数time_init()设置时间和日期，它通过调用函数get_cmos_time()从实时时钟中读取时间和日期，并初始化xtime。以后每个时钟中断处理函数都会调用函数update_times()更新xtime。

读写xtime变量需要使用xtime_lock锁，更新xtime的步骤为：

```
/*获取xtime的seq锁*/
```

```
write_seqlock(&xtime_lock);
...更新 xtime...
/*释放 xtime 的 seq 锁*/
write_sequnlock(&xtime_lock);
```

update_times()中更新 xtime 的步骤：

```
ticks = jiffies - wall_jiffies;  /*计算最近一次更新 xtime 后产生的 jiffies*/
if (ticks) {
    wall_jiffies += ticks;
    update_wall_time(ticks);
}
```

ticks 记录最近一次更新 xtime 后所产生的节拍数。通常情况下 ticks 应该等于 1。但是时钟中断也可能丢失，在中断长时间被禁止的情况下，就会出现时钟中断丢失的情况（这样的情况不常见，内核中出现通常意味着 bug）。wall_jiffies 值随后被加上 ticks，这时候 wall_jiffies 就等于最新的墙上时间 jiffies。接着内核调用 update_wall_time()函数完成对 xtime 的更新。

从用户空间获取墙上时间的主要接口是 gettimeofday()，在有时间戳计数器的 x86 体系结构上，该函数返回的时间精度可以达到微秒级。同时，C 库也提供了一些函数获取墙上时间，比如 ftime()和 ctime()。

4.1.5 更新系统统计

内核在定时相关的任务中必须周期性地更新若干数据，用于：

1) 检查运行进程的 CPU 资源限制
2) 计算平均系统负载
3) 监管内核代码

4.1.5.1 更新运行进程的 CPU 资源

update_process_times()函数更新类型为 kernel_stat 的 kstat 变量中存放的一些内核统计数。

在单处理器系统上，该函数由可编程间隔定时器的时钟中断处理程序 do_timer() 调用，在多处理器系统上由本地时钟中断处理程序 smp_local_timer_interrupt()调用。

具体更新 CPU 统计资源由 update_one_process()完成，这些 CPU 统计数可以通

过系统调用 times()得到，或者通过在 bash 中运行实用程序 time 得到，例如：

```
coosbay:~#time sleep 5
real    0m5.004s
user    0m0.001s
sys     0m0.002s
```

time 程序返回的三行分别代表 sleep 5 的实际运行时间，CPU 用户运行时间和 CPU 的系统运行时间。如何区别 CPU 在用户态和内核态所花费的时间呢？update_process_times(int user_tick)的参数 user_tick 由 user_mode(regs)获得，若时钟中断发生在用户空间，返回 1，如果在内核空间，返回 0。update_process_times(int user_tick)中有 int system=user_tick ^1，异或操作使得 system 和 user_tick 两个变量中只有一个为 1。这样当 update_one_process()在对 CPU 系统时间和 CPU 用户时间进行更新时，只有一个变量发生了变化，另一个保持不变。

```
p->utime += user;
p->stime += system;
```

这样的统计方法意味着内核对进程进行时间计数是根据中断发生时处理器所处的模式 user_mode(regs)进行分类统计的，并把一个完整的 tick 全部算入进程的 CPU 系统时间或者 CPU 用户时间。但是事实上，进程在一个时钟中断周期内可能多次进入和退出内核，甚至在一个时钟中断周期内，该进程也不一定是唯一一个运行进程，所以这样的统计方式并不精密。到目前为止，内核还没有找到更好的统计方法，所以仍采用这种方式进行 CPU 资源的统计，这也是为什么需要把 HZ 从 100 提高到 1000 的原因之一。

4.1.5.2 更新平均系统负载

内核需要记录系统进行了多少 CPU 活动，这些统计数据可由各种管理实用程序使用，如 top 和 uptime。系统负载具体指系统最后 1 分钟、5 分钟，15 分钟的平均负载。单处理器系统上，负载为 0 意味没有活动的进程，swapper 进程 0 除外；负载 1 表示一个单独的进程 100%占用 CPU，负载大于 1 说明几个进程同时共享 CPU。

计算系统负载数据由 calc_load()函数完成，该函数被 update_times()调用。系统负载统计的对象包括处于 TASK_RUNNING 或 TASK_UNINTERRUPTIBLE 状态的进程。下面是一个用 uptime 获取系统负载统计的例子。

```
coosbay:~# uptime
```

14:52:24 up 1:53, 1 user, load average: 0.43, 0.23, 0.13

4.1.5.3 监管内核代码

Linux 内核包含一个代码监管器（profiler），内核开发人员用其发现内核在内核态的什么地方花费时间。监管器确定内核的“热点”（hot spot），即执行最频繁的内核代码片段。确定内核的“热点”十分重要，这有助于内核开发人员发现进一步需要优化的内核代码。

监管器基于蒙特卡罗算法：在每次时钟中断发生时，内核确定该中断是否发生在内核态；如果是，内核从堆栈取回发生前的 eip 寄存器值，并用这个值分析中断前内核正在做什么。最后，采样数据集聚在“热点”上。

Profile_tick(CPU_PROFILING, regs)函数为代码监管器采集数据。与更新 CPU 资源相似，这个函数在单处理器系统上由可间隔定时器的时钟中断处理程序 do_timer_interrupt()调用；在多处理器系统上，由本地时钟中断处理程序 smp_local_timer_interrupt()调用。

为了激活代码监管器，在内核启动时必须传递参数“profile=N”,这里 2^N 表示要监管代码段的大小。采集的数据可以从/proc/profile 文件中读取。可以通过修改这个文件来重置计数器；在多处理器系统上，修改这个文件还可以改变抽样频率。系统命令 readprofile 用来直接访问/proc/profile。

4.2 延迟及各种延迟实现方法的比较

内核除了使用定时器和下半部机制外还需要其他方法来推迟执行任务。推迟通常是等待硬件完成某些工作，而且等待的时间往往非常短。比如重新设置网卡需要花费 2 毫秒，所以在设定网卡后，驱动程序必须至少等待 2 毫秒才能继续运行。

内核提供了许多延迟方法处理不同长短的延迟要求。有些是在延迟任务时挂起处理器，防止处理器执行任何实际工作；另一些不会挂起处理器，所以不能确保被延迟的代码能够在指定的延迟时间运行。

4.2.1 长延迟

有时内核驱动程序需要延迟比较长的时间，即长于一个时钟滴答。实现长延迟有好几种途径。

4.2.1.1 忙等

如果想要把执行延迟若干个时钟滴答，或者对延迟的精度要求不高，最简单的实现是一个监视jiffies计数器的循环。实现方法如下(j1是延迟终止时的jiffies)：

```
while (time_before(jiffies, j1))
    cpu_relax();          /*忙等*/
```

对cpu_relax()的调用与处理器体系结构有关，在许多系统上，这个函数根本不会做任何事情。因为jiffies在内核中被声明为volatile型变量，所以每次C代码访问它时都会重新读取它，因此可以起到延迟的作用。但忙等待循环严重降低了系统性能，所以并不推荐使用。

4.2.1.2 调度

忙等待为系统增加了沉重的负担，因此有必要寻找更好的延迟技术。另一个长延迟的方法是在不需要CPU时主动释放CPU，即调用schedule()函数：

```
while (time_before(jiffies, j1)) {
    schedule();       /*重新调用进程*/
}
```

虽然调度比忙等提高了效率，但当前进程仍然在运行队列中，如果系统中只用一个可运行进程（即该进程），则该进程又会立即运行。换句话说，系统地负荷（运行进程的平均数）至少为1，且空闲(idle)任务（进程号为0，也称为swapper）从来不会运行。尽管这个问题看似不重要，但运行空闲进程可以减轻处理器负荷，降低处理器温度，增加处理器寿命（对笔记本电脑来说电池的影响更明显）。所以调度也不是十分理想的解决办法。

4.2.1.3 超时

总之，通过监视jiffies计数器实现延迟可以工作，但不理想。如果内核驱动程序使用等待队列来等待其他一些事件，而我们同时希望在特定时间段中运行，则可以使用wait_event_timeout或者wait_event_interruptible_timeout函数：

```
#include <linux/wait.h>
long wait_event_timeout( wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout( wait_queue_head_t q, condition,
                                       long timeout );
```

上述函数会在给定的等待队列上休眠，并在超时(用jiffies表示)到期时返回。

这里的 timeout 表示要等待的 jiffies 的值，而不是绝对时间。

4.2.2 短延迟

某些设备处理延迟非常短，最多涉及到几十个毫秒。在这种情况下，依赖时钟滴答显然不是正确的方法。

ndelay(), udelay()，mdelay()这几个内核函数可很好的完成短延迟任务。他们分别指定数量在纳秒，微秒和毫秒时间，原型如下：

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);          /*纳秒级的短延迟*/
void udelay(unsigned long usecs);          /*微秒级的短延迟*/
void mdelay(unsigned long msecs);          /*毫秒级的短延迟*/
```

这些函数的实现包含在<asm/delay.h>中，与具体体系架构有关。

需要提醒的是，这三个延迟函数均是忙等待函数，因而在延迟过程中无法运行其他任务。实现毫秒级或更长的延迟还有另一种方法，这种方法不涉及忙等待。<linux/delay.h>中声明了这些函数：

```
#include <linux/delay.h>
void msleep(unsigned int millisecs);       /*毫秒级的短睡眠*/
unsigned long msleep_interruptible(unsigned int millisecs);
/*毫秒级的、不可中断的短睡眠*/
void ssleep(unsigned int second);          /*秒级的睡眠*/
```

4.3 Linux 2.6 内核中的定时器

定时器是一种软件功能，允许在将来某个时刻调用指定的函数。超时（time-out）表示与定时器相关的时间间隔已经用完。内核和用户进程广泛地使用定时器。大多数设备驱动程序利用定时器检测异常情况——例如软盘驱动程序在软盘暂时不被访问后就关闭设备引擎，而并行打印机设备利用定时器检查错误的打印机情况。编程人员经常利用定时器在将来某一时刻强制执行特定的函数。

Linux 有两种类型的定时器，即动态定时器（dynamic timer）和间隔定时器（interval timer）。动态定时器由内核使用，而间隔定时器由用户进程在用户态使用。

本节主要讨论动态定时器在 Linux2.6 内核中的设计与实现，并给出间隔定时

器为用户空间提供的函数接口。

4.3.1 Linux 2.6 内核中的动态定时器的使用

内核提供了一组用来声明、注册和删除内核动态定时器的函数，下面列出一些基本的接口：

```
#include <linux/timer.h>
struct timer_list {
    struct list_head entry;                 /*定时器链表头*/
    unsigned long expires    ;              /*定时器到期时间*/
    spinlock_t lock;                        /*自旋锁*/
    unsigned long magic;
    void (*function)(unsigned long);        /*定时器发生时调用的函数*/
    unsigned long data;                     /*定时器调用函数的参数*/
    struct tvec_t_base_s *base;
};
static inline void init_timer(struct timer_list * timer);
static inline void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

function 字段包含定时器到期时执行函数的地址。data 字段指定传递给定时器函数的参数。正是由于有了 data 字段，可以定义一个单独的通用函数来处理多个设备驱动程序的定时问题，在 data 字段存放设备 ID 或其他有意义的数据，定时器函数可用这些数据区分不同的设备。

expires 字段指定时器到期时间，时间用节拍数表示，其值为系统启动以来所经过的节拍数。当 expires 的值小于或等于 jiffies 的值时，就说明定时器到期了。

list 字段包含双向循环链表的链接。总共有 512 个双向循环链表保存动态定时器。根据 expires 的值大小，每个定时器被插到其中一个链表中。

定时器的创建和使用过程如下：

1) 创建定时器时需要先定义它：

```
struct timer_list my_timer;
```

2) 初始化定时器：

```
init_timer(&my_timer);
```

```
my_timer.expires = jiffies+delay;      /*定时器超时时刻节拍数*/
my_timer.date=0;                       /*给定时器处理函数传入参数 0*/
my_timer.function=my_function;         /*定时器超时时调用的函数*/
```

3) 激活定时器：

```
add_timer(&my_timer);
```

至此，定时器就可以工作了。

修改或者删除定时器的函数如下：

```
mod_timer(&my_timer, jiffies+new_delay);
del_timer(&my_timer);
del_timer_sync(&my_timer);
```

4.3.2 实现定时器

内核在时钟中断发生后执行定时器，定时器作为软中断在下半部（bottom half）上下文中执行。时钟中断处理程序会执行 update_process_timers()函数，该函数随即调用 run_local_timers()触发软中断。

```
void run_local_timers(void)
{
    raise_softirq(TIMER_SOFTIRQ);
}
```

run_timer_softirq()函数处理软中断 TIMER_SOFTIRQ，从而在当前处理器上运行所有超时的定时器。

合适的数据结构对动态定时器性能影响很大。把所有定时器放在一个单独的链表中会降低系统的性能，因为每个时钟中断去遍历一个定时器长链表太费时。另一方面，维护一个排序的链表效率也不高，因为插入和删除操作也很费时。

Linux 2.6 内核把 expires 值划分成不同的大小，并允许动态定时器从大 expires 值的链表到小 expires 值的链表有效穿过。其主要数据结构是一个叫 tvecs 的数组，其元素指向五组链表，分别由 tv1,tv2,tv3,tv4,tv5 表示。图 4-2 列出了与动态定时器相关的链表组。

tv1 结构的类型为：

```
struct timer_vec_root{
        int index;
        struct list_head *vec[TVR_SIZE];
```

};

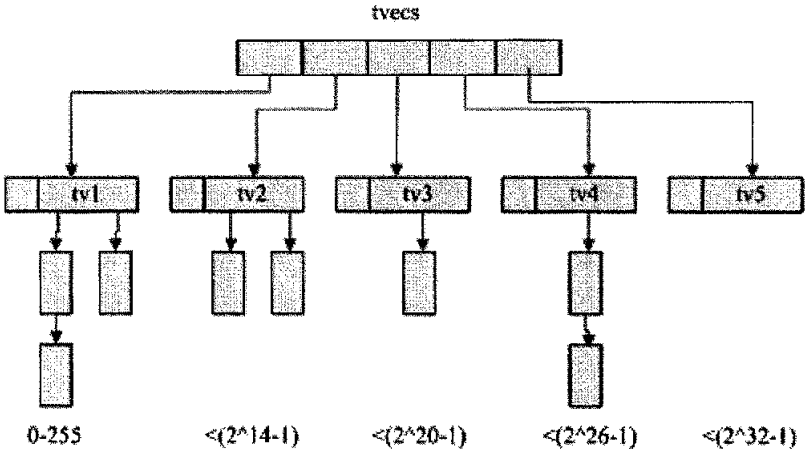


图 4-2 与动态定时器相关的链表组

TVR_SIZE 为 256，vec 数组包含未来 256 个节拍内将要到期的所有动态定时器。类型为 struct timer_vec 的 tv2,tv3,tv4 分别包含在紧接着到来的 2^{14}-1,2^{20}-1, 2^{26}-1 个节拍内到期的所有动态定时器。tv5 与前三组定时器数组略有不同，它可以包含任意多个定时器。

当定时器超时值接近时，定时器将随数组一起下移，tv2-->tv1, tv3-->tv2，以此类推。这样的分组设计方法可以在执行软中断的多数情况下，确保内核尽可能减少搜索超时定时器所带来的负担。因此定时器管理代码是十分高效的。

4.3.3 Linux 2.6 内核定时器竞争条件

定时器与当前执行代码是异步的，因此就有可能存在潜在的竞争条件。所以修改定时器的超时时间需要谨慎，应使用内核函数 mod_timer()来实现。

mod_timer(&my_timer, jiffies+new_delay);

如果需要提前停止定时器，则应使用 del_timer_sync()，而非 del_timer()。因为无法确定在删除定时器时，该定时器是否在其他处理器上运行。

4.4 电信级 Linux 实时性能的要求

新体系结构和新技术的出现，如下一代网络（NGN）、IP 多媒体服务、移动

网络，使得电信级应用市场面临着巨大的技术挑战。实时性是各种基于 IP 的新应用和新协议（如 VoIP, SIGTRAN, RTP）需要解决的问题，实时能力直接影响着这些应用和协议提供给终端用户的服务质量[10]。增强 Linux 在实时性方面的性能有助于将许多目前运行在其它实时操作系统上的应用迁移到 Linux，有助于 Linux 成为电信级的操作系统。

目前，业界对于软实时(soft real-time)和硬实时(hard real-time)的划分已经达成共识，电信级 Linux 仅提供软实时能力的支持[11]。本文中所提到的所有与实时相关的概念均指软实时。

电信级 Linux 3.0 规范对 Linux 的实时能力提出了很多改进的要求，其中包括：

1) 限定最大调度和中断延迟[12]
2) 支持 1 毫秒时钟中断频率
3) 支持高精度定时器
4) 支持 POSIX 实时特性

对于实时性较强的应用来说，及时响应和处理实时任务是非常重要的，时钟中断可以被看作是一个调度点。在该时刻，如果有更高优先级的实时任务正在等待，则当前运行的任务将被抢占。因此时钟中断的粒度是实时任务能否被及时调度的重要因素，也是影响实时能力的一个重要方面。

Linux 2.6 内核已经把 HZ 的默认值从 100 提高到 1000，时钟中断的粒度为 1 毫秒。满足了电信级 Linux 3.0 规范对时钟中断频率的要求。但根据电信级 Linux 3.0 规范中 performance 部分的要求，电信级 Linux 需要一种精度在 100 微秒及以上的定时器；并能让应用程序通过符合 POSIX1003 标准的接口管理使用这样的定时器。这两个需求的具体描述如下：

1) 提供一种单次（one-shot）定时精度达到 100 微秒及其以上的高精度定时器，并且能使应用程序使用和管理这种高精度定时器。
2) 提供一组符合 POSIX1003.1b 第 14 节（时钟和定时器部分）API 规范的应用程序接口[13]，这组 API 接口有 clock_gettime()，clock_settime()，clock_getres()，clock_nanosleep，timer_settime()，timer_gettime()，timer_getoverrun()，timer_create()。

目前 Linux 2.6 内核中的定时器尚不能满足电信级 Linux 中高精度定时器的要求。

通过之前对内核时间子系统和定时器的研究发现，影响定时器响应时间的原因主要有以下三点[14]：

1) 操作系统的时钟中断粒度。
2) 到期定时器服务程序的执行顺序。有些操作系统在处理定时器时采用LIFO(Last In First Out)策略，即最优先处理距离当前时间最近到期的定时器。
3) 执行定时器服务程序所花的时间。这是最不可预测的因素，有些中断服务程序的执行时间可能会导致长时间的中断延迟。

4.5 常见Linux时钟粒度细化方案的分析和比较

4.5.1 KURT Linux的细粒度时钟机制

KURT Linux由美国堪萨斯大学所开发。不同于硬实时/软实时应用，他们提出“严格（firm）”实时应用的概念，如一些多媒体应用和ATM网络应用。KURT是为这样一些应用设计的“严格”的实时系统，并且可以提供微秒级的定时精度。

在x86体系结构中，系统时钟可以达到的最高频率超过了1MHz，Linux通过对它编程，将时钟频率设定为1000HZ，即时钟中断间隔为1毫秒。对于实时操作系统而言，这种时钟粒度还不高。如何解决这个问题呢？最容易想到的莫过于提高时钟频率。然而简单的提高时钟频率意味着时钟中断的处理过程也越加频繁，占用过多的处理器时间，最后使得整个系统的有效利用率急剧下降，所以这不是个好办法。

KURT的设计者经过分析发现，时钟精度和定时器的频率有很大区别，时钟中断应该能够发生在任何一个微秒时刻，但并不是每个微秒都需要发生时钟中断。KURT修改了Linux内核时钟中断机制，改变了时钟中断的固定频率模式，通过重新设定使得时钟得以在微秒为单位的任何需要的时刻产生中断。

其具体实现方法是利用x86体系结构的CPU所提供的时间戳计数器TSC跟踪系统时间。显然，TSC的精度远远高于内核中jiffies计数精度，以到期TSC时标和系统当前TSC时标之差作为时钟发生芯片的精度，设置硬件时钟发生频率。因此，KURT增加了两个全局变量：jiffies_u和jiffies_intr，前者以微秒为单位记录在一个jiffy内的时间流逝，后者当jiffies_u大于一个tick时加1。KURT的研究者把这样的机制称之为UTIME。在具有UTIME的系统中，提高了系统的实时能力，保证了任务的响应时间。

4.5.2 RFRTOS 的细粒度时钟机制

RFRTOS——红旗 Linux 实时操作系统，是我国第一个基于 Linux 的实时操作系统，由中科院软件研究所自主研发设计[15][16][17]。在该系统中，通过中断线程化、互斥锁机制的改进、每次中断返回都判断是否需要重新调度等方式实现了内核的可抢占性，在此基础上，为了防止优先级反转，增加了优先级继承协议的支持。在时钟管理部分，细化了时钟粒度，并且支持对称多处理器。

RFRTOS 提供了与 Linux 内核时钟并行运行的一个具有精密时钟刻度的实时核心时钟，与原 Linux 内核时钟区别开发。这样不但提高了系统的稳定性和效率，而且独立的实时核心时钟易于维护改进。另外，RFRTOS 采用“堆”的数据结构来组织和管理系统中的实时定时器队列。

4.5.3 比较与评价

以上介绍了两种流行的 Linux 时钟细粒度化方案，这两种实现方案各有特点，比较如下：

KURT Linux 的 UTIME 提高 Linux 系统时钟精度的方法是通过将时钟芯片设置为单次触发状态(one-shot mode)，即每次给时钟芯片设置一个超时时间，然后到该超时事件发生时，在时钟中断处理程序中再次根据需要给时钟芯片设置一个超时时间。这种实现机制的优点是：系统在没有过大地增加系统开销的情况下实现了微秒级的定时粒度，提高了系统的响应。

缺点：UTIME 方案对内核改动较大，系统中很多活动都依赖于定时的时钟中断，取消定时的时钟中断对内核其它部分影响非常大。

RFRTOS 的细粒度时钟机制实现方法最大的优点是：精密时钟刻度的实时核心时钟与 Linux 核心时钟独立开来，在系统中并行运行。这样系统的效率和实时性都能大大增强。同时，为实时定时器设计的堆数据结构使得对定时器队列的维护简单化。

缺点：双内核时钟的方案增加了系统的复杂性。

4.6 高精度定时器的设计与实现

我们通过对 Linux 2.6 内核时钟机制的分析和研究，并借鉴目前各种流行的时钟细粒度方案，设计实现了基于 Linux 2.6 内核的高精度定时器，并提供符合 POSIX 1003.1b 实时扩展标准的 API 接口。该高精度定时器方案提高了传统 Linux 内核的

定时器精度，满足了电信级 Linux 实时处理的要求，有力地促进了 Linux 在电信级市场的运用前景。

高精度定时器的基本设计思想与 KURT 的思想有些相似：增加一个内核变量 sub_jiffie 记录在一个时钟周期中流逝的时间（即区间[jiffies, jiffies+1)内所流逝的时间），通过时间戳计数器可以跟踪这个 sub_jiffie 的大小。但不同的是，高精度定时器仍然保留频率为 HZ 的系统时钟，如果在一次时钟周期中有高精度的定时请求，内核通过 jiffies+sub_jiffie 的方法获取超时时间，然后对时钟中断设备进行编程（要么是可编程间隔定时器 PIT，要么是本地 APIC 定时器，用户可以在编译内核时选定其一），并与时间戳计数器中跟踪的当前时间比较，超时则产生中断。

这样内核中既存在按频率 HZ 产生的系统时钟中断，也存在按用户需求产生的高精度定时中断。内核根据两种中断的不同类型，分别发出中断。这样既避免了 KURT 的设计取消系统时钟中断带来的影响，又汲取了 RFRTOS 双内核时钟的思想。

为精确获取当前墙上时间 xtime，需要利用硬件时钟源(TSC, ACPI-PM 等)计算[jiffies, jiffies+1)之间所流逝的时间。这些硬件时钟源提供了从最近一次时钟中断到当前时刻的 ΔT，内核通过结构 timer_opts 中的 get_offset()获取该值：

```
struct timer_opts{
        char* name;
        int (*init)(char *override);
        void (*mark_offset)(void);                    /*通过 TSC 获取 ΔT*/
        unsigned long (*get_offset)(void);
        unsigned long long (*monotonic_clock)(void);
        void (*delay)(unsigned long);
};
```

每次时钟中断处理程序执行时调用 mark_offset()纪录 ΔT, 以后内核就可以调用 get_offset()获得 ΔT。

这种高精度定时器不用再等到系统时钟到来后才检查执行所有超时定时器，根据定时器精度的不同可以在需要的时候产生时钟中断，及时进行处理，提高了定时器的精度。下面分别详细阐述高精度定时器各部分的设计与实现。

4.6.1 高精度定时器的中断源

如上所述，高精度定时器内核中仍然保留了按 HZ 频率产生的系统时钟中断。采用哪种硬件定时器产生高精度动态定时器中断取决于 CPU 上是否有本地 APIC 定时器：如果 CPU 上没有本地 APIC 定时器，那么依然使用可编程间隔定时器产生高精度定时中断；如果有本地 APIC 定时器，最好采用它作为高精度定时器的中断源。因为相比可编程间隔定时器 PIT，本地 APIC 定时器不仅能产生更高精度的中断（PIT 为 32 位，本地 APIC 定时器 64 位），而且让可编程间隔定时器既产生系统时钟中断，又产生高精度定时中断，效率的确不高。

PIT 一共有 6 种编程模式。在单次（one-shot）触发模式下，时钟芯片并不是每个 jiffy 都编程产生中断，只是对于系统中有到期时间小于 1/HZ 的任务是才对时钟芯片编程，设置一个计数器溢出时间，等到这个数值被加载后才继续按照原来的 1/HZ 产生中断。

表 4-1 Intel 8254(PIT)工作模式

PIT 通道工作模式	模式功能描述
Mode 0 (000)	周期中断。
Mode 1 (001)	硬件单次重触发
Mode 2 (002)	信号发生器
Mode 3 (003)	方波信号发生器
Mode 4 (004)	单次触发状态(one-shot)
Mode 5 (005)	硬件触发

当然还要注意 PIT 和本地 APIC 定时器的区别，前者所产生的中断是全局性的，对称多处理器上的每个处理器都会收到这个中断；而本地 APIC 定时器的中断只对本地 CPU 有效。

下面两个函数是调度系统时钟中断和高精度定时器中断的核心函数：

```
schedule_jiffies_int()      /*在下一个 jiffies 产生系统时钟中断*/
schedule_hr_timer_int()     /*在指定的时刻产生高精度定时中断*/
```

图 4-2 为高精度定时器中处理流程。

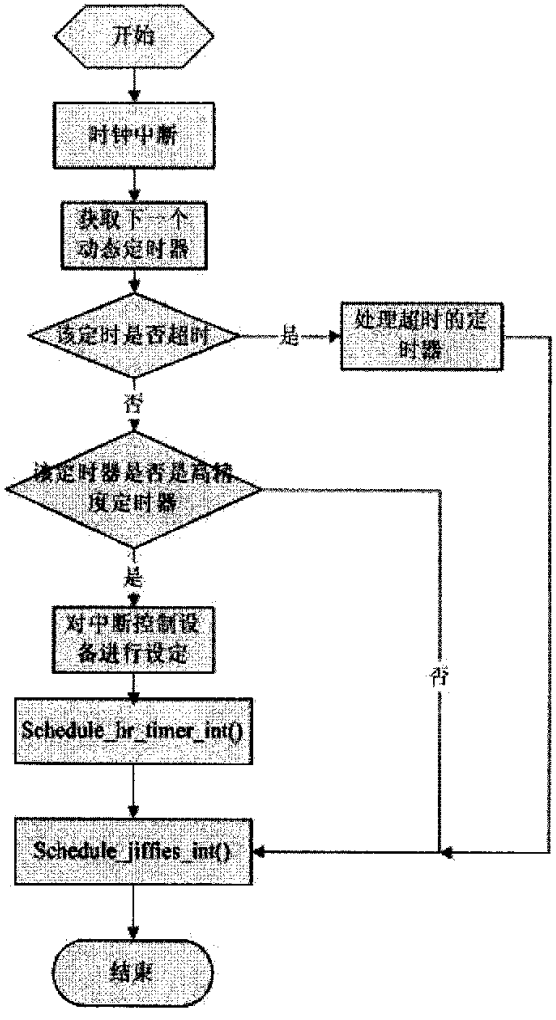


图 4-2 高精度定时器处理流程

4.6.2 高精度定时器的时间机制

高精度定时中断发生在连续两个 jiffies 之间，即[jiffies, jiffies+1)。要确定该中断的产生时间，就必须通过 TSC 获取当前时间并与 jiffies+sub_jiffie 表示的高精度超时时刻比较。通过 get_arch_cycles(ref_jiffies)可以得到当前时刻的 sub_jiffie，其精度是 CPU 的时钟周期。

4.6.3 高精度定时器链表的处理

所有动态定时器都以链表的形式存放在一起。为了寻找超时的定时器而遍历整个链表的做法是低效的；同样，将链表以超时时间进行排序也不是一种很高效的做法，这样做使得在链表中插入和删除定时器的开销较大。

Linux 2.6 内核将定时器按它们的超时值划分为五组。当定时器超时值接近时，定时器将随组一起下移。采用这种分组定时器的方法能尽可能减少内核搜索超时定时器所带来的开销。

但由于高精度定时器以(jiffies+sub_jiffie)表示动态定时器的超时值，所以需要修改当前 Linux 2.6 内核中动态定时器链表的结构，这在一定程度上不可避免的降低了原 2.6 内核定时器管理的高效性。为尽可能地减小这种影响，高精度定时器链表采用 hash 表结构，将所有具体相同 jiffies 值的定时器链在同一个 hash 表项上，这些定时器再根据 sub_jiffie 的大小进行排序，如图 4-3 所示：

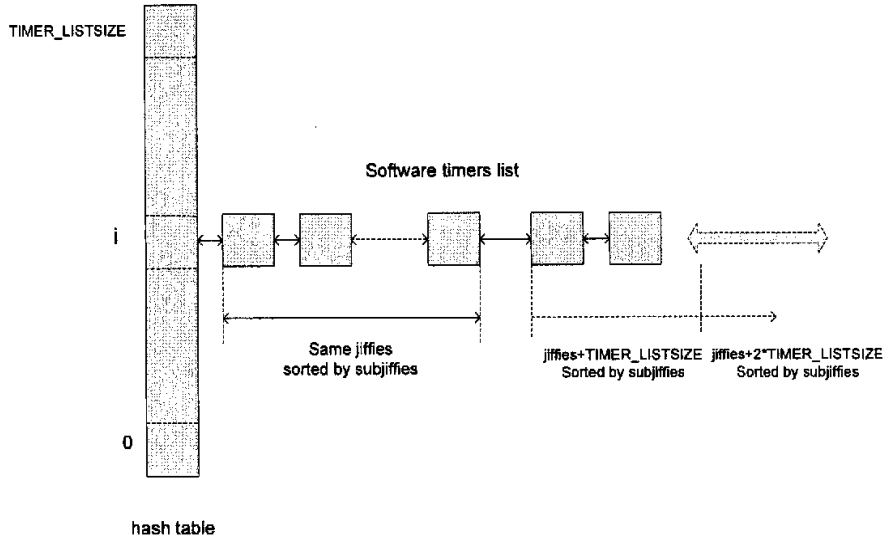


图 4-3 定时器 hash 表结构

每次按下列步骤处理定时器链表：

1) 获取当前时间（jiffies + sub_jiffie）；
2) 获得链表上下一个定时器；
3) 若该定时器已超时，调用其中断处理函数并返回 2）；
4) 若该定时器是高精度定时器，则调用 schedule_hr_timer_int()；
5) 若在当前 jiffies 到 jiffies+1 之间没有高精度定时器，则调用

schedule_jiffies_int();

该 hash 表结构的设计在一定程度上减少了改变 Linux 2.6 中定时器链表结构所带来的系统开销。

遵循 POSIX 1003.1b 实时扩展标准，高精度定时器内核至少支持以下两种 POSIX 标准的时钟：

1) CLOCK_REALTIME_HR: 高精度定时器支持的实时时钟
2) CLOCK_REALTIME: 普通内核支持的实时时钟

4.6.4 高精度定时器主要函数的实现

由于动态定时器在时钟中断的下半部的 softirq 中执行，而 softirq 被激活到 TIMER_SOFTIRQ 真正执行这期间可能会有几次时钟中断。因此内核必须记住上一次运行定时器是在什么时候，即内核必须保存上一次运行定时器时的 jiffies 值。为此，在 kernel/timer.c 中定义了全局变量 timer_jiffies 来表示上一次运行定时器时的 jiffies。

为了使得小于 1/HZ 的时钟中断得以发生，我们在 include/linux/time.h 中定义了两个全局变量

extern unsigned long sub_jiffie;

extern unsigned long int jiffie_intr;

前者以 TSC 所支持的最大精度为单位记录在一个 jiffy 内的时间流逝；后者当 sub_jiffie 大于一个 tick 时加一。函数 update_jiffies()更新 sub_jiffie 的值，同时也更新全局变量 jiffies 和 jiffie_intr。

如果没有使用 APIC 产生高精度定时器中断，我们需要把 PIT 设置为单次触发模式。这时时钟中断不再是周期性的间隔发生，所以需要一个函数来计算下一次中断的执行时间。我们使用 time_to_next_event()实现这个功能。

内核更新完 jiffies 和 sub_jiffie 之后，便调用定时器处理函数 run_timer_list()。这个函数检查是否有等待处理的定时器。如果有，则调用相应的处理函数；最后，内核更新当前进程以及内核中时间相关的统计信息，并根据这些信息作出相应的动作执行诸如更新处理器时间，调度新新进程等。

run_timer_list()算法：

1) 获取定时器列表操作自旋锁 timerlist_lock；
2) run_timer_list_again：读取当前时间 jiffies 以及 TSC 的读数，把 TSC 的读

数赋给 sub_jiffie;

3) 比较当前 jiffies 与上一次中断发生时间 timer_jiffies。这两个差值表示自前次中断以来，期间一共有多少正常的时钟中断发生。显然，在这个函数中将为每一次中断补上定时器服务。
4) 扫描 HRT HASH 表。比较定时器 expires 值与当前 jiffies，同时比较 sub_expires 和 sub_jiffie。
5) 若 timer->expires > jiffies，则跳出函数。
6) 若 timer->expires <= jiffies，或者 timer->expires = jiffies&&timer->sub_expires<= sub_jiffie,则执行相应的定时器服务函数。在这里实现了比 1/HZ 更高精度的时钟中断响应。
7) 若 timer->expires = jiffies，timer->sub_expires > sub_jiffie，说明当前高精度定时器还没有到期，跳转到 2);
8) 执行 timer_jiffies++;
9) 结束，释放定时器操作自旋锁。

从上面过程可以看出，这个算法能确保小于 1/HZ 的高精度定时器可以发生，并执行相应的定时器服务程序。

高精度定时器时钟中断处理程序依然分为与体系结构相关部分和与系统结构无关的 do_timer()。

do_timer()与普通内核中的实现完全一样。

高精度定时器对时钟中断处理程序的影响体现在 do_timer_interrupt_hook()中：

```
#ifdef CONFIG_HIGH_RES_TIMERS
...
discipline_PIT_timer();
do{
    do_timer(regs);
    stake_cpuctr();
}while ((unsigned)get_arch_cycles(jiffies) > arch_cycles_per_jiffy);
```

如果配置高精度定时器，需要调用 discipline_PIT_timer()。discipline_PIT_timer()的作用是对 PIT 产生的偏移作处理，这是由于用 PIT 同时产生系统时钟中断和高精度时钟中断而产生的。所以 discipline_PIT_timer()的现实还跟是否使用有 CPU local APIC 有关，若有，则不需要 discipline_PIT_timer()做任何事情。

```
#ifdef CONFIG_X86_LOCAL_APIC
```

```
#define USE_APIC_TIMERS
/*如果配置了 APIC，则将* discipline_PIT_timer()定义为空*/
#define discipline_PIT_timer()
#else
#define TIMER_NEEDS_DISCIPLINE
#define discipline_PIT_timer() discipline_timer()
#endif
```

schedule_hr_timer_int()和 schedule_next_int()是调用系统时钟中断和高精度定时器中断的核心函数，它们在内核中定义为宏：

```
#define schedule_hr_timer_int(a, b)   _schedule_next_int(a, b)
#define schedule_jiffies_int(a)       _schedule_jiffies_int(a)
```

a 表示以 1/HZ 为单位的时间值，b 表示小于 1/HZ 的时间值（即大于 jiffies,小于 jiffes+1 之间的时刻）。

_schedule_jiffies_int(a)不带参数 b，其作用是判断是否应该触发下一个精度为 1/HZ 的系统时钟中断。

_schedule_next_int(a, b)含有参数 a、b，其作用是判断是否应该在时刻(a+b)动态触发一次精度高于 1/HZ 的时钟中断。

根据系统及内核配置的不同，_schedule_jiffies_int()的实现有较大差异，其具体实现在 arch/i386/kernel/time.c 中。 如果内核既配置了高精度定时器又使用了 APIC 定时器，由于产生系统周期性时钟中断和高精度时钟中断的设备不同，所以不存在 PIT 的时间补偿，_schedule_jiffies_int(a)直接通过比较 a 和 jiffies 的大小就可以完成判断；如果没有配置 APIC 定时器，则需要 PIT 同时产生两种时钟中断，必然存在中断延迟，所以需要记录延迟。

为简化_ schedule_jiffies_int()，当内核没有配置 APIC 定时器是，通过_schedule_next_int() 的方式完成，具体做法是传递适当的参数给_schedule_next_int()。

```
#ifdef CONFIG_HIGH_RES_TIMERS      /*内核配置了 HRT*/
#ifdef USE_APIC_TIMERS             /*内核使用 APIC 定时器*/
int _schedule_jiffies_int(unsigned long jiffie_f)
{
    long past;
    unsigned long seq;
```

```
    if (unlikely(!hrtimer_use)) return 0;
    do {
        seq = read_seqbegin(&xtime_lock);
        past = get_arch_cycles(jiffie_f);
    } while (read_seqretry(&xtime_lock, seq));

    return (past >= arch_cycles_per_jiffy);
}
#else
int _schedule_jiffies_int(unsigned long jiffie_f)
{
    /*通过_schedule_next_int()实现_schedule_jiffies_int()*/
}
#endif
```

通过传递适当的参数，_schedule_next_int()可以完成跟_schedule_jiffies_int()相似的功能，即触发周期性的时钟中断。

_schedule_next_int()的实现：

```
int _schedule_next_int(unsigned long jiffie_f,long sub_jiffie_in)
{
    long sub_jiff_offset;
    unsigned long seq;

    /*如没有使用 HRT，直接返回 0*/
    if (unlikely(!hrtimer_use)){
        return 0;
    }
    do {
        seq = read_seqbegin(&xtime_lock);
        sub_jiff_offset = sub_jiffie_in - get_arch_cycles(jiffie_f);
    } while (read_seqretry(&xtime_lock, seq));
    /*已经超时，返回 1  */
    if (sub_jiff_offset <= 0)
```

```
        return 1;
    /*对时间误差进行处理*/
    __last_was_long = arch_cycles_per_jiffy == sub_jiffie_in;
    reload_timer_chip(sub_jiff_offset);
    return 0;
}
```

符合 POSIX1003.1b 第 14 节（时钟和定时器部分）API 规范的应用程序接口包括 clock_gettime()，clock_settime()，clock_getres()，clock_nanosleep，timer_settime()，timer_gettime()，timer_getoverrun()，timer_create()这一组函数。它们均通过系统调用在 kernel/posix-timers.c 中实现。

简单地说，API 是内核中对应系统调用在用户空间中的实现。图 4-4 为 API 函数与内核中系统调用的关系。

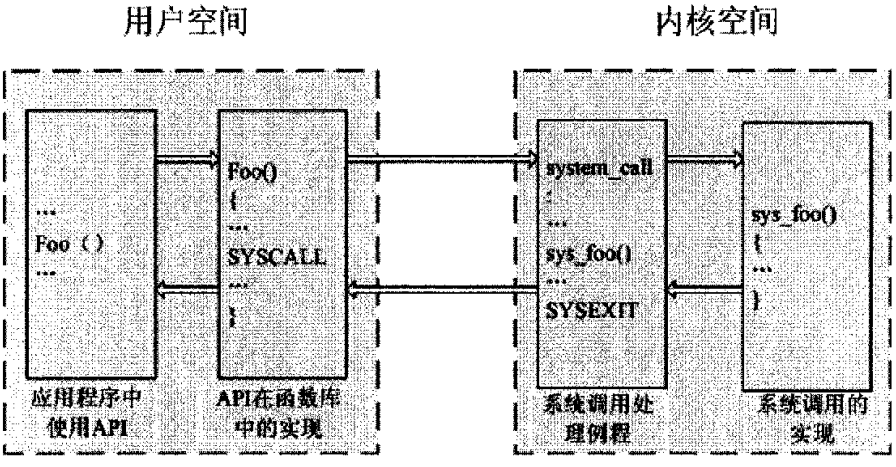


图 4-4 API 函数与内核中系统调用的关系

下面以 clock_gettime()为例简要说明 POSIX API 与系统调用的关系。

```
asmlinkage long
sys_clock_gettime(clockid_t which_clock, struct timespec __user *tp)
{
    struct timespec kernel_tp;
    int error;
    /*调用 do_clock_gettime()，获取内核态下的时间参数*/
```

```
    error = do_clock_gettime(which_clock, &kernel_tp);
    /*copy_to_user()完成用户态和内核态间的数据拷贝*/
    if (!error && copy_to_user(tp, &kernel_tp, sizeof (kernel_tp)))
        error = -EFAULT;
    return error;
}
```

系统调用在内核中均以 asmlinkage 开头，且函数名称统一以 sys_为前缀。系统调用 sys_clock_gettime()通过调用内核函数 do_clock_gettime()获得内核空间内的时间参数的地址，最后通过内核中常用的 copy_to_use()将内核空间内的时间参数地址传给用户空间时间参数。

第五章 高精度定时器的测试

5.1 测试设计

高精度定时器的测试计划主要根据电信级 Linux3.0 规范中高精度定时器的描述和 POSIX 1003.1b 规范进行。大体上，高精度定时器的测试分两步完成。

1) 第一步，集成测试（IC，Integration Check）和自动基本可接受性测试（ABAT， Automated Basic Acceptance Testing）；
2) 第二步，按照测试目的的不同，测试再分为性能测试（performance testing）和兼容性测试(Conformance testing)两部分。

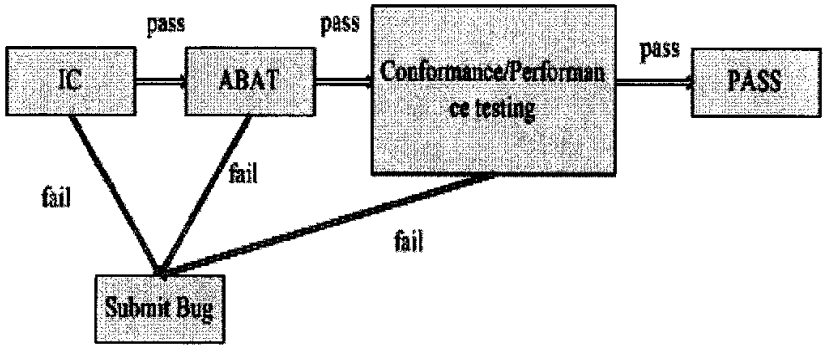


图 5-1 基本测试流程

5.1.1 集成测试和自动化基本可接受性测试

集成测试是对内核高精度定时器检查的第一步，其目的简单明确：检查内核是否支持高精度定时器；通过对内核配置文件的检查，寻找内核是否有高精度定时器选项。集成测试由 shell 脚本实现。根据 Linux 发行版本的不同，依次检查三个内核配置文件：/proc/config.gz 、/lib/modules/`uname -r`/build/.config 和 /usr/src/linux/.config 中是否有高精度定时器的相关选项 CONFIG_HIGH_RES_TIMERS，CONFIG_HIGH_RES_TIMER_TSC。

自动化基本可接受性测试比集成测试更进一步，它将检查高精度定时器选项是否已经成功地编译进入内核。所以在完成集成测试之后，测试人员必须重新编译内核，将高精度定时器的内核选项打开。

内核编译过程简单描述如下：

1) mak menuconfig；

2) 进入时钟、定时器配置相关选项页面（图 5-2，内核时钟、定时器配置页面入口）；

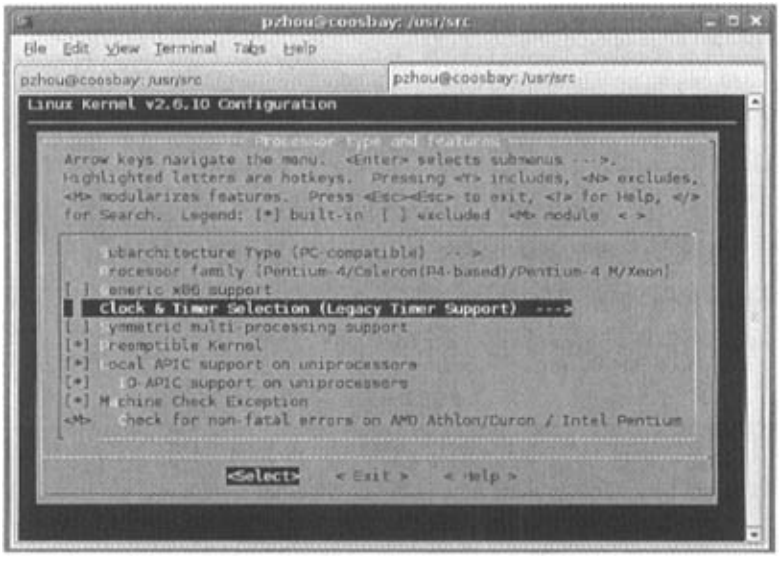


图 5-2 内核时钟、定时器配置页面入口

3) 打开高精度定时器支持选项（图 5-3，内核高精度定时器配置界面）；

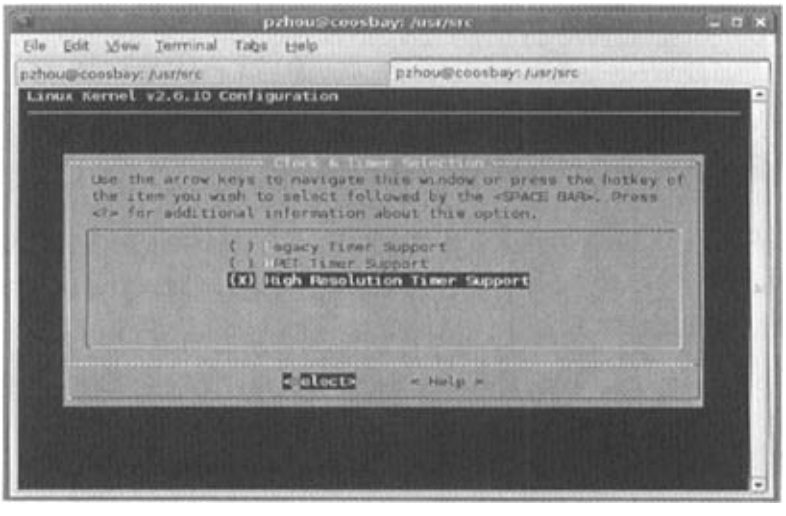


图 5-3 内核高精度定时器配置页面

4) 选择定时器时钟源为 TSC（图 5-4，打开高精度定时器选项后的内核配置

界面）；

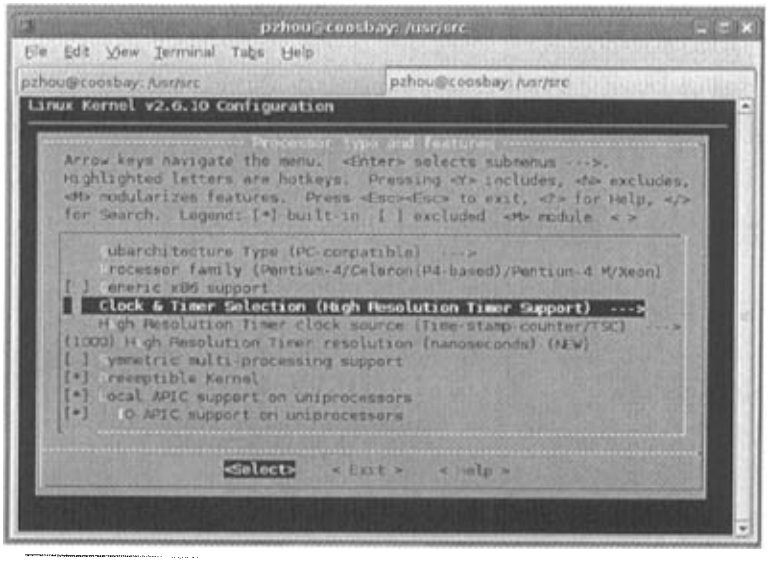


图 5-4 打开高精度定时器选项后的内核配置界面

5) 设置定时器精度（图 5-5，设置内核定时器精度）；

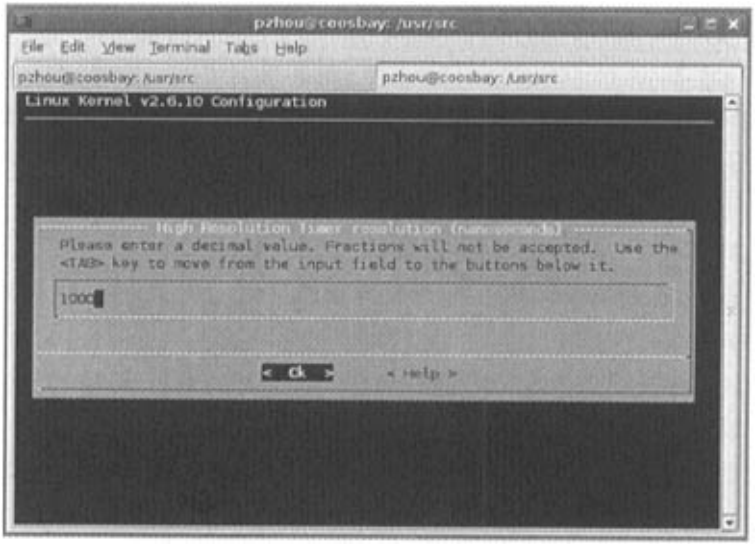


图 5-5 设置内核定时器精度

6) 编译新的内核

make

7) 安装新的内核并生成 initrd

make install

重新启动计算机进入刚刚编译成功的内核中，这时再检查配置文件/proc/config.gz、/lib/modules/`uname -r`/build/.config 和/usr/src/linux/.config 中的选项 CONFIG_HIGH_RES_TIMERS 是否等于“Y”。如果是，自动基本可接受性测试返回成功，否则表示内核尚不能成功地支持高精度定时器。

5.1.2 性能测试和兼容性测试

性能测试主要从应用程序使用的角度，分别在不含高精度定时器的内核和含有高精度定时器的内核上对 POSIX 时钟定时器函数接口进行测试，最后对比分析两个平台的测试结果。

在不含高精度定时器的环境中，调用 POSIX 函数时使用 CLOCK_REALTIME 类型的时钟；在含有高精度定时器的环境中则使用 CLOCK_REALTIME_HR 类型的时钟[18]。需要特别说明的是，我们在测试时把内核高精度定时器的精度设置为 10 微秒。

性能测试的第一个实验测量 clock_nanosleep()在不同睡眠时间下的平均延迟。方法为测试 clock_nanosleep()的实际睡眠时间，睡眠时间从 100 纳秒到 4 秒，按 100 纳秒为单位递增。

性能测试的第二个实验测量软件定时器的平均偏差。方法为利用 timer_settime()，timer_gettime()，和 timer_create()等系列函数设定间隔定时器 6000 次，每次间隔时长为 10 微秒。

性能测试的第三个实验反复测量 clock_nanosleep()实际睡眠时间[19]。方法为调用 clock_nanosleep()10000 次，每次睡眠时间均为 50 微秒。其目的是检查 clock_nanosleep()是否出现睡眠时间偏差较大的情况。

以上性能测试均在普通内核和高精度内核上进行，最后对比两个平台测试数据并进行分析。测试结果和数据见 5.4 节测试结果和分析。

兼容性测试，顾名思义就是测试函数接口的表现形式是否与函数规范中提到的形式一致。高精度定时器的兼容性测试侧重于 POSIX 函数接口自身行为的测试，即根据 POSIX 1003.1b 所描述的各个函数接口的参数，返回值，使用方法等[20]对高精度定时器在用户态的 API 接口进行验证。

兼容性测试的依据主要是 POSIX 1003.1b 中对时钟和定时器的描述，以及 Linux 的 man 手册。测试人员需要在理解这些规范和手册的基础上，列出有价值的

测试参考点，写出初步的测试计划，并在测试人员之间互相审核，以最大限度覆盖规范和手册中描述的情形。最后测试人员根据各个测试点参考点，分别编写测试用例，检查函数接口的表现形式是否与规范、手册中描述的情况相吻合。

5.2 测试环境

所有测试均在电信级服务的主流平台 AdancedTCA 上进行。AdancedTCA 符合电信级模块化和标准的思想，具有开放的设计规范。由于 AdancedTCA 是业界各大公司合力推动的平台，所以选择它作为测试的硬件平台比较具有代表性。

硬件环境：

1) AdancedTCA 服务器；
2) 2 * Intel® Xeon™ 1.6GHz 处理器；
3) 内存 2GB；

软件环境：

1) Linux 内核 2.6.10 及其源代码；
2) 基于 Linux 内核 2.6.10 的高精度定时器补丁；
3) 选择本地 APIC 定时器作为高精度定时器中断源，并将内核定时器精度配置为 10 微秒。

5.3 测试结果及分析

性能测试一是测量在高精度定时器内核和普通 2.6.10 内核上睡眠（使用 clock_nanosleep()）不同时间的平均延迟，睡眠时间从 100 纳秒到 4 秒，按 100 纳秒为单位递增[20]。

在普通内核上，由于定时器精度是 1 毫秒，可以预测平均延迟时间在毫秒级；在高精度内核上，我们将精度配置为 10 微秒，由此预测平均延迟时间在 10 微秒级。

实际测试结果见表 5-1：

表 5-1 平均睡眠延迟对比

内核版本	普通 2.6.10 内核 (精度＝1 毫秒)	基于 2.6.10 的高精度定时器内核 （精度＝10 微秒）
平均延迟时间	1996892 纳秒	24934 纳秒

从上表可以看出，应用程序的实时响应延迟有效的从毫秒级降低到 10 微秒级，和之前的预测相符合。

性能测试二是测量定时器偏离理论时刻的振荡值。实验预测分析与性能测试一类似，定时器实际发生时刻偏离理论发生时刻的振荡值应该又毫秒级下降到数十微秒级。

表 5-2 列出了在普通内核和高精度定时器内核上定时器平均振荡时间。

表 5-2 平均振荡时间对比

内核版本	高精度定时器内核 （精度＝10 微秒）	普通内核 （精度＝1 毫秒）
平均振荡时间	13.26 micro-second	998.40 micro-second

从上表可见，高精度定时器内核中的平均振荡时间比普通内核的平均振荡时间有大幅度的减少。这表明定时器的实际发生时间偏离期望的理论时间越来越近。

性能测试三反复测量 clock_nanosleep()的实际睡眠时间。图 5-6、图 5-7 分别绘制了在普通 2.6.10 内核和基于 2.6.10 的高精度定时器内核上睡眠 50 微秒 10000 次所得到的实际睡眠时间。

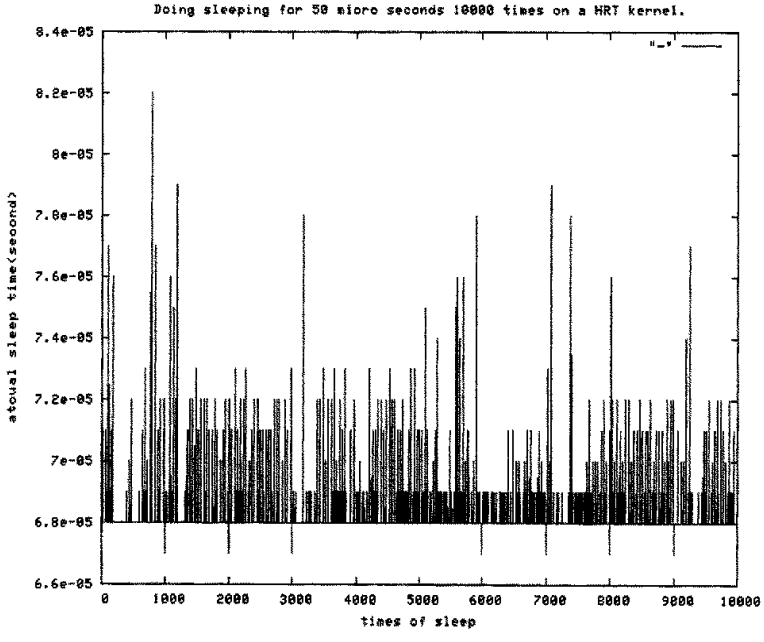


图 5-6 在高精度定时器内核上睡眠 50 微秒 10000 次

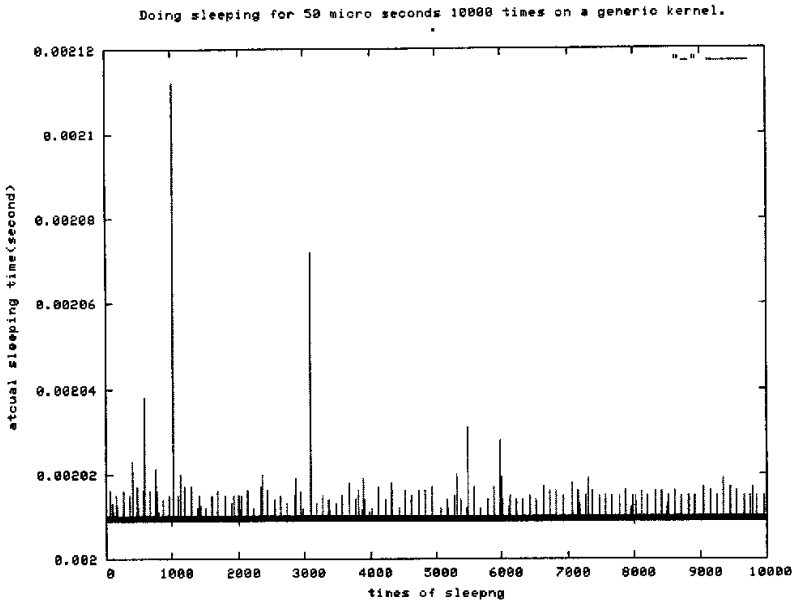


图 5-7 在普通内核上睡眠 50 微秒 10000 次

图 5-6 和图 5-7 中可以看出，高精度定时器内核中定时器的时间睡眠时间范围为 67 微秒到 82 微秒；普通的内核定时器受 1 毫秒系统时钟中断的限制，实际睡眠时间范围为 2.01 毫秒到 2.12 毫秒。由此可以看出，对于定时时间小于 1 毫秒的应用来说，高精度定时器内核对应用性能的改善较为明显。

测试结果表明：高精度定时器有效地提高了内核动态定时器的精度，增强了 Linux 在电信级应用的软实时能力，符合 CGL3.0 规范的要求，有利于将电信应用平台向 Linux 移植。

第六章 结束语

7.1 全文总结

本课题所涉及的内容是 Intel 中国软件中心（上海）为满足电信级 Linux 的实时性需求，对 Linux 内核所做的增强与改进。目前开发和测试工作已经告一段落。该方案及其设计实现以内核补丁的形式运用在各电信级 Linux 发行版本中。

本文作者从理论和实践两方面对 Linux 2.6 内核的时间子系统和定时器作了深入全面的研究分析；并结合电信级服务和模块化通信平台对 Linux 提出的新的要求，设计实现了一种高精度的内核定时器，其主要工作如下：

1) 深入理解 2.6.10 内核中的时间子系统及内核定时器：阅读分析 Linux 内核源代码，学习内核开发的基本特点和开发流程。
2) 理解基于标准的模块化网络平台和电信级 Linux：学习总结模块化网络平台的体系结构和特性；分析电信级 Linux 的组织实施方案和需求；理解高精度定时器在电信级服务中的应用背景。
3) 设计实现一种满足电信级 Linux 服务需求，具有 POSIX 兼容接口的高精度定时器。测试结果表明该设计有效地将定时器精度减小到微秒级。

7.2 下一步工作

随着 Linux 在服务器及嵌入式设备上越来越广泛的应用，尤其是虚拟技术 XEN 的兴起，各种服务对分时、通用 Linux 系统的实时性能要求不断提高，本课题中的设计仍存在值得进一步探讨和改进的地方：

1) 在保留系统时钟的周期性中断的基础上， 如何在加入高精度定时中断时减小系统开销。
2) 如何使 HASH 表结构组织的定时器列表减小对原内核中定时器列表瀑布模型性能的影响。
3) 由于虚拟技术的兴起，如何有效地处理多个虚拟 Linux 系统中的各时间系统和 CGL 中的时间子系统是我们所面临的挑战。该挑战极有可能从根本上改变长期以来以周期性时钟中断为基础的时间子系统设计结构：操作系

统并非一定需要固定周期的时钟中断。理想的办法是只使用一个统一的定时器，通过编程设定，让它在前一个时间到期时开始计时，在它被触发执行后，再设定定时器为下一个事件服务，如此反复。这种方法不再需要周期性的时钟中断，可以大量减少系统为处理时钟中断所带来的开销。

但这样做需要解决两个问题：

1) 没有了 jiffies，如何跟踪系统的运行时间。

2) 如何克服完全动态管理定时器所带来的负担。

上述问题都需要我们在以后的工作和学习中进一步深入探索。

致 谢

我的毕业设计是在导师周明天教授的耐心指导下完成的。在课题设计过程中，无论是遇到问题，还是毕业论文撰写，周老师都给了我耐心的指导和深刻的启示。他在学术方面的深厚造诣和渊博知识，严谨而敏捷的思维，不断创新的精神，耐心宽厚的师长风范，给我留下了深刻的印象，也将使我一生受益。在此，我向给予了我无私关怀、指导和帮助的周老师表示衷心的感谢！

我的毕业设计课题得到了 Intel（上海）中国软件中心开源技术项目中心(Open Source Technology Center)的大力支持，尤其要感谢谢钟梅，路易斯. 庄, 刘宏，杜松波等各位领导和同事的帮助。他们在我的毕业设计过程中给予了我很多指导，使我能够圆满完成毕业设计任务，感谢他们对我学习和工作的帮助和启发！

同时也向一切给予我帮助，使我顺利完成硕士研究生学业的人们表示衷心的谢意。

我的父母是我来到电子科大学习深造的动力和支持，也是我生活和精神上的支柱，在此，我要特别感谢他们。

参考文献

[1] www.osdl.org.

[2] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel,2nd edition. O'Reilly Publishing, 2002.

[3] Robert Love, Linux Kernel Development . USA: Sams Publishing, 2004.

[4] www.kernel.org.

[5] Open Source Development Labs. Carrier grade Linux requirements definition (Version 3.0).http://www.osdl.org/docs/cgl_perf_req_def_30.pdf, 2005.

[6] I A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity OS". In 5th Symp. Operating Systems Design & Implementation, pp. 165–180, Dec 2002.

[7] Barabanov M, Yodaiken V. Introducing real-time Linux. Linux Journal 34, Feb 997. http://www.linuxjournal.com/article.php?sid=0232.

[8] Srinivasan B, Hill R, Pather S, Niehaus D. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In: Proceedings of the Real-Time Technology and Applications Symposium. Denver, 1998. 112~119.

[9] J. Vidal, F. Gonzálvez, I. Ripoll, POSIX TIMERS implementation in RTLinux, rtportal.upv.es/apps/rtl-timers/posix_timers-0.1/timers-info.pdf

[10] M. Aron and P. Druschel, "Soft timers: efficient software timer support for network processing". ACM Trans. Comput. Syst. 18(3), pp. 197–228, Aug 2000.

[11]Yu-Chung Wang and Kwei-Jay Lin, Enhancing the Real-Time Capability of the Linux Kernel, Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference, 1998.

[12] Yoav Etsion, Dan Tsafrir, Dror G. Feitelson, Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes, In the Intl. Conf. on Measurement & Modeling of Comput. Syst. (SIGMETRICS), Jun 2003.

[13] Bill O. Gallmeister, Chris Lanier, "Early Experience With POSIX 1003.4 and POSIX 1003.4A", Proceedings of the IEEE Symposium on Real-Time Systems, pages 190-198, 1991.

[14] Hill R, Srinivasan B, Pather S, Niehaus D. Temporal resolution and real-time extension to

Linux. Technical Report, ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Departmaent, University of Kansas, 1998.

[15] 李小群，“RFRTOS：一个基于Linux的实时操作系统”，中国科学院软件研究所博士论文，2002.

[16] 李小群，赵惠斌，叶以民，孙玉芳，“RFRTOS：基于Linux的实时操作系统”，软件学报， vol. 14, No.7, 2003.

[17] 李小群，赵惠斌，叶以民，孙玉芳，“一个基于时钟粒度细化Linux实时方案”，计算机研究与发展， vol. 40, No.5, 2003.

[18] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, Jonathan Walpole. "A Measurement-Based Analysis of the Real-Time Performance of Linux", Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02), vol. 00, p.133-p.142, IEEE Computer Society Washington, DC, USA, 2002.

[19] Andy Pfiffer, study of Linux 2.5 Timer Scalability, www.osdl.org

[20] IEEE Std 1003.1-2001, System Interfaces, Issue 6 – for descriptions of interfaces of POSIX clocks and timers. http://standards.ieee.org/catalog/olis/posix.html.

个人简历

周鹏，男，出生于1981年5月。2003年7月毕业于电子科技大学计算机学院，获计算机科学与技术学士学位及“电子科技大学优秀毕业生”称号。2003年9月进入电子科技大学计算机学院，攻读计算机软件与理论专业研究生。

研究生期间参与的科研项目：

1) 2004年8月至2005年7月，在Intel中国软件中心（上海）的开源技术中心（OTC, Open Source Technology Center）实习，从事CGL(电信级Linux)的开发与测试。
2) 2006年3月至今，在NOKIA成都研发中心实习，参与Rich Call Linux Client的开发与测试。

研究生期间获奖情况：

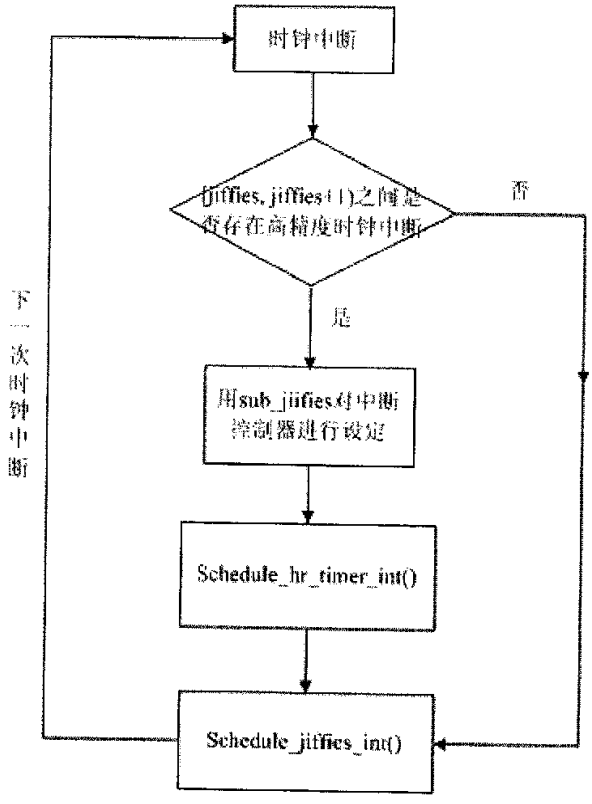
1) 荣获2003年-2004年度研究生三等优秀奖学金。

研究生在校期间论文发表情况：

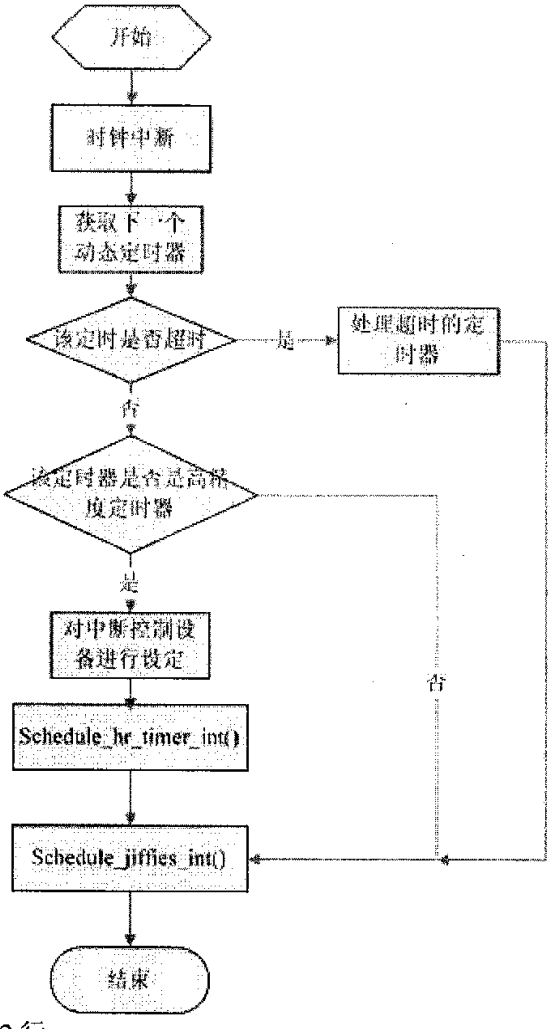
1) 《Linux内核中一种高精度定时器的设计与实现》，微机发展，2006年5月。

1. 51 页图 4-2 高精度定时器处理流程

原文：



改为：



2. 66 页第 5 段第 2 行

原文：

测试结果表明该设计有效地将定时器精度减小到微妙级。

改为：

测试结果表明该设计有效地将定时器精度减小到微秒级。

3. 68 页第 2 段第 2 行至第 3 行

原文：

尤其要感谢 Xie May, Louis Zhuang, Liu Hong, Du Songbo 等...

改为：

尤其要感谢谢钟梅，路易斯．庄，刘宏，杜松波等...

4. 删除 3.4.3 节（23 页）中关于第三方机构对电信级平台性能的测试对比。

导师：

学生：