顺序号（硕）：SS021014
姓名：张斌
学号：210802118
学院：210802118
专业：计算机软件与理论

分类号_____密级_____
UDC _____

研究生学位论文

Linux 系统进程调度策略研究

研究生姓名 张斌
指导教师姓名 魏振钢 教授

申请学位级别 硕 士 专业名称 计算机软件与理论
论文答辩日期 2011 年 5 月 28 日 学位授予日期 2011 年 6 月

中 国 海 洋 大 学

谨以此论文献给我敬爱的导师魏振钢教授

----------张斌

Linux 系统进程调度策略研究

学位论文完成日期：2011.5.26

指导教师签字：

答辩委员会成员签字：

独 创 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，也不包含未获得（注：如没有其他需要特别声明的，本栏可空）或其他教育机构的学位或证书使用过的材料。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。

学位论文作者签名：张纲 签字日期：2011年 5月 26日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，并同意以下事项：

1、学校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。

2、学校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。同时授权清华大学“中国学术期刊(光盘版)电子杂志社”用于出版和编入CNKI《中国知识资源总库》，授权中国科学技术信息研究所将本学位论文收录到《中国学位论文全文数据库》。（保密的学位论文在解密后适用本授权书）

学位论文作者签名：张纲 导师签字：

签字日期：2011年 5月 26日 签字日期：2011年 5月30日

Linux 系统进程调度策略研究

摘　要

操作系统的进程调度策略一直是一个研究热点。随着技术的进步，多核体系结构逐渐成为主流，这为操作系统的进程调度研究带来了挑战。其中，负载均衡问题扮演着很重要的角色。负载均衡研究的是如何把任务比较合理地均匀地分配到多个处理器核心上，从而比较充分地发挥多核的优势，在整体上提高系统的性能和缩短任务的平均响应时间。

本文借助 Linux 内核的开源特性来研究操作系统的进程调度及多核负载均衡等问题。首先阐述了进程、线程等概念在 Linux 内核中的实现及进程调度方面的基本算法；然后结合 Linux 内核的发展，重点分析讨论了 Linux2.6 内核所采取的两种主流的调度系统：O(1)调度及 CFS 调度，在理解现有调度系统的基础上分析其不足，特别是其在负载均衡方面的缺陷；最后提出针对 Linux 内核的负载均衡策略的优化与改进。

针对Linux内核在负载均衡问题上只考虑CPU队列长度而忽略进程本身特点的弊端，本文提出一种基于综合指标的负载均衡策略，综合考量进程使用 CPU、内存及 I/O 的情况，重点研究了负载均衡问题中迁移进程的选择问题及迁移算法的实现，最后通过修改 Linux 内核的相关数据结构及函数并重新编译内核来生成。实验证明，基于综合指标的负载均衡策略在一定程度上优于基于队列长度的负载均衡策略。

关键词：Linux；多核；进程调度；负载均衡

The Research on Scheduling System of Linux Operating System

Abstract

The scheduling system of operating system has been a hot research topic for many years. As the technology develops, the multi-core processor has become the mainstream, and this brings new challenges to the research on task scheduling of operating system, especially to the research on the function of load balancing. Load balancing aims to distribute the tasks evenly among the several cores of a processor. This is very important for improving the performance of the whole system and reducing the average time of task execution.

In this thesis, we focus on the topic of task scheduling and load balancing of Linux kernel due to its free access to the source code. Firstly, the definitions of task, or process, and thread in the kernel are introduced as well as the basic task scheduling algorithms. Then we probe the two main task schedulers used in Linux 2.6 kernel: O(1) scheduler and CFS scheduler. Based on this, we analyze the drawbacks of the current scheduler, especially its limitation in load balancing. At last, we bring forward measures to improve the load balancing system of Linux kernel.

The current load balancer of Linux kernel is based on the number of tasks in the running queue of CPU and neglects the different types and characteristics of tasks. So we put forward a new load balancing algorithm based on multi-targets taking the usage of CPU, memory and I/O into account. Then we investigate how to decide which tasks can be immigrated and how to immigrate them. At last, we implement our algorithm by modifying some data structures and several functions in the kernel. As the experiment shows, our new load balancer is better to some extent.

Keywords: Linux; Multi-core; Task Scheduling; Load Balancing

目　录

第一章 绪论

1.1 研究背景

元器件技术的进步一直是计算机技术不断发展、计算机性能不断提高的积极因素之一。从第一台电子计算机“ENIAC”诞生到现在，电子计算机已经经历了电子管、晶体管、集成电路、大规模集成电路、超大规模集成电路等几代的发展[1]。随着微处理器芯片集成度的提高以及对应的影响运算速度的关键指标—主频的提高，现代计算机的能力显著增强。二十世纪六十年代，摩尔指出：大约每隔18个月芯片可集成的晶体管数量就能翻一番，同时性能也会提升一倍[2]。这就是著名的摩尔定律。但是到本世纪，微处理器主频的数量级已经达到GHz，集成的晶体管数量已达几亿，甚至十几亿。计算性能的提高已很难再遵循摩尔定律的思路，生产工艺似乎已遭遇到瓶颈。相关研究表明，当芯片电路导线的宽度小于10^{-7}m 时，电压引起的雪崩击穿和强电场、隧穿现象随距离缩短而呈指数增加以及随着集成度提高而产生的耗散热量的急剧增高，将非常突出。同时光刻技术难度和成本问题也会成为传统大规模集成工艺的极大障碍[3]。

另一方面，在计算机发展的过程中，计算机应用已经发生了很大的演变[4]。现在，以商用事务处理和Web服务为代表的应用日益成为主流。回顾计算机的近30年发展历史，我们不难发现，计算机应用已经从传统的计算密集型的科学技术应用，发展到了如今的数据密集型应用，表现出完全不同的执行和数据访问的特征。传统的计算密集型应用，对于数据的运算操作远远多于数据的装入操作，因而具备很高的代码和数据访问的局部性，能够有效的利用预取操作数、Cache等技术来弥补内存带宽的不足以及内存访问未命中所造成的时间延迟。而当前面向商用事务处理的主流应用，则是数据密集型的应用。也就是说，这些应用所需数据的时间和空间局部性很差，数据重用的可能性很低。这时，传统的高性能计算机的体系就不能适应新的应用的需求了。

因此，和计算机应用的演变相适应，高端计算机的核心评价指标正在发生根本性的变化，从高性能计算（High-Performance Computing）转向高效能计算（High-Productive Computing），从强调单任务的性能到更多的强调多任务处

理的吞吐量（Throughput）。为了提高系统整体的性能，就势必要求突破传统局限，提高运算的并行性。

为此，研究人员改变思路提出了新的多核处理器体系结构。IBM、Sun、Intel和 AMD 都已经将他们的芯片生产线从制造单核 CPU 变为制造多核 CPU。多核结构是指在同一个芯片内部集成两个或多个处理器核心，每个处理器核心都有各自独立的控制器、运算器、中断控制器以及一级二级缓存等。即所谓的多核体系结构是指通过增加计算机系统中物理处理器的数量实现真正意义上的并行执行，从而提高计算机系统的性能。多核体系结构的提出代表了计算技术的一次创新，它与单核体系结构相比具有相当大的优势，多核体系结构已经成为被广泛采用的计算模型。

多核处理器的发展，对操作系统的相关研究既提供了机遇，又提出了更高的要求。第一，与单核相比，如何在多核环境下各个 CPU 间合理地均匀地调派任务才能充分发挥计算机系统结构优势，提高系统整体的性能？其次，如何降低这种底层核心架构的改变对上层计算机系统用户的影响，保持对外接口的稳定？对计算机系统各种使用者而言，最好是实现计算机体系结构尽可能平滑地过渡。一方面，用户界面最好不要发生改变，另一方面，最好能保持对以前开发的软件及程序兼容，节省重复开发的成本。这些都是在设计和开发多核体系结构下系统软件所必须面对和解决的问题。

但归结为一点，如何高效率的调派任务以便将多核体系结构的性能发挥到极限是多核平台下研发操作系统的核心命题，这是人们对多核系统软件的最大要求。

1.2 国内外研究现状

针对新的多核平台的相关的操作系统方面的研究从起步至今还比较短暂，许多方面需要完善，特别是缺乏比较完备的进程调度策略。因此，多核平台下操作系统的进程调度问题是当今比较前沿的一个研究热点。在本文，我们基于 Linux 内核研究其在多核环境下的进程调度问题。

尽管相对于其他操作系统的漫长历史来说，Linux 的历史非常短暂，但 Linux 在从其问世到现在短短的时间之内得到了非常迅猛的发展，已成为主流的多用户

多任务的操作系统之一，而且具有良好的特性，特别是其开放性、可靠的安全性及良好的可移植性使其获得了广泛的应用[5]。Linux 与 Unix 完全兼容并且开放源代码，也使其成为操作系统的研究人员的不二选择。

实际上我们通常所指的Linux操作系统准确而言应该称做GNU/Linux操作系统,Linux仅代指该操作系统的内核Kernel。Linux内核由芬兰人Linus Torvalds在上世纪90年代初期设计。核心的开发和规范一直是由Linux社区控制着，版本也是唯一的。操作系统的内核版本指的是在Linus本人领导下的开发小组开发出的系统内核的版本号。截至本文写作时，最新的稳定的内核版本号为2.6.37.4[6]。

Linux 内核实现了很多重要的体系结构属性。在或高或低的层次上，内核被划分为多个子系统[7]。Linux 也可以看作是一个整体，因为它会将所有这些基本服务都集成到内核中。这与微内核的体系结构不同,后者会提供一些基本的服务，例如通信、I/O、内存和进程管理，更具体的服务都是插入到微内核层中的。

Linux 内核的主要模块(或组件)分以下几个部分：存储管理、CPU 和进程管理、文件系统、设备管理和驱动、网络通信，以及系统的初始化(引导)、系统调用等。其中进程管理是最重要的一个模块。而进程调度是进程管理模块中解决如何使资源分配策略最优化的关键。在多用户多任务的操作系统中，进程调度问题是一个核心问题，它影响着操作系统的整体设计与实现和计算机系统整体的性能。

从二十世纪六十年代进程的概念由 J.H.Sallexer 等人提出以后，人们对进程和任务的组织与调度问题的研究一直是一个热点。Linux 操作系统之所以受到好评，是因为它的高效率很大程度上要归功于其内核进程调度系统的超凡设计。同时，我们又可以借助其开源特性，将最新的操作系统方面相关的思想、研究和技术融合于 Linux 操作系统中，通过修改其内核来个性化定制并进一步完善、优化它。近年来，基于 Linux 的进程调度研究比较活跃。文献[8]分析了 Linux 内核的任务调度流程,指出 Linux 内核的调度策略综合了时间片轮转和可剥夺式优先级两种调度策略。高珍等人[9]分析了 Linux 内核对 SMP 的实现方式。安智平、张德运等[10]设计了进程调度的 Master/Slave 模型，并考虑了该模型在 Linux 环境下的实现。

同时，还有部分研究致力于在 Linux 环境下实现现有的一些调度算法。如黄斌[11]等实现了操作系统进程调度的经典的多级反馈队列算法，改进了 Linux 内核的性能，降低了 Linux 系统执行任务的平均周转时间。

1.3 主要研究内容及论文组织结构

在接下来各章节，本文首先在第二章分析了操作系统的进程管理功能及主流的多核体系结构。其中，2.1 节介绍了进程、线程的概念及其在 Linux 内核中的实现；2.2 节介绍了当前几种主流的多核体系结构；2.3 节介绍了进程调度方面的相关理论。

第三章详细研究 Linux 内核调度系统的实现。其中，3.1 节介绍了 O(n)调度器；3.2 节介绍了 O(1)调度器；3.3 节介绍了完全公平调度器；3.4 节分析了 Linux 在多核调度方面存在的问题及改进思路。

第四章分析并设计了基于综合指标的 Linux 内核多核环境下的负载均衡系统。4.1 节介绍了为什么要进行负载均衡；4.2 节抽象出了描述 Linux 内核的负载均衡模型，并通过模型进行分析阐述；4.3 节探讨了几个在负载均衡设计时几个重要问题；4.4 节说明详细的算法设计；4.5 节对实验结果进行分析；4.6 节对本章进行小结。

第五章是总结与展望。

第二章 进程管理及其调度

2.1 基本概念

2.1.1 进程

进程（Process）的概念是在上世纪六十年代被提出的，最初是由 MIT 的 Multics 和 IBM 的 TSS/360 系统引用。至今人们从各方面对进程做出过许多种定义[12]。主要考虑了：

（1）进程的并发执行性（S. E. Madnick, J. T. Donovan）；

（2）进程作为独立的被系统调度的单位（E. Cohen, D. Jofferson）；

（3）进程的抽象性以及任务调度时作为系统分配和释放各种资源的单位（P. Denning）；

（4）进程与程序的区别。程序是行为规则的集合，程序的运行即体现为进程（E. W. Dijkstra）；

（5）进程是具体操作的序列（Brinch Hansen）。

以上关于进程的描述，尽管角度不同，但它们在实质上是相通的，即进程的动态执行性。因此，进程可被定义为可并发执行的程序对相关数据的一次具体的执行过程，是系统调配资源的单位。进程的基本特征有：并发性；动态性；独立性；异步性以及结构特征。

进程在并发执行过程中总是相互制约的。进程在其活动期间至少具备三种基本状态，它们是：执行状态、就绪状态和等待状态。进程的状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而转换。进程在执行期间，可以在三个基本状态之间进行多次转换。

2.1.2 进程在 Linux 内核中的实现

进程由操作系统创建。具体到 Linux 环境下，系统可以通过调用 fork()函数复制一个现有进程来生成新的进程[13]。调用 fork()函数的进程是父进程，复制出的进程是子进程。fork()函数返回后，父进程从返回点继续执行，子进程从

返回点独立运行。fork()函数在父进程和子进程的返回值不同。另外，fork()函数又是通过clone()函数实现的。为了使子进程执行新的任务，可以通过exec()相关函数装载新的任务。最后通过 exit()函数退出。exit()函数的作用是结束进程及释放分配给该进程的各种资源。父进程和子进程可以通过相关函数实现同步。如wait4()函数可用来查询子进程是否结束。进程结束后即成为僵尸进程，由其父进程通过wait()或waitpid()函数进行进一步处理。可以发现，Linux进程之间存在一个明显的继承关系。所有的进程都是PID(进程标识值)为1的init进程的后代。内核在系统启动的最后阶段启动init进程。

值得说明的是，Linux 内核通常把进程称做任务（task）。并使用类型为task_struct、称为进程描述符（process descriptor）的结构来描述进程[14]。该结构定义在<linux/sched.h>文件中。进程描述符包含一个具体进程的所有信息，主要有：

1、进程的状态（Process State）

进程从产生到结束期间会经历几种状态的改变。进程的状态是操作系统决定如何对其调度的重要属性。Linux环境下进程状态如表2-1所示。

表2-1　Linux进程的状态

内核表示	含义
R（TASK_RUNNING）	可执行状态
S（TASK_INTERRUPTIBLE）	可中断的等待状态
D（TASK_UNINTERRUPTIBLE）	不可中断的等待状态
T（TASK_STOPPED或TASK_TRACED）	暂停状态或跟踪状态
Z（TASK_DEAD - EXIT_ZOMBIE）	退出状态，进程成为僵尸进程
X（TASK_DEAD - EXIT_DEAD）	退出状态，进程即将被销毁

进程的状态在不同条件下可以相互转换，如图2-1所示：

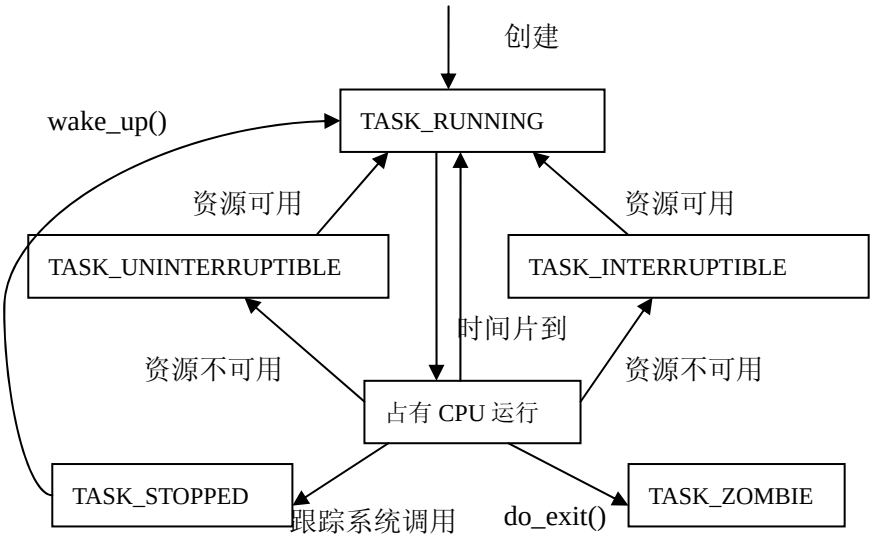

图 2-1 进程状态转化图

2、和进程调度相关的信息

这些信息一般反映了系统对进程调度的组织方式，比较重要的如进程的优先级，进程是普通进程还是实时进程。相关字段参见表 2-2 说明。

表 2-2 进程调度信息

域名	含义
need_resched	调度标志
Nice	静态优先级
Counter	动态优先级
Policy	调度策略
rt_priority	实时优先级

当 need_resched 字段被设置时，在“下一次的调度机会”就调用调度程序 schedule()。 counter 代表进程剩余的时间片，是进程调度的主要依据，也可以说是进程的动态优先级，因为这个值在不断地减少；nice 值代表静态优先级，反映进程拥有的时间片，影响 counter 值，可以用 nice()函数改变 nice 值；policy 代表了操作系统该进程的调度方式，如该进程是按照实时进程的方式被调度还是按照普通进程的方式被调度；rt_priority 是操作系统对实时进程进行调度时的重要依据。

表 2-3 说明了进程的调度策略类型。

表 2-3　进程调度的策略

名称	解释	适用范围
SCHED_OTHER	其他调度	普通进程
SCHED_FIFO	先来先服务调度	实时进程
SCHED_RR	时间片轮转调度	

只有 root 用户能通过 sched_setscheduler（）系统调用来改变调度策略。

3、标识符（Identifiers）信息

最基本的如进程标识符（Process Identifier），其他如用户标识符（User Identifier）、组标识符（Group Identifier）。

4、和进程通信相关的一些信息（IPC）

主要为了使进程在执行期间能够与其他进程进行信息交换。

5、进程链接信息（Links）

相关信息如表 2-4 说明。进程通过这些指针可以组织成一颗进程树。

表 2-4　进程链接信息

名称	中文解释［指向哪个进程］
p_opptr	祖先
p_pptr	父进程
p_cptr	子进程
p_ysptr	弟进程
p_osptr	兄进程
Pidhash_next、Pidhash_pprev	进程在哈希表中的链接
Next_task、prev_task	进程在双向循环链表中的链接
Run_list	运行队列的链表

6、和时间以及定时器相关的信息（Times and Timers）

进程的生存期（lifetime）是指该进程从产生到结束的这段时间。在此期间，内核要详细统计并更新进程使用 CPU 的时间，一般包括用户态执行时间和系统态执行时间。

有了“时间”的概念，可以实现进程的“定时”操作，即判断系统时间是否到达某个时刻，是否应该执行相关的操作。Linux 提供了许多种定时方式，用户

可以灵活使用这些方式来为自己的程序定时。

表 2-5　与时间有关的域

域名	含义
Start_time	进程创建时间
Per_cpu_utime	进程在某个 CPU 上运行时在用户态下耗费的时间
Per_cpu_stime	进程在某个 CPU 上运行时在系统态下耗费的时间
Counter	进程剩余的时间片

7、和文件系统相关的信息（File System）

用来存储进程对文件操作的信息，如访问文件的文件描述符等等。

8、虚拟内存相关信息（Virtual Memory）

进程通过自己的 mm_struct 数据结构描述自己独立的地址空间。

9、内存页面管理相关信息

当系统内实际内存分配不足时，Linux 内核内存管理模块会把某些页面搬移到硬盘等辅助存储器。

10、对称多处理器（SMP）信息

Linux 内核对 SMP 进行了全面的支持。

表 2-6　与多处理器相关的域

定义形式	解释
Int has_cpu	进程是否当前拥有 CPU
Int processor	进程当前正在使用的 CPU
Int lock_depth	上下文切换时内核锁的深度

11、和处理器相关的环境（上下文）信息（Processor Specific Context）

进程调度过程中需要保存上下文切换时的处理器现场信息。

12、与进程相关的其他信息

task_struct 结构是 Linux 内核进行进程管理的依据：首先 Linux 内核根据相关的调度信息判断某个进程是否需要调度以及如何调度；当该进程分配到 CPU 时，又需要根据处理器现场信息恢复进程执行的上下文环境；同时还要依据相关存储信息，查找指令和数据，实现进程间的通信等操作。由此可见，几乎所有的

操作都要依赖该结构，所以，task_struct 结构是一个进程存在的唯一标志。

2.1.3 线程及其实现

在传统的操作系统中，进程是系统进行调度和资源分配的基本单位，在任一时刻只执行一个控制流程，这就是单线程（结构）进程（Single Threaded Process）。然而随着并行技术、网络技术和软件设计技术的发展，研究人员提出了多线程（结构）进程（Multiple Threaded Process）的概念[15]，其思想是把“分配资源”与“被调度”这两项功能独立。进程仍然是操作系统分配资源的独立单位，可以适当避免由于进程被频繁调度而在进程间切换；线程来作为操作系统新的调度单位。可以说，进程实现了程序的并发执行，提高了系统效率；那么线程则可以有效减少系统开销，使多任务系统的并发性能更好。

线程作为可被调度执行的独立实体是进程的组成要素，若某个进程内包含有多个线程，那么该进程就是多线程进程。该进程中的线程共享操作系统分配给该进程的各种资源。多线程进程的内存布局如图 2-2 所示。

由于线程具有许多传统进程所具有的特征，所以，也把线程称为轻量进程 LWP（Light-Weight Process）。我们期望通过线程在操作系统和程序设计中来改善系统和应用程序的性能。

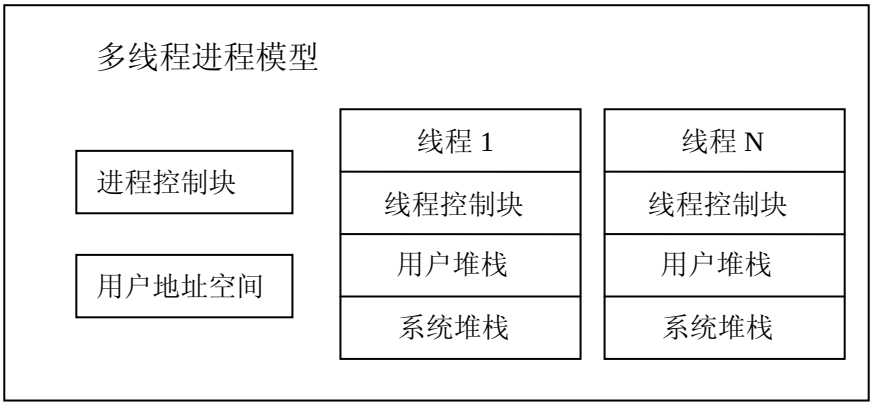

图 2-2 多线程进程模型

然而，在 Linux 环境下，我们并没有线程这样的结构。Linux 内核并没有对线程做什么特殊的对待，它把线程当做进程处理，也没有特殊的数据结构和调度策略专门为线程服务。线程同样用 task_struct 结构来描述并按照进程的管理和调度策略来对待。因此，线程在 Linux 环境下可认为是普通的进程。Linux 系统的这种处理方式区别于 Microsoft Windows、Sun Solaris 等系统的处理方式。

对 Linux 操作系统而言，线程并不是什么“轻量级进程”，而仅仅是共享系统内各种资源的一种手段。这一方面简化了线程的设计同时在调度方面，可以不用区分进程和线程，关于线程的调度实际上就是对进程的调度。

2.2 多核体系结构概述

2.2.1 指令级并行与线程级并行

当今主流应用的 x86 系列处理器均采用冯·诺依曼模型。该模型以计算为中心，采取存储程序式（Stored Program）的执行方式。CPU 通过对存储器的访问获取相关指令和数据。

在冯·诺依曼模型的框架下，处理器性能的提高主要依靠芯片指令级并行性（ILP，Instruction Level Parallelism）的提高。相关技术有超标量技术、多级缓存技术、指令预测技术、深度流水技术等等。

然而随着计算机应用的不断扩展，指令级并行技术已很难满足人们对计算机性能提升的需求，甚至成为发展的瓶颈。在这种现状下，线程级并行技术（TLP, Thread Level Parallelism）应运而生[4]。线程级并行比指令级并行层次更高。其核心思想是：在线程因相关资源的访问而发生等待时，由系统切换进其他的已就绪线程来占有 CPU 执行。线程级并行可以极大提高处理器的利用率，从而提高系统整体的性能。我们可以从图 2-3、图 2-4 中看出 TLP 相对于 ILP 的优势。传统的 ILP 方式下，单纯的提高 CPU 的运算速度，对于整个系统性能的提升所起到的作用微乎其微。TLP 可以将由于内存访问而导致的延迟经由导入其他线程的执行而抵消。

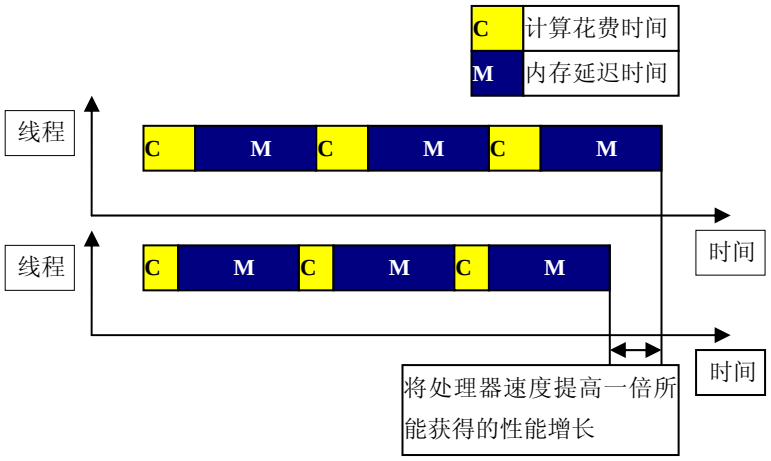

图 2-3 ILP 示意图

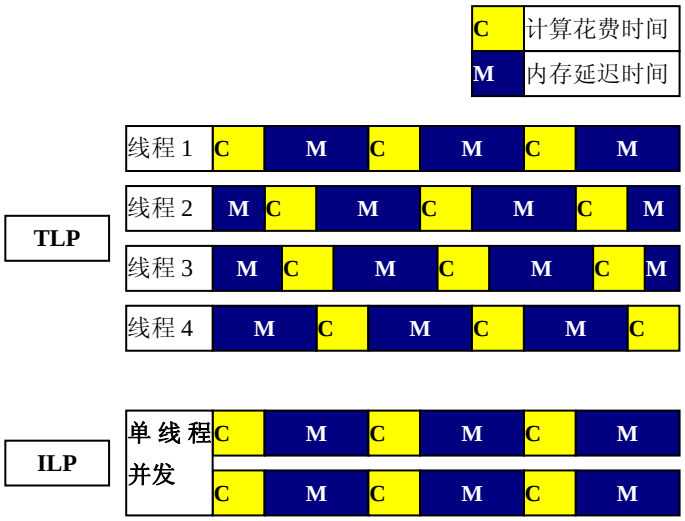

图 2-4 ILP 与 TLP 对比图

2.2.2 SMP、CMP、CMT 结构

SMP 的全称是”对称多处理”（Symmetrical Multi-Processing）技术，可以看作是一种从宏观角度支持 TLP 的体系结构技术。SMP 是指在一个系统内部集成对咯处理器，各处理器可共享存储器及系统总线，是一种应用十分广泛的并行技术。

然而，传统的 SMP 系统内部的多个处理器通过片外总线互连，共享总线带宽，传统总线所固有的低带宽、高延迟已经成为 SMP 系统的性能瓶颈。于是研究人员将 SMP 系统搬到一块芯片内部，利用片内的高带宽总线来代替片外总线，实现片内 SMP。

CMP 的基本思想即是在单个芯片上实现 SMP，每一个处理器核心实质上都是一个相对简单的单线程处理器。CMP 结构可支持多个线程同时在多个 CPU 核心上并行执行。CMP 主要是依赖 SMP 体系结构来实现 TLP 的。而 CMT，简言之，就是 SMP 系统为了支持线程级并行而在单芯片上融合多个多线程处理器。即 CMT=CMP+MT（MT 即 MultiThreading，多线程）。

而在我们考虑 Linux 内核进程调度问题时，并不需要过多考虑具体的物理实现细节，由于 CMP、CMT 等结构都是建立在 SMP 基础上的，我们以对 SMP 的支持表示对多核处理器的支持。

2.3 进程调度策略研究

2.3.1 进程调度及其性能评价

进程调度的主要功能是按照某种原则决定就绪队列中的哪个进程能获得处理器，并将处理器出让给它进行工作[15]。进程调度程序是操作系统最为核心的部分，执行十分频繁。进程调度策略优劣直接影响到整个系统的性能，因而，这部分代码要求精心设计，并常驻内存工作。

有两类进程调度方式：

第一类称剥夺方式：当一个进程或线程正在处理器上执行，若有另一个更高优先级或紧迫的进程或线程产生，则立即暂停正在执行的进程或线程，把处理器分配给这个更高优先级或紧迫的进程或线程使用。

第二类称非剥夺方式：一旦某个进程或线程开始执行后便不再出让处理器，除非该进程或线程运行结束或发生了某个事件不能继续执行。

剥夺式策略的开销比非剥夺式策略来得大，但由于可以避免一个进程或线程长时间独占处理器，因而，能给进程或线程提供较好的服务。对不同调度类型，相应的调度算法也不同，进程调度的核心是采用何种算法把处理器分配给进程或线程。

一个好的调度算法应该考虑很多因素，典型指标如：

- 资源利用率——使得 CPU 或其他资源的使用率尽可能高且能够并行工作，CPU 的利用率=CPU 有效工作时间/CPU 总的运行时间，而CPU 总的运行

时间为CPU 有效工作时间+CPU 空闲等待时间。

- 响应时间——交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间。使交互式用户的响应时间尽可能短，或尽快处理实时任务。
- 周转时间——批处理用户从作业提交给系统开始，到作业完成为止的时间间隔称作业周转时间，应使作业周转时间或平均作业周转时间尽可能短。这是批处理系统衡量调度性能的一个重要指标。
- 吞吐率——使得单位时间内处理的作业数尽可能多。
- 公平性——确保每个用户每个进程获得合理的 CPU 份额或其他资源份额，不会出现饿死情况。

当然，这些目标本身就存在着矛盾之处，操作系统在设计时必须根据其类型的不同进行权衡，以达到较好的效果。

2.3.2 进程调度的主要策略

进程调度主要有以下几种策略[15]：

1、先来先服务（FIFO）算法

先来先服务算法根据进程进入处理器运行队列的顺序来分配处理器。最早进入队列的进程首先被安排处理器执行，进程分配到处理器后将占据处理器直至进程结束或被阻塞，是一种非剥夺式调度。该算法简单易实现，但显然效率较低，不适合 I/O 操作比较频繁的进程的调度。

2、时间片轮转调度算法

时间片轮转调度是由调度程序为 CPU 运行队列的每个进程分配一个时间片（time slice）。当进程的时间片消耗完后，该进程必须让出处理器给下一个进程同时插入到队列中等待。时间片轮转调度算法的优点是可以避免某些进程长时间地占有 CPU 而另外的一些进程，如 I/O 进程无法获得执行机会。

时间片轮转调度算法是剥夺式调度，进程切换比较大，系统开销较高，这就要求合理的确定为进程分配的时间片的大小。决定时间片的大小需要考虑很多因素，如运行队列任务数、进程切换的系统开销、任务响应时间的要求等等。

3、优先级（数）调度算法

优先级调度算法的思想是为CPU运行队列的每个进程明确一个优先级，哪个进程的优先级最高，哪个进程就占有CPU执行。而进程优先级的决定可以考虑以下一些情况：交互式进程优先级高，这样可以保证该类进程的响应时间；对用户重要的进程其优先级高，这样可以保证尽快得到进程执行结果；进程等待时间越长，优先级越高，这样可以减少进程的执行时间等等。这些都是静态优先级的确定依据。效率高性能好的进程调度可采用动态优先数法，在创建一个进程时，根括进程类型和资源使用情况确定一个优先数，而当进程耗尽时间片或重新被调度时，再次计算并调整所有进程的优先数。基本原则是：①根据进程占有CPU 时间多少来决定，当一个进程占有CPU 时间愈长，那么，在它被阻塞之后再次获得调度的优先数就越低，反之，进程获得调度的可能性越大；②根据进程等待CPU 时间多少来决定，一个进程在队列中等待CPU 的时间愈长，那么，在它再次获得调度时的优先数就越高，反之，进程获得调度的可能性越小。基于优先数的低级调度算法可以按调度方式不同分为剥夺式和非剥夺式优先数调度算法。

4、多级反馈队列调度

多级反馈队列调度这种方法又称反馈循环队列或多队列策略。其主要思想是将就绪进程或线程分为两级或多级，系统相应建立两个或多个就绪队列，较高优先级的队列一般分配给较短的时间片。处理器调度每次先从高一级的就绪进程队列中选取可占有处理器的进程，同一队列中按先来先服务原则排队，只有在选不到时，才从较低一级的就绪进程队列中选取。进程的分级可以事先规定，例如，使用外围设备频繁者属于高级。在分时系统中可以将终端用户进程定为高级，而非终端用户进程为低级。进程分级也可以事先不规定，一个新进程进入内存后，首先进入高优先级队列等待调度执行，如能在该时间片内完成，便可以撤离系统；凡是运行超越时间片后，就进入低优先级就绪队列，以后给较长的时间片；凡是运行中启动磁盘或磁带而成为等待的进程，在结束等待后就进入中优先级就绪队列。多级反馈队列调度算法具有较好的性能，能满足各类用户的需要。

我们所研究的Linux内核在进程调度方面基本上还是根据优先数（级）把进程组织成多个调度队列，每个队列按照先来先服务的策略进行调度。同时，或者采用基于优先数的其他指标来确定进程执行的优先顺序，或者采用比队列更高级的数据结构来组织进程，如后面介绍到的红黑树。

2.3.3 多核处理器进程调度

在多核时代，操作系统的调度程序必须考虑多处理器的调度。显然，单个处理器的调度和多处理器的调度有一定的区别，现代操作系统往往采用进程调度与线程调度相结合的方式来完成多处理器调度。

1、同步的粒度

同步的粒度，就是系统中多个进程之间同步的频率，它是刻划多处理系统特征和描述进程并发度的一个重要指标。一般来说，可以根据进程或线程之间同步的周期（即每间隔多少条指令发生一次同步事件），把同步的粒度划分成以下 5 个层次：细粒度（fine-grained）；中粒度（medium-grained）；粗粒度（coarse-grained）；超粗粒度（very coarse-grained）；独立（independent）。

对于那些具有粗粒度和超粗粒度并行性的进程来说，并发进程可以得益于多处理器环境。如果进程之间交互不频繁的话，分布式系统就可以提供很好的支持，而对于进程之间交互频繁的情况，多处理器系统的效率更高。

无论是有独立并行性的进程，还是具有粗粒度和超粗粒度并行性的进程，在多处理器环境中的调度原则和多道程序系统并没有太大的区别。但在多处理器环境中，一个应用的多个线程之间交互非常频繁，针对一个线程的调度策略可能影响到整个应用的性能。因此，在多处理器环境中，主要关注的是线程的调度。

2、多处理器调度的设计要点

多处理器调度的设计要点有三个：为进程分配处理器、在单个处理器上支持多道程序设计和如何指派进程。

多处理器调度的设计要点之一是如何把进程分配到处理器上。设在多处理器系统中所有的处理器都一样，对存储器和输入输出的访问方式也相同，则各处理器可被放入一个处理器池（pool）。如果采取静态分配策略，把一个进程永久的分配给一个处理器，分配在进程创建时执行，每个处理器对应一个低级调度队列。这种策略调度代价较低，但容易造成在一些处理器忙碌时另一些处理器空闲。也可以采取动态分配策略，所有处理器共用一个就绪进程队列，当某一个处理器空闲时，就选择一个就绪进程占有该处理器运行，这样，一个进程就可以在任意时间在任意处理器上运行。对于紧密耦合的共享内存的多处理器系统来说，由于所有处理器的现场相同，因此，采用此策略时进程调度实现较为方便，效率也较好。

多处理器调度的设计要点之二是是否要在单个处理器上支持多道程序设计。对于独立、超粗粒度和粗粒度并行性的进程来说，回答是肯定的。但是对于中粒度并行性的进程来说，答案是不明朗的。当很多的处理器可用时，尽可能的使单个处理器繁忙不是那么重要，系统要追求的可能是给应用提供最好的性能，并非是让每个处理器都十分忙碌。

多处理器调度的设计要点之三是如何指派进程。在单处理器的进程调度中讨论了很多复杂的调度算法，但是在多处理器环境中这些复杂的算法可能是不必要的、甚至难以达到预期目的，调度策略的目标是简单有效且实现代价低，线程的调度尤其是这样。

3、多处理器调度算法

大量的实验数据证明，随着处理器数目的增多，复杂进程调度算法的有效性却逐步下降。因此，在以 Linux 为代表的大多数采取动态分配策略的多处理器系统中，进程调度算法往往采用最简单的先来先服务算法或优先数算法，就绪进程组成一个队列或多个按照优先数排列的队列。

多处理器调度的主要研究对象是线程调度算法。线程概念的引进把执行流从进程中分离出来，同一进程的多个线程能够并发执行并且共享进程地址空间。尽管线程也给单处理器系统带来很大益处，但在多处理器环境中线程的作用才真正得到充分发挥。

多个线程能够在多个处理器上并行执行，在共享用户地址空间进行通信，线程切换的代价也远低于进程切换。多处理器环境中的线程调度是一个研究热点，下面讨论几种经典的调度算法。

1）负载共享调度算法

负载共享（load sharing）调度算法的基本思想是：进程并不分配给一个特定处理器，系统维护一个全局性就绪线程队列，当一个处理器空闲时，就选择一个就绪线程占有处理器运行。这一算法有如下优点：

把负载均分到所有的可用处理器上，保证了处理器效率的提高。

不需要一个集中的调度程序，一旦一个处理器空闲，操作系统的调度程序就可以运行在该处理器上以选择下一个运行的线程。

运行线程的选择可以采用各种可行的策略(雷同于前面介绍的各种进程调度算法）。

Leutenegger，S．和 Vernon，M．分析了三种不同线程的负载分配算法：（1）先来先服务。用户进程到达时，它的所有线程被连续地排到就绪队列尾，依先后次序被调度执行；（2）最少线程数优先。共享就绪队列组织成一个优先级队列，如果一个用户进程包含的未被调度的线程数最少，则给它指定最高优先数，被优先调度执行；（3）有剥夺的最少线程数优先。刚到达的用户进程的线程数少于执行的进程的线程数时，前者有权剥夺后者。

这一算法也有不足：

就绪线程队列必须被互斥访问，当系统包括很多处理器，并且同时有多个处理器同时挑选运行线程时，它将成为性能的瓶颈。这也是 Linux2.4 内核的弊端之一。

被抢占的线程很难在同一个处理器上恢复运行，因此，当处理器带有高速缓存时，迁移或恢复高速缓存的信息会带来性能的下降。

如果所有的线程都被放在一个公共的线程池中的话，所有的线程获得处理器的机会是相同的。如果一个程序的线程希望获得较高的优先级，进程切换将导致性能的折衷。

2）群调度算法

群调度（gang scheduling）算法的基本思想是：把一组进程在同一时间一次性调度到一组处理器上运行。它具有以下的优点：

当紧密相关的进程同时执行时，同步造成的等待将减少，进程切换也相应减少，系统性能自然得到提高。

由于一次性同时调度一组处理器，调度的代价也将减少。

群调度算法针对多线程并行执行的单个应用来说具有较好的效率，因此，它被广泛应用在支持细粒度和中粒度并行的多处理器系统中。

3）处理器专派调度（dedicated processor assignment）算法

处理器专派调度算法的基本思想是：为每个进程分配一组处理器，这个进程的每个线程占有该组内的一个处理器执行直至进程终止。采用这一算法之后，这

些处理器将不适用多道程序设计，即该应用的一个线程阻塞后，该线程对应的处理器不会被调度给其他线程，而将处于空闲状态。

显然，这一调度算法追求的是通过高度并行来达到最快的执行速度，它在应用进程的整个生命周期避免进程调度和切换，且毫不考虑处理器的使用效率。对于高度并行的计算机系统来说，可能包括几十或数百个处理器，它们完全可以不考虑单个处理器的使用效率，而集中关注于提高总的计算效率。处理器专派调度算法适用于此类系统的调度。

最后值得指出的是，无论从理论上还是从实践中都可以证明，任何一个应用任务，并不是划分的越细，使用的处理器越多，它的求解速度就越快。在多处理器并行计算环境中，任何一种算法的加速比提高都是有上限的。

4）动态调度（dynamic scheduling）算法

在实际应用中，一些应用提供了语言或工具以允许动态地改变进程中的线程数，这就要求操作系统能够调整负载以提高可用性。

动态调度算法得核心是由操作系统和进程协作实现进程调度。操作系统的任务时是在待执行的进程间分配处理器。而进程在分配给它的处理器上执行可运行线程的子集，哪一些线程应该执行，哪一些线程应该挂起完全是应用进程自己的事（当然系统可能提供一组缺省的运行库例程）。相当多的应用将得益于操作系统的这一特征，但一些单线程进程则不适用。

调度过程中，当新进程到来或进程要求分配处理器时，调度器作如下处理：

若有处理器空闲，则分配给进程。若没有，从分配出去的处理器中回收一个分配给新进程。

若无法满足进程申请处理器的要求，则等待到有处理器可以分配或进程的申请主动取消。

当处理器被释放时，按照先来先服务的原则将新释放的处理器分配给那些等待处理器的进程。

第三章 Linux 内核任务调度系统研究

进程调度是操作系统的核心功能。从 Linux 2.4 到 Linux 2.6，内核进程调度程序历经数次改进（Linux 内核的修订与改进是相当频繁的），每一次内核调度程序的改善都使内核的性能上升到更高层面。

3.1 O(n)调度器

O(n)调度器是指 Linux 2.4 版本内核所采用进程调度程序，基于静态优先级实现[16]。该调度器为系统内所有 CPU 维护一个全局运行队列，调度时每次为该队列中优先级最高的那个进程分配 CPU 执行。

进程在产生时会得到一个时间片。当进程占有 CPU 后，其时间片随着进程的运行逐渐减少至零。当该进程的时间片已消耗完，该进程就必须让出系统分配给它的 CPU。当所有在运行队列中的进程的时间片都消耗完后，由内核重新分派时间片。

O(n)调度器的性能受全局运行队列中的任务数量约束。就绪进程越多，查找下一个要运行的进程耗时越长。系统的就绪任务越多，O(n)调度器的效率就越低。而且，给进程分配多长的时间片才合适也很难判断。同时，由于系统内只有一个全局的运行队列，该调度器不适合在多核体系结构下应用。任意一个处理器调度时都需要访问这个全局唯一的运行队列，这样就必须加锁以控制各处理器因访问队列而产生的竞争，而各处理器对锁的争用又会产生新的系统性能瓶颈[17]。

3.2 Linux 内核 O(1)调度器

为了改进 O(n)调度器，使系统性能在就绪队列拥有大量进程的情况下有所提升，Linux 内核调度器的设计者 Ingo Molnar 提出并实现了 O(1)调度器[18]。O(1)调度器在进程调度上所需要的时间是恒定的，和就绪队列的任务数没有关系。该调度器效率很高，得到了广泛应用。Linux2.6.23 内核版本之前各版本均采用 O(1)调度器。该调度器还被集成到 Linux 2.4 内核中以改进调度性能。

3.2.1 运行队列 runquene 数据结构说明

运行队列结构 struct runqueue 是 O(1)调度器中一个关键的的数据结构，它主要用于存储每个 CPU 的就绪队列信息。限于篇幅，本文在此只说明 struct runquene 最重要的数据成员：prio_array_t *active，*expired。

这几个成员是 runqueue 中最重要的内容。在 Linux 内核中，每个 CPU 只维护一个和它一一对应的就绪队列 rq，但 rq 又根据时间片的使用区分为 active 和 expired 两个队列。active 队列是就绪队列中时间片尚未消耗完的就绪进程组成的队列；expired 队列是就绪队列中时间片已消耗完的就绪进程组成的队列。active 和 expired 指针都是 prio_array_t 类型。由

```
typedef struct prio_array prio_array_t;
```

知 prio_array_t 类型实际上是 prio_array 类型。数据结构 prio_array 的定义如下：

```
struct prio_array{
  unsigned int nr_active;
  unsigned long bitmap[BITMAP_SIZE];
  struct list_head queue[MAX_PRIO];
};
```

其中：BITMAP_SIZE=5，MAX_PRIO=140。

MAX_PRIO 定义系统拥有的优先级个数，其默认值为 140。每一个优先级在内核中都拥有与自己对应的运行队列。同时内核还维护一个优先级位图 bitmap，其作用是提高查找当前系统中拥有最高优先级的可执行进程时的效率。优先级位图各个位的初值为零，只有当某个优先级的运行队列不为空时，响应的标志位才为 1。优先级数组的示意图如图 3-1 所示。

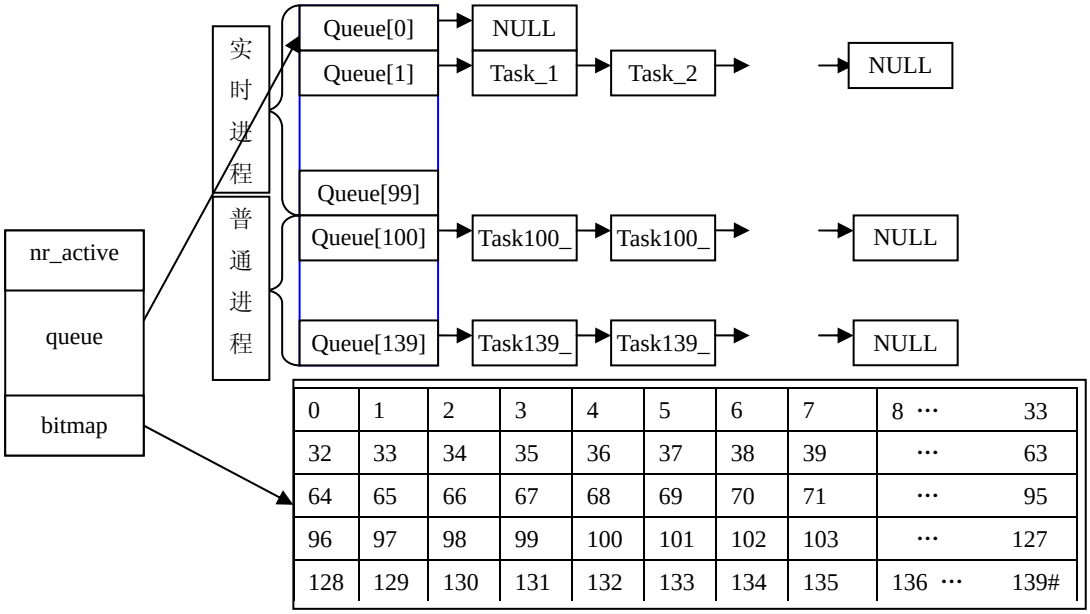

图 3-1 优先级数组示意图

3.2.2 O(1)调度器主函数 schedule() (kernel/sched.c)

schedule()函数是用来挑选出下一个应该被调度执行的进程，并且完成进程执行时的 CPU 上下文切换的工作。schedule()函数是进程调度的主要执行函数，它的性能将直接决定 Linux 内核的性能，包括两个重要数据：prev 和 next。其中，prev 当前正在运行的进程，也就是即将被切换出 CPU 的进程；next 下一个就绪进程，也就是即将被切换进 CPU 的进程。进程调度过程如下：

a. 如果队列 rq 的 active array 中进程数为 0，则将 active array 和 expired array 进行切换。这一处理方式是 O(1)调度器的核心所在。仅需要以下三行代码实现指针互换：

```
array = rq->expired; //存储过期队列指针
rq->expired = rq->active; //活动队列变过期队列
rq->active = array; //过期队列变活动队列
```

b. 内核使用函数sched_find_first_bit()找到最高优先级进程队列的偏移量idx， queue[idx]->next 即为所找的next，代码如下：

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue+idx;
next = list_entry(queue->next, task_t, run_list);
```

Linux内核什么时候调用schedule()函数，如何调用也是一个关键问题。schedule()函数有两种调用方式：一种是主动调度，由内核直接调用函数schedule()，如当进程退出，或转入睡眠状态时。另一种是强制调度，将当前进程task_struct里面的need_resched字段的值设为1。那么当进程从内核态返回用户态的时候内核将检查这个字段，如果发现已经被置位，内核将调用schedule()函数；有以下三种情况可能会设置task_struct结构的need_resched字段：

1、时钟中断程序发现进程用完分配的时间片，需要切换出CPU的时候；

2、当一个睡眠进程被唤醒时，如果发现该进程比当前正在CPU上运行的进程更有资格占有CPU；

3、系统调用引起的进程的调度策略、nice 值的改变等。

3.3 CFS 调度器

自 Linux 2.6.23 版本始，内核引入了完全公平调度器(CFS，Completely Fair Scheduler)[19]。CFS 不再使用各种复杂的计算公式对进程进行划分，而是对所有的进程平等对待。完全公平调度器的设计思想是在实际环境中模拟出理想的多任务处理环境。所谓理想的多任务处理环境，是指无论系统中存在多少待运行的任务，所有任务的执行时间都按照同一规则来决定。当然这一点在真实的物理环境中是无法保证的。CFS 会对运行时间过长的的进程做出惩罚，尽量保证实现进程间分配运行时间的公平性。CFS 调度器整体结构如图 3-2 所示：

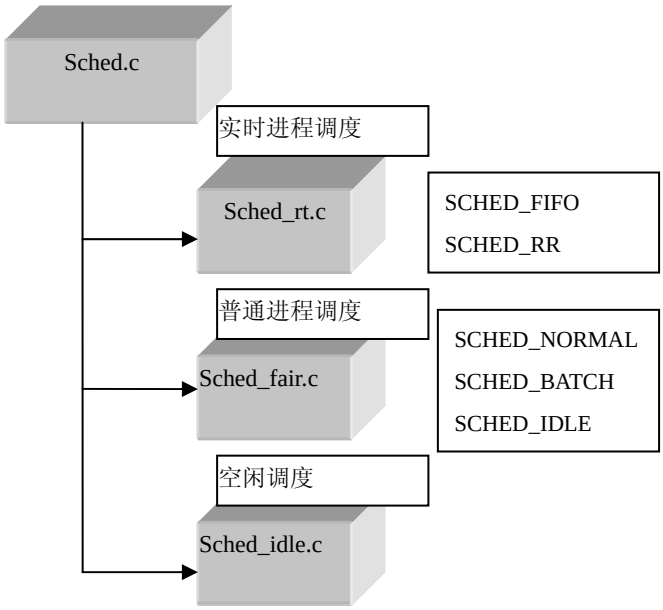

图 3-2 CFS 调度器示意图

CFS 调度器在全局上依旧将进程区分为实时进程和普通进程。该调度器只改进了对普通进程的调度策略，它淘汰了以前调度器采用的优先级队列的组织形式，不再使用 active 队列和 expired 队列，而是代之以红黑树结构组织进程，红黑树的每个节点的键值是调度器定义的虚拟运行时间(vruntime)。实际上，虚拟运行时间 vruntime 也是由优先级间接决定的。树中所有进程均保持各自独立的虚拟运行时间（vruntime），进程在红黑树中的位置由其虚拟运行时间值决定，红黑树最左边的叶子节点是虚拟运行时间最小的进程。调度时，该红黑树最左边的叶子节点首先被摘下，移除出树并获得 CPU。执行一段时间后，由于 vruntime 可能得到更新而使当前正在运行的进程放弃 CPU。因为由于 vruntime 的增加，该进程就有可能不再拥有最小的虚拟运行时间，从而被放置在红黑树相对偏右的节点。这种情况下就应该选择现在虚拟运行时间最小的那个进程来执行，这就实现了调度器的进程切换。Vruntime 能够反映进程的优先级，对于优先级越高的进程，其虚拟运行时间增长得就越慢，就越可能对应靠左的叶子节点，就有更大的概率获得处理器。通过这种方式 CFS 就间接实现了基于优先级的调度方式。如图 3-3 所示：

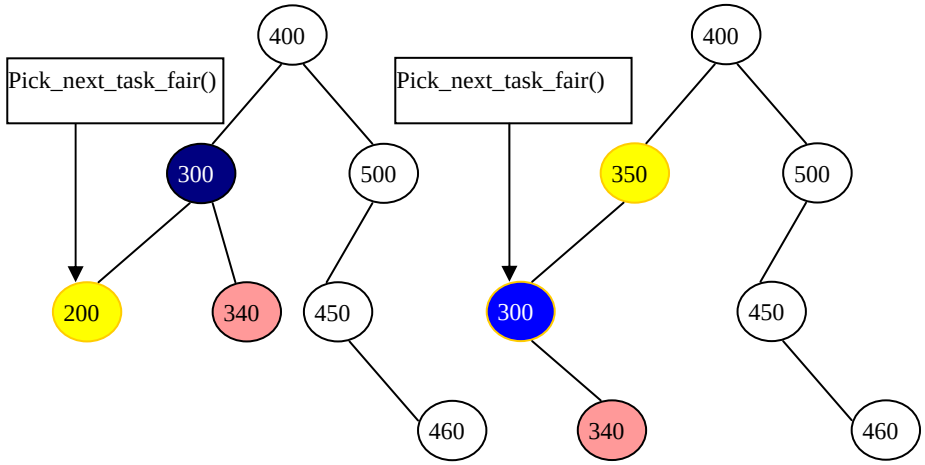

图 3-3　进程红黑树示意图

CFS 对红黑树结构主要有两种基本的操作。一种是查找红黑树中最靠左边的叶子节点，将相应进程其移出红黑树；另一种是将从 CPU 上切换出的进程插入到红黑树中。这两种操作的时间复杂度为 O(log n)。分析该调度器，似乎没有之前的调度器性能好。但通过测试发现，任务较多时，完全公平调度器的调度时延还是相对比较合理的，与 O(1)调度器的差距不大。

3.3.1 完全公平调度相关结构说明

内核中的任务在CFS中还是由数据结构task_struct描述，但该结构不再是调度的对象，CFS的调度对象是是task_struct中一个新的结构成员sched_entity。数据结构Sched_entity包含两个重要数据，一个是虚拟运行时间vruntime，另一个就是与进程优先级有关的load_weight。Load_weight计算如下：设优先级120对应的load_weight值是1024，当优先级每增加1，进程load_weight相比要增加25%。

在CFS调度器中，我们不只拥有了新的调度对象sched_entity，而且拥有了新的完全公平调度运行队列cfs-rq，该队列维护load_weight，组织调度对象，存储虚拟运行时间最小的调度对象的地址。同时cfs-rq也是其上层完全公平调度运行队列的调度对象，接受上层调度。也就是说，在CFS调度器中，被调度的对象既包含具体的任务也包含队列cfs-rq。

3.3.2 完全公平调度相关函数说明

3.3.2.1 Schedule()函数

Schedule()函数是Linux内核进程调度器的核心。该函数首先从任务队列中选取一个任务，如果该任务尚未占有CPU，那么接着完成处理器执行环境的上下文切换工作，让被选择的任务占有CPU。由于该过程需要修改进程的虚拟运行时间，schedule()函数通常要将运行的进程移回到红黑树中，接下来调用pick_next_task_fair()函数。该函数摘取红黑树最左面的叶子作为下一个待执行的任务。Schedule()函数一般在时钟中断或相关函数结束时被调用。

3.3.2.2 Pick_next_task_fair()函数

Pick_next_task_fair()函数的功能简单而言就是选择红黑树最左边的叶子，将其作为下一个分配CPU的任务。需要注意的是，该函数还要解决内核调度的层级结构问题以及记录进程运行时相关的时间参数等等。

3.3.2.3 Scheduler_tick()函数

Scheduler_tick()函数的执行频率最高。它需要解决的问题是：记录CPU队列的时间信息，修改正在运行的任务的虚拟运行时间，最后还要争取实现负载平衡。其中，最主要的任务是更新调度实体的虚拟运行时间，因为vruntime直

接关系到调度实体在红黑树中的节点位置。

3.3.2.4 Task_tick_fair()函数

该函数的任务是修改调度对象的虚拟运行时间（vruntime）。首先修改变量delta_exec 的值保存调度对象的累计运行时间，接着根据这个累计运行时间计算 delta_exec_weighted，最后在调度对象的虚拟运行时间上加上delta_exec_weighted。

3.3.2.5 place_entity()函数

place_entity()函数需要解决任务执行过程中的状态转换问题。任务由休眠状态变为执行状态时，由于其睡眠期间虚拟运行时间得不到更新而一直偏低，就会造成该任务持久占有 CPU 运行。这就需要调度器做额外处理。CFS 调度器必须对该进程的虚拟运行时间做出调整，以实现进程调度的公平。处理方式如下，调度器首先获取 CPU 运行队列中最小虚拟运行时间的值（min_vruntime），然后将缺省的理想时间片 20ms 根据该进程的 load_weight 转换成相应的虚拟运行时间 vruntime。进程的 vruntime 被设置为这两者之差。

3.4 Linux 内核对 SMP 的支持

在 SMP（Symmetric Multi-Processor Architecture），也就是对称多处理体系结构中， 多个 CPU 之间的地位都是平等的，不分主次，共享内存和外设。但为了减少内存访问的冲突，各 CPU 均拥有彼此间相互独立的高速缓存 cache，它们彼此间也可以在需要的时候互相通信，交换数据。与传统的单处理器（UP）相比，Linux 内核对 SMP 体系结构有单独的代码支持，这部分代码被独立编译。

在单 CPU 环境下,某一时刻只有一个进程在运行。而在对称多处理的体系中，可能有多个进程在同一时刻并行执行，因此，Linux 内核需要额外的数据来记录处理多核环境下的运行情况。首先进程的 task_struct 结构通过预编译的方式扩充了一个 oncpu 字段。

oncpu 字段的含义是表征某个人物是否正占有 CPU 运行， 1 表示正在执行，0 表示进程尚未运行。只有在 oncpu 字段为 0 时，进程才可被调度。Linux 内核对 SMP 的支持方式是通过函数 task_running()来判别某个任务的状态：

```
static inline int task_running(struct rq*rq,struct task_struct*p)
        {
         #ifdef CONFIG_SMP
         return p->oncpu;
         #else
         return rq->curr==p;
         #endif
        }
```

3.4.1 对称多处理环境下的进程调度分析

内核调度器通过两个函数 try_to_wake_up()以及 wake_up_process()唤醒相关任务。第一个函数首先设置任务的状态，将其改变为 TASK_RUNNING，接着将该任务放入本地 CPU 的运行任务队列，同时通过 active_task()函数改变任务的优先级。这又是通过 recalc_task_prio()函数实现的。该函数可以维护任务的动态优先级及其平均睡眠时间。若新任务获得更高的优先级，那么设置其进程描述符 task_struct 的 need_resched 字段为 1。

在 Linux 用于 SMP 体系结构的进程调度程序中，对每个被剥夺 CPU 的非空闲进程，内核尽量去安排一个空闲的 CPU 让其继续运行。值得注意的是，由于在某些多核体系结构中，一级缓存是集成在各个核内的，即每个核都有自己的 L1 Cache。此时，将进程从某个核心迁移到另一个核心将会带来一定的开销。这些特殊情况在考虑任务的调度问题时必须解决。

3.4.2 多个 CPU 运行队列之间的负载平衡问题

多核环境下，只有多个 CPU 间的任务分配相对均衡时，系统才能有较好的性能，这就必须考虑多核结构中多个 CPU 的拓扑结构。为此，Linux 内核采取一种

基于“调度域”的均衡算法。内核每隔一段时间检查运行队列的进程数量是否均衡，并在负载不均时将某些进程从一个 CPU 的运行队列转移到另一个 CPU 的运行队列中去。

在 Linux 内核中，调度域是 CPU 的集合，其负载应当由内核来维持一种相对平衡的状态。调度域采用一种层级结构的表现方式。调度域是系统拥有的全体 CPU 的子集，调度域又可包含子调度域。同时，调度域也可进行进一步的划分，分割成一个或多个调度组，调度组是其所从属的调度域包含的全体 CPU 的子集，所谓的负载平衡问题即是在某调度域的各个调度组之间进行负载重新分配均衡。若调度域中存在某个调度组的任务负载小于另外的调度组的负载，那么就需要在它们之间进行的任务数量的平衡。调度域和调度组分别用 sched_domain 结构和 sched_group 结构来描述。

多核系统内的任意一个 CPU 都从属一个“基本”的调度域，只要系统预编译执行定义了 CONFIG_SMP 宏。Linux 内核一般调用 cpu_sched_domain(i)函数以及 this_sched_domain()宏来操作调度域。其层级结构通过指针->parent 来组织。->parent 指针要保证以 NULL 结束，而且要为每个 CPU 分配调度域以便于 CPU 的增加。

每一个调度域都包含几个 CPU（由->span 指针来组织）。子调度域的范围不能超出上层调度域的范围。最顶层的调度域将包含系统中所有的 CPU。

每一个调度域都包含几个 CPU 调度组（struct sched_group），调度组之间通过其->groups 指针构成一个单向循环列表。某个调度域所包含的所有调度组的 cpumask 都一样，但两个不同的调度组之间的 cpumask 不能相交。->groups 指针所指向的组必须包含本域所拥有的 CPU，而不能是其他调度域中的 CPU。

我们实现的负载平衡是在某个调度域范围内以调度组为单位的负载平衡，即在调度组之间协调其任务量。这样调度组就是我们的调度对象，其负载是其所包含的全部的 CPU 的负载的总和。只有某个调度组的负载破坏系统整体的均衡情况时才会在调度组间迁移进程。

系统内每个 CPU 都按一定周期执行 run_rebalance_domains()函数，该函数在 kernel/sched.c 内核源文件中。该函数在相关 CPU 的基础调度域内判断现在是否应该尝试进行负载均衡。若是则调用 load_balance()函数进行负载均衡。

load_balance()函数会判断当前调度域的负载情况，查找最繁忙队列，寻找合适迁移的进程。如果当前调度域不需要进行负载均衡，那么调用run_rebalance_domains()函数，该函数将向上检查父调度域，进而是父调度域的父调度域，如此下去直至parent指针为空。

3.4.3 Linux内核的CPU亲和力特性

Linux内核的CPU亲和力(afinity)是指任务尽量长时间地占用某个CPU运行而不被迁移到其他处理器的特性。

在描述任务的task_struct结构中，cpu_allowed字段与CPU亲和力紧密相关。这个字段有n位，与系统中n个逻辑处理器一一对应。要使某个任务在指定的CPU上运行，就需要设置该任务的cpu_allowed字段的相对应的数据位为1。因此，如果要使某个任务能在任意一个CPU上运行，并且能够在CPU之间进行迁移，那么该任务在该字段所有数据位上就应该全是1。这也是Linux内核中进程的cpu_allowed字段的默认情况。要想改变cpu_allowed字段的值，可以通过调用sched_set_affinity()函数以及sched_get_affinity()函数实现。但值得注意的是，cpu_affinity 会被父进程传递给子进程，因此应当适当地调用sched_set_affinity。

CPU亲和力特性在某些情况下可以提高系统的性能。如果某任务被迁移到其他CPU上去执行，就会影响系统缓存的性能和效率。因此，当多个任务要存取同样的数据时，可以将这些任务全部安排到某一个事先指定的CPU上来执行，从而提高缓存数据的使用效率。

3.5 Linux 进程调度分析及改进思路

综合分析Linux内核的进程调度系统我们可以看到，内核调度系统的每次升级，都带来了系统性能的极大提升，但同时我们也发现Linux内核的进程调度算法在以下几个方面还有待改进：

1、Linux 针对实时进程的调度不能保证及时的响应，特别是在一些嵌入式应用的场合。针对Linux内核对实时进程采取的静态调度策略，目前的研究思路是采取动态调度策略如EDF算法、LSF算法等对其实时进程调度进行优化[20][21]。

2、Linux 的调度算法是一种强调公平的算法，但这种公平是从进程角度而言，并没有考虑到用户需求的差异。一方面，用户提交的任务数量的差异在极端情况下会造成极度的不公平。如用户 A 提交了 1 个任务，用户 B 提交了 99 个任务，那么根据 Linux 内核的调度系统，用户 A 只能享受到系统 1%的资源来处理该任务，而 99%的资源都分配给了 B，此时用户 A 可能得不到及时有效的相应，这对用户 A 显然是不公平的。目前国内外有很多研究致力于解决该问题。如吴凌俊、李之棠等[22]提出的基于 Linux 的用户公平分享调度等。另外，针对 Linux 内核的 CFS 调度，已有相应的补丁包（Patch）可以在一定程度上解决此问题，如 CFS 组调度策略[23]补丁包。

3、Linux 内核在多核环境下进行系统负载均衡时，只考虑 CPU 运行队列的进程数这一个指标[24]。固然采用该指标在实现负载均衡时相对简单一些，但同时也应该考虑不同类型的进程平衡时的系统开销问题。因此，为实现均衡而不得不迁移进程时必须考虑进程本身的特征：

1）进程占用的内存资源的大小。迁移一个比较大的进程会极大的增加系统的开销从而影响系统的性能；

2）进程的类型。如果待迁移的进程是 CPU 受限型，那么将其调度到其他 CPU 可以提高系统并行的效率；如果待迁移的进程是 IO 受限型，那么即使将其迁移到其他 CPU，也不能有效地提高系统的效率。

3）进程之间的联系。如果两个进程之间通信频繁，那么显然将它们分配在同一个处理器上工作会更好。

另外我们还应该看到当负载不均衡的状况不是很严重时，应该避免花费过多的系统开销用于均衡负载，从而避免由于任务在系统各个 CPU 之间频繁的迁移降低系统的效率。为解决这些问题，本文在第四章提出了一种基于综合指标的进程负载均衡模型及算法，并通过修改内核相关数据结构及函数实现。

第四章 面向多核 CPU 负载均衡的 Linux 内核进程调度模型

4.1 负载均衡的目的、意义与途径

负载均衡主要目的还是为了能够在整体上缩短任务的平均响应时间，提高系统性能。具体而言就是希望能够比较均匀地、充分地利用整个系统的资源。任务的平均响应时间由很多因素决定，其中一个关键因素即是系统需要执行的任务量，任务量分配越均匀，资源利用就越合理，整体上系统效率就越高，就会缩短任务的执行时间。在考虑实现系统的负载均衡时，必须要解决的问题主要有以下几个：负载均衡调度的启动时间、待转移任务的选择和转移数量以及确定均衡调度的源点和终点。

负载不均衡是导致任务执行因空闲等待而延迟的重要因素之一。可以说操作系统的负载均衡策略在很大程度上决定着系统整体的性能指标。而在具体进行负载平衡时，既要解决任务因分配不合理而造成的延迟问题，又要注意均衡负载的系统开销。必须综合考量，探索最优解决思路。

前人的大量相关研究给我们总结了许多宝贵的经验[25] 26] 27]：

1)负载相对比较均衡的系统的性能要好得多。

2)状态依赖的负载均衡策略开销较大，但比概率论策略性能要好。

3)概率论策略不能充分反映系统的动态改变。

4)进程迁移的目的 CPU 对负载均衡的影响比较大。

5)进行负载均衡不需要太多的信息，负载指标过多反而可能会降低负载均衡的效果。

6)负载均衡依据的状态信息必须是系统当前的即时信息。

从负载均衡的实现方式来看，其策略又可分为静态负载均衡和动态负载均衡。静态负载均衡主要是对静态任务划分的研究。这个过程可以通过遗传算法（Genetic Algorithm）、启发式算法（Heuristic Algorithm）等算法的应用来实现。而要实现动态负载均衡则可以应用概率论策略以及近邻契约算法等算法[28]。

具体分析负载均衡问题，其影响因素可能很多。Kim[29] 等人认为实现均衡

可以经历 4 个阶段：对系统负载程度的估计、对均衡行为产生的收益的估计、适合迁移的任务的判别选择和具体的任务迁移过程。李冬梅、史海虎等[28]通过深入研究，针对分布式系统的负载均衡问题设计了一个通用模型。该模型采用四维向量来描述，具体有网络环境，任务集合，负载评价方式及指标，调度算法等。陈华平等[30] 设计的调度模型分析了以下问题：负载均衡的时机、进程迁移的起点和终点、待迁移进程的判别选择。

4.2 面向多核 CPU 负载均衡的 Linux 内核进程调度模型

在深入分析各种调度模型以及现有的各种负载均衡调度策略的基础上，整体考虑影响系统负载均衡的各个要素，我们可以将 Linux 内核的负载均衡调度模型用四维向量<E,T,L,S>来表示。其中：

E(Environment)：代表运行环境；

T(Task)：代表系统的任务集；

L(Load)：代表系统的负载；

S(Strategy)：代表系统的负载均衡策略。

4.2.1 运行环境分析

运行环境 E 可以用一个三维向量(SE，SD，SG)来描述。

SE(System Environment)，即系统环境，需要描述各节点的性能特征及其物理上的拓扑结构。如多核环境是同构还是异构？ SMP 至少包含两个相同的处理系统，每个处理系统的芯片、核心都一样。各个处理单元可共用内存及缓存。Linux 一直采用对称多处理，与其他 CPU 相比，所有处理单元地位相同，内核不应该偏向其中任何一个。

SD(Scheduling Domain)，即调度域，指根据运行环境的特点及负载均衡策略的不同所采取的逻辑上的拓扑结构。

SG(Scheduling Group)，即调度组，指某调度域内的划分出的 CPU 子集。调度组是我们考虑负载均衡调度的基本单位，负责处理具体的调度行为。调度程序实现的是某调度域的调度组之间的任务均衡。当某调度域中的每个调度组都只含有一个 CPU 时，即是在讨论实现 CPU 间的负载均衡。

4.2.2 任务分析

考虑到用户提交的任务的不同，参与调度的任务可以用二维向量(TT，TC)来表示。TT(Task Type)，即系统待执行的任务类型，Linux 根据调度策略的不同从系统层面上将任务划分为普通进程和实时进程。TC(Task Constraints)，代表任务的约束条件。譬如，实时进程要先于普通进程运行，对实时进程和普通进程的调度要采用不同的优先级算法。

4.2.3 系统负载分析

衡量分析系统的负载可以使用一个五维向量(P，F，T，S，M)来表示。

1、P(Parameter)，即负载指标。负载指标的选择与对于分析系统的负载均衡问题非常关键。负载指标是对系统负载程度进行量化的依据。选择不同的负载指标对负载均衡的策略和效果影响非常大。目前衡量负载程度的指标主要有：系统任务数、CPU 利用率、内存占用率、平均响应时间、尚未完成的任务数、可用资源的数量、到达的任务数等等。根据情况，我们可以选择某个或者综合采用多个指标对系统负载做出衡量。因此，负载指标应该是多个因子的集合。

选择合适的指标一般应从以下几个角度考虑：

获得指标的代价低，这允许对该指标的频繁测量以确保反映系统负载的即时情况；

能体现产生争用的所有资源上的负载；

各个负载指标在计算及衡量上应该彼此独立。

Linux 内核采用运行队列任务数以及前文介绍的 raw_weight_load 作为衡量系统负载的指标。如果用 rq 指向运行队列，用 p 指向系统中的任务，则

```
for(int i=0;i<nr_running;i++){
  rq->raw_weighted_load+=p->load_weight;
  p=p->next;
}
```

任务的 load_weight 计算方式如下：

对内核中的迁移任务

```
p->load_weight=0;
```

对实时任务

```
p->load_weight=RTPRIO_TO_LOAD_WEIGHT(p->rt_priority);
```

对普通任务

```
p->load_weight=PRIO_TO_LOAD_WEIGHT(p->static_prio);
```

2、F(Function)即负载函数。设 R 为实数集，则负载函数可表示为：

$$F=F(P),\ P\in R \qquad (4\text{-}1)$$

若使用 CPU 就绪进程数 nr 作为单一指标，那么 Linux 内核的负载函数可以写成：

$$F=F(nr),\ nr\geq 0 \qquad (4\text{-}2)$$

如前所述，调度组是执行调度基本单位，当某调度组的负载超出均衡时才需要进行进程的迁移，因此负载函数并不依赖于某个 CPU 上的就绪进程数，而是依赖于该组所有 CPU 的就绪进程数即 nr_running 之和与 raw_weight_load 之和，设某调度组拥有 n 个 CPU，则该调度组的负载函数可表示为：

$$F=F(sum\_nr\_running, sum\_weight\_load) \qquad (4\text{-}3)$$

其中，$sum\_nr\_running=\sum_{1}^{n} nr\_running$，

$$sum\_weight\_load=\sum_{1}^{n} raw\_weight\_load$$

3、T(Threshold)即负载阈值。负载阈值一般是一个数对$<\beta_1, \beta_2>$，其中$\beta_1, \beta_2\in R$，$0<\beta_1\leq\beta_2$。通过计算调度组的调度函数 F 的值并与与阈值β_1和β_2比较得出其负载状态。

Linux 内核在进行负载调整平衡时，通过 update_load()函数更新系统目前的负载状况，通过 find_busiest_group()函数判断哪个调度组的负载最大。

1)sum_weighted_load 变量的最大值：

```
avg_load=(SCHED_LOAD_SCALE*total_load)/total_pwr;
```

相关变量的计算如下：

```
unsigned long total_load=0;
unsigned long total_pwr=0;
unsigned long avg_load=0;
unsigned long load;
int load_group;
struct sched_group*group=sd->groups;
do{
 int local_group=cpu_isset(this_cpu,group->cpumask);
 for_each_cpu_mask(i,group->cpumask){
  if(local_group)
   load=target_load(i,load_idx);
  else
   load=source_load(i,load_idx);
 avg_load+=load;
 }
 total_load+=avg_load;
 total_pwr+=group->cpu_power;
 group=group->next;
}while(group!=sd->groups);
```

2)sum_nr_running 变量的最大值：

```
group_capacity=group->cpu_power/SCHED_LOAD_SCALE;
```

Linux 的调度组通过 cpu_power 变量的值来描述该调度组包含的 CPU 的计算能力，用于在不同的调度组间平衡负载时使用。一般在 SMP 结构中某调度域中所有调度组的 cpu_power 都一样。cpu_power 可以间接反映该调度组除本身负载外还可以执行的任务数。

4、S(State)即负载状态。负载状态一般有过载（over load）、适载（suitable load）、轻载（light load）、空载（idle）四种。负载状态与阈值 T 密切相关。

$$S=\begin{cases}\text{过载，if } F>\beta_2\\ \text{适载，if } F>\beta_1 \text{ and } F\le\beta_2\\ \text{轻载，if } F>0 \text{ and } F\le\beta_1\\ \text{空载，if } F=0\end{cases} \quad (4\text{-}4)$$

5、M(Modifying Factor)即负载状态的修正因子，用于修正负载阈值。

4.2.4 负载均衡策略分析

负载均衡策略是一个二维向量(ST，SC)。

ST(Strategy Type)即均衡策略类型。一般可分为静态均衡策略及动态均衡策略。

静态均衡策略主要依靠现有状态进行，事先把任务相对合理地安排给每个节点。每个节点的任务数都是固定的，运行时进程不再重新分配。该方法的优点是可以相对简单地实现最优负载均衡，但在现实运行环境中较难实现。

动态均衡策略是指实时收集、处理并分析系统运行时的信息，在执行过程中动态地在负载比较大的节点和负载比较轻的节点间实现一种相对均衡的状态。动态均衡一般可以比较及时地体现系统任务的分布情形，当相对实现比较复杂，需要占用系统更多的资源，但能够产生更好的收益。

动态均衡策略的包括以下四个部分：

1、收集系统运行时的即时状态；

2、确定进程迁移调度的源点；

3、确定转移哪些进程。当然转移代价越低的进程应该优先被调度。

4、确定进程迁移调度的终点；

Linux 内核的动态负载均衡算法是基于“调度域”的。当某调度域的某个调度组的任务数量远低于该调度域的另一个调度组的任务数量时，才进行进程的迁移。Linux 内核启动均衡算法的时机有：

1)当某 CPU 即将进入 ilde 状态时，idle_balance()函数从其他 CPU 的运行队列中选择部分任务转移至本地 CPU 执行。

2)时钟中断周期性地检查每个调度域的负载是否均衡，如果不均衡就调用

load_balance()函数进行均衡。

Load_balance()函数依次在该调度域中寻找最繁忙的调度组以及该组中最繁忙的 CPU；如果能够找到最繁忙的 CPU，则遍历其任务队列，对所有的候选任务进行选择：

1)该进程并没有在 CPU 上运行；

2)该进程的 task_struct 结构的 cpu_allowed 字段允许该进程迁移；

3)该进程不是“高速缓存命中”的。

SC(Strategy Constraints)即均衡策略约束。在 Linux 内核中，用户可以设定自己的任务在哪个 CPU 上执行，这就是所谓的 CPU 亲和力(affinity)的概念。用户可以通过对进程控制块中的 cpu_allowed 字段进行设置实现。

4.3 实现均衡调度的重点问题分析

4.3.1 指标的选择

梁根等[31]设计的平衡算法中实现了一种公平指标，能够提高整体的吞吐量。Kunz 等[32]综合衡量了多个指标，包括各种资源利用率、系统调用发生的频率、队列长度等等，发现其中效果最好的单项指标是运行队列的长度，不过其实验存在一定弊端。Banawan 、Zahorjan 等[33]采取相关技术综合考察了一些指标，发现即时任务数、资源使用效率、平均任务数、平均响应时间等是相对比较理想的指标。国际上现有的负载均衡系统多采用运行队列的任务数作为单一负载指标[24]，但是这种负载指标存在不区分任务特征的缺点。比如，一个任务对 CPU 或内存的使用程度有可能比其他多个任务的使用程度还要高，交互式任务和计算型任务对 CPU 时间的消耗也有很大差异：生命周期很长的交互式任务可能只需要很少的 CPU 资源，但计算型任务则有可能在极短的时间内占据系统内所有的计算资源。同时，如果将 Linux 内核的系统任务也计算到 CPU 的任务数中去，该指标的性能就会更低，这是由于系统任务本身只需要极少的资源。若换个角度考量，我们直接使用 CPU 利用率、内存占用率和 I/O 利用率作为负载指标应该能够更准确的反映系统在 CPU 占用率高、内存占用率高以及输入输出设备占用率高等情况下的负载情况。基于此，本文综合考虑进程使用 CPU、内存及 I/O 的情况，采取综

合指标改进 Linux 内核的负载均衡策略。

4.3.2 进程迁移的开销分析

Leland 和 Ott[34]在 VAX750 和 780 上通过对 9.5×10^6 个 Unix 任务进行分析，提出任务的生命周期具有 UBNE 的特点，即 used better than new in expectation。也就是说，任务到目前为止已经累计运行的时间越长，那么到任务完成所需要的时间一般也越长。

经过大量的调查研究，Harchol Balter 和 Downey[35]指出：应该首先选择比较“旧的”任务进行迁移，由于这样的任务相对其他任务而言拥有相对较长的生命周期，能够弥补迁移的代价。

因此，进程均衡策略应该选择那些未来运行时间较长的进程来迁移。

Leland 和 Ott 等 [34] 还指出了如何计算进程迁移的代价：

$$c = f + \mu / b \tag{4-5}$$

f 是一个常数, μ 为任务占用的存储空间数量，b 是传输速率。

若任务 p 还需要 T(p)的时间才能完成，那么由上式可得：当 T(p)>c，即 T(p)>f+μ/b 时，进程 p 可以被迁移。

4.3.3 任务的生命周期（lifetime）

前人已对此问题进行了许多比较深入的研究，主要是采用统计分析的方法总结进程生命周期的规律。其中，Harchol Balter[35]运用数理统计技术，总结出了当进程应经运行的累计时间 age 长于 1 秒时其生命周期大于 age 的概率符合以下分布：

$$P\{Lifetime > age\} = age^{k} \tag{4-6}$$

其中,k 的值由于系统运行环境的差异通常在区间[-1.3，-0.8]内，一般偏向-1。基于该分布规律，任务的未来运行时间可以间接地通过其当前已经运行的时间来衡量，因为其未来的运行时间 T 大于等于进程已占用 CPU 的时间 age 的可能性很大。那么我们可以使用当前进程占用 CPU 时间 age 来代替进程未来的运行时间 T。因此进程的迁移条件就转化为：

当 age>c，即 age>f+μ/b 时，进程可以被迁移。

4.3.4 迁移进程的系统开销分析

多核结构中，进程的迁移含义是将某个进程在其存续期间有系统内的一个 CPU 搬运到其他 CPU 的操作。多核系统中的迁移与以前的迁移方式相比有其自身的特点。多核环境中的进程迁移在操作系统的内核中实现可以充分利用系统的各种功能，获取相关对象的各种状态，实现效率较高。但在某些多核体系结构中，L1 Cache 不是共用的而是是集成于核内，即每个核心都有各自独立的 Cache。在这样的体系结构中，进程从一个核心向另一个核心的迁移会带来一定的开销。

在 Linux2.6 内核中已有函数 can_migrate_task()来判断我们选择的进程是否合适被迁移，我们还是继续沿用该函数。也就是我们选择的迁移进程一般应符合以下要求：

1)该进程并没有在 CPU 上运行；

2)该进程的 task_struct 结构的 cpu_allowed 字段允许该进程迁移；

3)该进程不是“高速缓存命中”的。

同时我们在在下面的讨论中不再显式说明，最后选择的被迁移进程默认都已经满足上述三个条件。

进程迁移的主要任务是收集进程运行时的状态，以便迁移后根据这些状态再现该进程的执行环境，继续存取它所需要的资源并运行下去。进程的状态一般有：

1)进程执行时的处理器状态：比如在发生进程调度时的处理器现场，包括 CPU 各种寄存器状态等。

2)内存状态：地址空间信息、虚拟内存信息以及堆栈等。

3)控制关联信息：包括进程本身 PID、优先级、父进程 PID 等。

4)与文件操作相关的信息：如文件描述符，文件缓冲块等。

5)与消息相关的信息：如进程缓冲连接的控制状态等。

在多核体系结构中，当某任务被暂停运行时，对应 CPU 的上下文环境应该由该任务的 task_struct 结构保存。上下文环境主要有 CPU 的寄存器状态：通用寄存器，浮点寄存器，程序计数器（PC），栈指针寄存器（SP），处理器控制寄存器，内存管理寄存器等。当内核要迁移该进程到其他核心去时，内核用进程描述

符合适的字段来重新装载保存的CPU寄存器就状态即可在指定的CPU核心上恢复该进程的执行。

在Linux内核中，由于我们选择的被迁移的进程都处于非执行状态的也没有被“高速缓存命中”，因此实现进程迁移操作相对较简单：

```
dequeue_task(p,src_array);
dec_nr_running(p,src_rq);
set_task_cpu(p,this_cpu);
inc_nr_running(p,this_rq);
enqueue_task(p,this_array);
```

第一步从源CPU队列array中将选中的被迁移任务p移除，同时更新该队列的活动任务数以及该CPU的任务数，接下来将任务p的thread_info->cpu变量的值修改为目的CPU，同时将目的CPU的任务数增加1，最后将任务p安排进目次CPU相应的优先级队列中去。

由于任务的迁移需要两个运行队列的参与，因此将我们选择的待迁移任务的task_struct结构从一个队列搬移至另一个队列所需要的系统开销是一个常数，我们用f表示。

由于存储系统在大多数的多核环境下都是共享的，所以迁移进程时没有必要将其运行时的上下文都传送到目的CPU核心。同时，按照我们的策略应该被迁移的进程并没有被高速缓存“命中”，所以也不存在在CPU的高速缓存间复制进程的结构和数据的情况。但为了获取进程信息以明确其是否适合被迁移，我们还是要访问其存储空间，耗费的系统开销为进程占用空间的大小μ与存取速率 b 的商。

经过上述分析可获得迁移非执行状态的、不是“高速缓存命中”的进程的总开销是c=f+μ/b。这与前文分析的进程迁移代价的计算公式一致。

4.4 算法设计

本文综合考虑进程使用CPU、内存及I/O的情况，使用综合指标来衡量系统负载信息。定义CPU的负载向量为（%CPU，%MEM，%IO），其各分量分别代表了CPU利用率、内存利用率以及I/O利用率。

在 Linux 内核中，有一个全局变量 Jiffies。Jiffies 代表时间，其单位随硬件平台的不同而不同。系统里定义了一个常数 HZ，代表每秒钟最小时间间隔的数目。这样 Jiffies 的单位就是时钟滴答 tick，即 1/HZ。Intel 平台 Jiffies 的单位是 1/100 秒，是系统能分辨的最小时间间隔。CPU 利用率就是执行用户态和系统态的 Jiffies 数 J_{run} 除以总的 Jifffies 数 J_{sum}。设 M 为可以使用的内存总量，m(i)为某个进程 i 占用内存的数量，内存占用率即为此二者比值。IO 利用率为系统 IO 等待 Jiffies 数 J_{IO} 除以总的 Jiffies 数 J_{sum}。我们有

$$\%CPU = \frac{J_{run}}{J_{sum}} \times 100\% = \frac{\sum_{i=1}^{n} J_{run}^{i}}{J_{sum}} \times 100\% \quad (4\text{-}7)$$

$$\%MEM = \frac{\sum_{i=1}^{n} m(i)}{M} \times 100\% \quad (4\text{-}8)$$

$$\%IO = \frac{J_{IO}}{J_{sum}} \times 100\% \quad (4\text{-}9)$$

为了描述迁移进程时对源 CPU 的影响，定义进程 i 相对于源 CPU 的负载向量 R(s,i)为：

$$R(s,i) = (\%CPU_i^s, \%MEM_i^s, \%IO_i^s) \quad (4\text{-}10)$$

为了描述迁移进程时对目的 CPU 的影响，定义进程 i 相对应目的 CPU 的负载向量 R(d,i)为：

$$R(d,i) = (\%CPU_i^d, \%MEM_i^d, \%IO_i^d) \quad (4\text{-}11)$$

迁移进程对源CPU和目的CPU资源使用情况的影响可以通过对负载向量进行向量加减运算获得。我们的负载均衡算法的目标之一是找到体现各种资源利用率的负载向量之间的某种关系使系统负载均衡，即找到系统均衡时每个 CPU 负载向量角度的大小。

4.4.1 转移策略

本文的转移策略基于阈值策略：当某 CPU 核心的负载小于阈值 β_1 时，就将

该核心作为负载均衡的接受者；当某 CPU 核心的负载大于阈值β_2时，就将该核心作为负载均衡的发送者。

4.4.2 选择策略

也就是确定适合迁移的进程的集合P_m。根据本文前面讨论，我们可以采用进程的累计运行时间这个比较容易获得的数据来代替其未来所需要的执行时间，由此得出进程选择的条件：

$$P_m = \{p_i \mid p_i \in P_S \text{且} \mathrm{age}(p_i) > f + mem(p_i)/b\} \quad (4\text{-}12)$$

4.4.3 源 CPU 和目的 CPU 的定位策略

本文所讨论的负载均衡策略只考虑两类 CPU：一类是过载的 CPU；一类是空载或轻载的 CPU。过载 CPU 是多余负载的提供者 D (Donor)，轻载 CPU 是多余负载的接受者 R(Receiver)， R 中的 CPU 有能力接受从其他 CPU 迁移过来的负载。令 W 表示 CPU 负载，C 表示 CPU 能接受的负载；W_i表示某CPU_i的负载，C_i表示其能够接受的负载；W_D表示某个调度域中的所有 CPU 的负载，C_D表示该调度域能够接受的负载；W_G表示某个调度组中的所有 CPU 的负载，C_G表示该调度组能够接受的负载。其中，W_D和W_G是W_i的集合；C_D和C_G是C_i的集合。

算法流程参考图 4-1 所示：

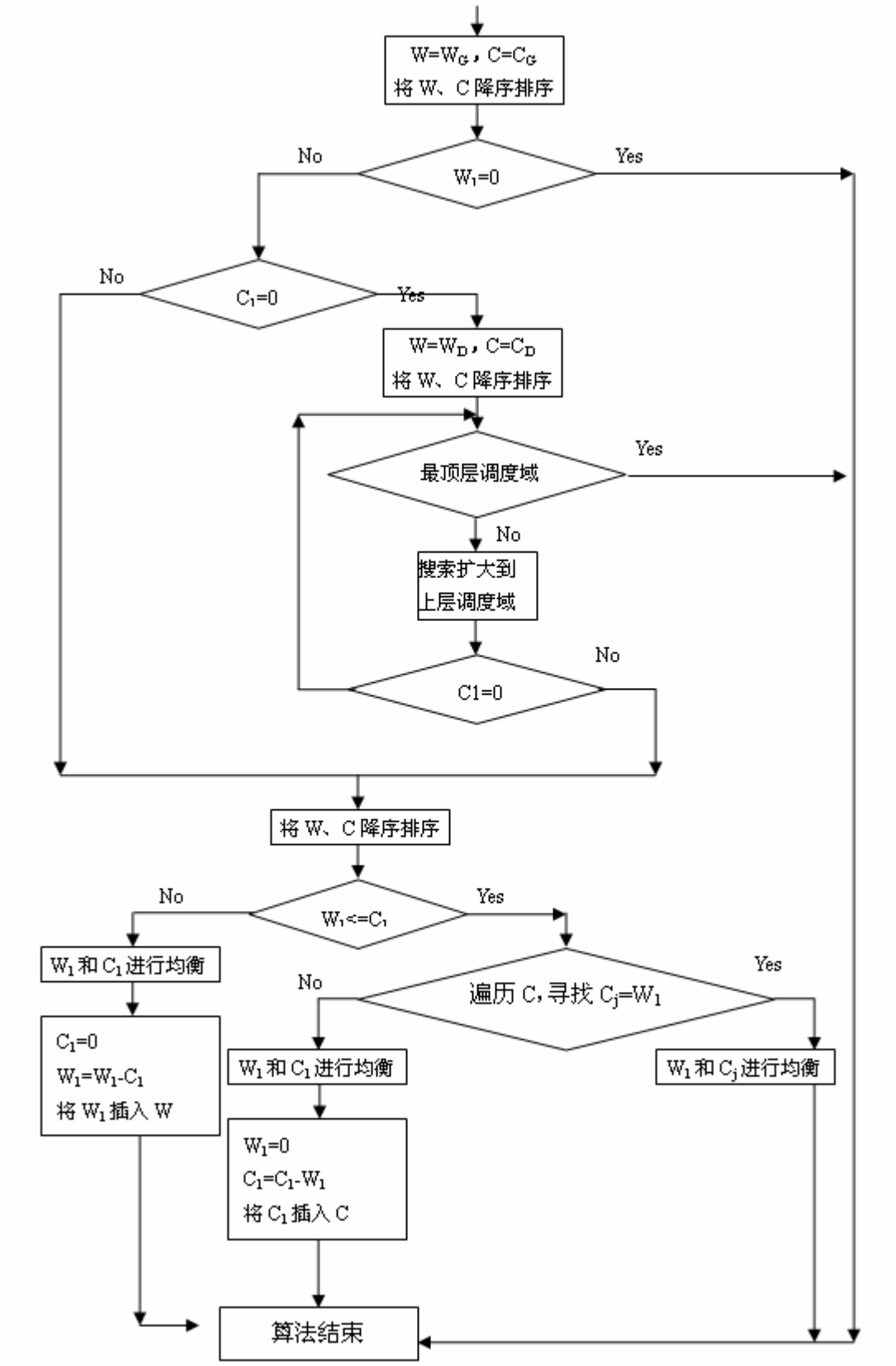

图 4-1 源 CPU 和目的 CPU 的定位算法流程图

4.5 实验测试及结果分析

实验基于以下硬件平台：Intel 4 核心 CPU Quad Q9500 2.83GHz，2G 内存。本文选择基于 Linux 2.6.21 内核的 Fedora 7 作为操作系统，按照上述算法思想对内核源代码进行相应的修改，然后对修改后的内核代码进行内核编译和调试，在修改后内核和修改前内核上分别运行测试程序进行对比分析。测试程序采用模块化设计，其模块说明见附录。实验时我们使主程序通过 fork()系统调用产生 10 个同样的子进程提交系统执行。

测试结果如表 4-1 所示：

表 4-1 负载均衡对比分析

算法	平均响应时间		平均 CPU 利用率		平均队列长度	
	测量值(s)	均方差	测量值	均方差	测量值	均方差
原调度系统	16.72	42.66	67.27	7.86	2.71	0.28
新调度系统	14.32	41.57	69.87	5.97	2.61	0.46

由上表可以看出，基于综合指标的新调度系统能获得更好的性能，但在平均队列均方差方面，则不如基于运行队列的原调度系统，这也是意料之中的事情，同时这说明同时考虑 CPU 运行队列及进程使用各种资源的情况应该能够获得最好的性能，这也为我们下一步的工作指明了方向。

4.6 本章小结

本章首先在阅读理解关于进程调度的国内外研究文献的基础上，建立分析 Linux 内核调度系统的模型。利用该模型针对负载均衡过程中的各种问题进行详细地分析，提出一种基于综合指标的基于 Linux 内核的新的负载均衡系统。实验说明，新负载均衡系统在一定程度上可以提高 Linux 内核在负载均衡方面的性能。

第五章 总结与展望

多核技术的相关研究有效地提高了系统的性能，已经成为应用的主流平台。同时，多核技术的蓬勃发展必然对计算机软件的研究与发展产生深远的影响。如何适应硬件平台的发展并提供有效地支持以发挥其性能是软件研发必须解决的问题。目前，以 Unix 系列、Linux、Windows 等为代表的操作系统在多核调度方面正处于一个快速发展的阶段。多核调度始终是操作系统研究的热点，其很多方面还处于探索阶段。这对我国软件的研发而言既是难得的机遇，也是巨大的挑战。我们应该充分利用这个机会，培养我们在软件研发特别是系统软件研发方面的实力，提高我们在核心领域的研究水平。

本文首先详尽分析了操作系统关于进程调度方面的理论，借助 Linux 内核的开源特性，对 Linux 的进程调度做了细致的调查研究。通过阅读理解分析大量的国内外文献，重点研究了 Linux 内核在多核调度负载均衡问题，设计了一种基于资源利用率的负载均衡系统，具有理论研究的价值以及实际意义。当然在该领域还有许多有价值的问题和研究方向等待我们去探索。针对本文研究内容，将来还需要在以下方面继续努力：

1、多种负载均衡策略并不是互相对立的，它们应该是互相补充的。接下来的任务之一就是把多种负载均衡策略集成，并设计实现相关的系统调用，以供用户针对不同的应用做出选择。

2、应该寻找途径提高现有算法的效率。在进程调度的相关算法中，查找是最基本的操作。在实现相关算法时，应该结合数据结构的研究，使用查找效率比较高的结构。

参考文献

[1] 陶树平等. 计算机科学技术导论（专业版 第 2 版）. 北京：高等教育出版社，2007.

[2] Moore’s law. http://en.wikipedia.org/wiki/Moore’s_law

[3] 计算机元器件技术展望. http://www.scuec.edu.cn/hpteach/cos/teach_ppt/lesson_h6_3.ppt

[4] 关于 ILP, TLP, SMP, CMP, CMT 的讨论. http://developers.sun.com.cn/blog/paul/entry/200701122

[5] 孟庆昌，牛欣源等. Linux 教程(第 2 版). 北京：电子工业出版社，2007.

[6] The Linux Kernel Archives. http://www.kernel.org/

[7] Linux 内核剖析. http://www.ibm.com/developerworks/cn/linux/l-linux-kernel/

[8] 钟小玲，袁宏春等. Linux 的进程调度. 计算机应用，2002，22（1）：42~44

[9] 高珍，吴齐明，周卫华等. Linux 操作系统内核对 SMP 的支持. 计算机应用研究，2002，62～64

[10] 安智平，张德运，党红梅，曜亚晖等. 基于 Linux 的进程主/从调度实现，计算机工程，2003

[11] 黄斌等. 多级反馈队列调度策略在 Linux 中的应用和实现，计算机工程，2004，30（20）：81~83

[12] 张尧学，史美林等. 计算机操作系统教程（第 2 版）. 北京：清华大学出版社，2000. 41

[13] 陈莉君，康华，张波等. Linux 内核设计与实现（第 2 版）. 北京：机械工业出版社，2006

[14] Linux 内核之旅. http://www.kerneltravel.net/kernel-book/第四章 进程描述/4.3.htm

[15] 孙忠秀等. 操作系统教程（第三版）. 北京：高等教育出版社，2003. 97～132

[16] Linux2.4 与 Linux2.6 内核调度器的比较研究. http://www.chinaaet.com/jishu/jslw/2008-05-27 /8789.shtml

[17] 叶经纬. 基于多核温度感知的 Linux 进程调度器研究与实现：[硕士学位论文]. 上海：上海交通大学，2010.1：19

[18]邹治锋. 基于 Linux 进程调度的改进与实现：[硕士学位论文]. 无锡：江南大学，2006.3：22

[19] Linux 进程管理之 CFS 调度器分析. http://blogold.chinaunix.net/u1/51562/showart_1840437.html

[20] 许占文，李歆等. Linux2.6 内核的实时调度的研究与改进，沈阳工业大学学报，2006，28（4）：438~441

[21] 曾占强，张钟澍等. 基于 Linux 系统的 SEDF 调度算法的研究与分析，成都信息工程学院学报，2007,22（6）：696~701

[22] 吴凌俊，李之棠等. 基于 Linux 的用户公平分享调度，华中科技大学学报，2003,31：247~249

[23] 完全公平调度. http://www.chinaunix.net/jh/4/1063798.html

[24] 鞠九滨，杨鲲，徐高潮等. 使用资源利用率作为负载平衡系统的负载指标，软件学报，1996，7(4)：16~20

[25] S.Zhou，X.Z.Delisle 等. Utopia:a Load Sharing Facility for Large,Heterogeneous Distributed Computer Systems，Software Practice and Experience，1993：1305~1336

[26] Y.T.Wang，J.T.Morris 等. Load sharing in distributed systems，IEEE Trans.Computers，1985，vol.C-34：204~217.

[27] D.L.Eager，E.D.Lazowska，J.Zahorjan 等. Adaptive load sharing in homogeneous distributed systems，IEEE Trans.Software Eng，1986，vol.SE-12：662~675

[28] 李冬梅，施海虎等. 负载平衡调度问题的一般模型研究，计算机工程与应用，2007，43(8)：121~125

[29] Marc H，Willebeek-L M 等. Strategies for dynamic load balancing on highly parallel computers，IEEE Transactions on Parallel and Distrubuted System，1993，4(9)：979~993

[30] 陈华平，计永昶，陈国良. 分布式动态负载平衡调度的一个通用模型，软件学报，1998，9（1）：25~28

[31] 梁根，郭小雪，秦勇等. 基于公平调度算法的分布式系统负载均衡研究，计算机工程与设计，2008，29(6)：1362~1363

[32] Kunz T.等. The influence of different workload description on a heuristic load balancing scheme，IEEE Transaction on Software Engineering，1991，17（7）

[33]Banawan S A，Zahorjan J 等. On comparing load indices using oracle simulation. 1990 Winter Simulation Conference.

[34] Leland W，Ott T 等. Load balancing huristics and process behavior，ACM SIGMETRICS

Conference on Measurement and Modeling of Computer System，1986.5

[35] M Harchol-Balter，A B Downey 等. Exploiting process lifetime distributions for dynamic load balancing，ACM Transactions on Computer Systems，1997，15(3)：253~285

附录：测试程序设计说明

测试程序整体结构整体结构如下：

```
global{
  test_name=”example”;
  duration=60s;
  run=interactive[1],background[3];
}
tasks{
  interactive{
    //具体任务定义
  }
  background{
    //具体任务定义
  }
}
```

进程结构简要说明如下：

```
worker_task{
  for iter=0 to inf do{
    work(working_time);
    sleep(sleeping_time);
  }
}
```

交互式进程结构设计如下：

```
interactive_task{
  if (interactive_signals_defined()) {
    set_signal_handler(signal_handler_function);
    create_periodic_timer(period);
```

```
  }
  if (interactive_sleep_defined())
    start_time = get_time();
  for iter = 0 to inf do {
    if (interactive_signals_defined())
      pause();
    if (interactive_sleep_defined()) {
      signal_handler_function();
      sleep_until(start_time + iter * period);
    }
  }
 }
  signal_handler_function() {
    save_start_time;
    simulate_event_processing;
    save_end_time;
 }
```

普通进程结构设计如下：

```
 universal_task{
  initializations();
  for iter=0 to inf do{
    iteration_specific();
    work(working_time);
    sleep(sleeping_time);
  }
 }
 initializations(){
  set_task_priority();
  if (interactive_signals_check()) {
```

```
        set_signal_handler(handler_function);
        create_periodic_timer(period);
      }
      if (interactive_sleep_check()) {
        start_time = get_time();
      }
    sync_start_for_all_tasks();
    }
    iteration_specific() {
      //具体进程处理
}
```

致　谢

在本论文完成之际，首先衷心感谢我的导师魏振钢教授。在研究生学习期间，感谢他对我在学业上的悉心指导以及在生活上无微不至的关怀，感谢他在研究方向上给予我的巨大的支持与鼓励，感谢他为我们提供了一个良好的学习、工作环境。在从开题报告到课题研究直至论文撰写的整个过程中，始终得到魏老师的精心指导和热情帮助，本论文的完成以及本论文取得的一些成果都凝聚了魏老师的大量心血。魏老师渊博的学识和敏锐的技术洞察力使我钦佩不已，他严谨的治学态度、不断进取的敬业精神使我终身受益。在此谨向敬爱的导师致以最诚挚的敬意！

在这里我还要感谢在学习和生活上给予我支持和帮助的师兄弟们，他们的支持和帮助使我克服了许多学习上的难关，扩展了我的知识面，与他们融洽的学习和交流使我受益匪浅。

我还要特别感谢中国海洋大学计算机科学系可亲可敬的老师们，没有你们的谆谆教诲及无私的奉献，也就没有我们的今天！你们的付出将永远被铭记！

最后，衷心感谢百忙中抽出宝贵时间评审本论文的各位专家、学者，并致以崇高的敬意！

个人简历

1983 年 4 月 21 日出生于山东省淄博市（县）。

2001 年 9 月考入中国海洋大学数学系信息与计算科学专业，2005 年 7 月本科毕业并获得理学学士学位。

2008 年 9 月考入中国海洋大学信息科学与工程学院计算机软件与理论专业攻读硕士学位至今。

发表的学术论文

[1] 基于距离估计优化初始聚类中心的 K-Means 算法. IFITA 2011，已录用，2011 年 7 月刊出

[2] 基于 J2ME 的手机蓝牙游戏开发与实现. 科技信息，2010（35）：74~76