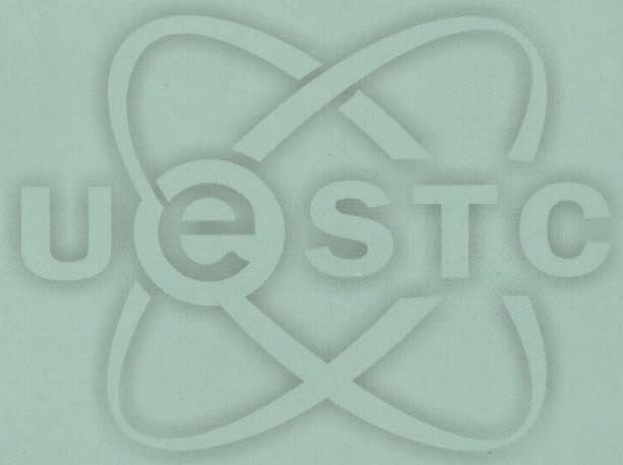
电子科技大学
UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER DISSERTATION



论文题目 基于Linux多核进程调度的研究

学科专业 计算机软件与理论
指导教师 顾小丰 高工
作者姓名 张修琪
学 号 200921060242

分类号 _____ 密级 _____

UDC[注1] _____

学 位 论 文

基于 Linux 多核进程调度的研究

（题名和副题名）

张修琪

（作者姓名）

指导教师姓名 顾小丰 高 工

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 2012.03 论文答辩日期 2012.05

学位授予单位和日期 电子科技大学

答辩委员会主席 _____

评阅人 _____

2012 年 月 日

注 1：注明《国际十进分类法 UDC》的类号。

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：张修祺　　日期：　　年　月　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：张修祺　　导师签名：

日期：　　年　月　日

摘 要

在计算机技术飞快成长的今天，处理器多核技术也得到了丰富和发展，并在我们的身边影响着我们的日常生活。现在，计算机硬件的指标的日渐提升，计算机系统的复杂程度也水涨船高，所以操作系统必须努力做出及时的、有效的改善，以达到硬件资源利用率的最大化的目的，而调度系统作为操作系统中最为重要的子系统之一，它性能的表现面临着严峻的考验，选择什么样的算法，如何进行调度、在出现负载不平衡时，SMP 调度系统如何做出调整，以及进行怎样的调整，才能使得软件与硬件相得益彰，这些都值得我们去深思，去探索。

本文首先对课题的背景进行了研究，概述了进程并行操作的两项技术：同时多线程 SMT（Simultaneous Multi-Threading）和片上多处理器 CMP（Chip Multi-Processor），并对进程、线程和它们状态之间的转换进行了介绍，接着对常见的调度策略和调度算法的基本知识进行了学习，并研究了一些 Linux 内核中的普遍用到的数据结构和宏操作，方便进行下一步的研究工作。

其次，以 Linux 2.6.36 内核为研究对象，细致的分析了它目前所使用的调度机制，完全公平调度系统 CFS 和与多核相关的 SMP 调度系统，为了对调度系统有一个整体的认识，本文对内核中关于 CFS 的数据结构和重要函数进行了剖析，并对 CFS 的工作原理进行了研究，为下一步地研究做准备。

再次，通过两条主线：Linux 检查系统中负载情况的时机，和针对出现负载不平衡时做出调整的策略，本文对 Linux kernel 中 SMP 的实现进行了解析，并概括了 SMP 的工作流程，并根据 SMP 的分析对已有调度系统的模型进行了改进。

最后，利用调度模型和对调度原理的分析，本文提出了一个针对 SMP 调度系统的基于唤醒信号的优化方案，以便于进一步的提升高速缓存的利用率。在原有的内核基础上，借助 FUTEX 锁机制，对优化方案进行了设计，对并 Linux 的相关源码进行了修改，对优化方案进行了实现，然后对使用了优化方案的内核系统和标准的内核系统分别进行了对比测试，并对结果进行了验证和分析。

关键词：多核，完全公平调度，对称多处理机，Linux 进程调度

Abstract

Nowadays, with the rapid development of computer technology, the processor multi-core technology has been enriched and developed, which affects our daily lives.Now, with the rising of the computer hardware performance, the complexity of the computer system have gone up, so the operating system must strive to make timely and effective improvement in order to achieve the purpose of maximizing hardware resource utilization, and scheduling systems as one of the most important subsystem in the operating system, its performance is facing severe test. And select what kind of algorithm, how the SMP scheduling system load imbalance at a pinch, how to make adjustments, as well as what kind of adjustments in order to makesoftware and hardware benefit each other, all of them are worth for us to ponder and explore.

Firstly the background of this subject is introduced in the dissertation, and an overview of the process of parallel operation of two technologies: multi-threading technology (Simultaneous Multi-Threading, SMT) and chip multi-processors (Chip Multi-Processor, CMP), and introduces the processe, the thread and the conversion of their states, then the basics of common scheduling policy and scheduling algorithm has been learned, besides some of the Linux kernel commonly used data structures and macro operation, which offer a plenty preparation for the below research.

Secondly, regarding the Linux 2.6.36 kernel as researching target, its current scheduling system has been analyzed, completely fair scheduling system and SMP scheduling system. In order to have an overall understanding of the scheduling system, the scheduling system from the data structures and important functions of CFS in Linux kernel have been studied, and the working principle of CFS have been studied.

Again, through two key approaches, namely when to check the system load and how to adjust for the load when imbalance occurs, the implementation of SMP in the Linux kernel has been parsed, and workflows of SMP have been summarized. Andaccording to the SMP analysis, an existing scheduling system model has been improved.

Finally, using the scheduling model and scheduling principles of analysis,a

wake-up-signal based optimization scheme for SMP scheduling system has been presented, in order to further enhance the cache hit rate. With FUTEX lock mechanism, the optimization has been designed, and Linux source code has been modified to implement the optimization scheme, then using the optimization kernel and the standard kernel system were carried out comparison tests, and the results were validated and analyzed.

Key words: Multiprocessor, Completely Fair Schedule, Symmetrical Multi-Processing, Linux Process Scheduler

目 录

第一章　绪 论

1.1 课题研究背景及意义

随着技术的发展，计算机科学也在不断前进，而 CPU 作为计算机最昂贵的资源，发展的速度很快，体积骤减和频率剧增，但是单核处理器的架构越来越繁琐、复杂，不仅给设计带来了相当大的困难，而且还使得 CPU 得不到最大化的使用，再加上体积变小，功耗却没有降下来，散热也成了问题，很难使得处理器的设计与实现以及频率的提升能够顺利的进行下去[1]。

一来处理器的频率提升出现了天花板效应，很难再提升。微处理器性能的提升很大程度是依赖于工业设计的发展，制造 CPU 所需的电阻电容等的体积在急剧的变小，这就使得在同一个处理器内集合很多的设计，然而它们之间的连接线路却成了它们的瓶颈，导致了频率不易于提升。所以一些大公司在研发了 3.8G 赫兹这样的高频率 CPU 后，就不在这个方向上继续投入开发力度。

二来处理器的功耗也不断飙升。CPU 的制造工艺一直在不停的改善，晶体管的体积革命性的变小，这就便于集成，再加上 CPU 的频率提升了，单位面积内功耗比以前更多了，发热量也更大了，这不利于 CPU 的稳定工作。

所以转向多核处理器是发展的需要。片上多核处理器 CMP（Chip Multi-Processor）[2]就是在那个关键的时刻被提出的，它在一个处理器芯片上封装了多个核心，各个核心能并行地运行各自的任务，非常接近传统的对称多处理器系统（Symmetric Multiprocessors，SMP）[3]。多核处理器中的每个核都可视为一个独立的单元，并且它的实现起来相对容易，对以后的研发也便于扩展，并且发热也没有比单核处理器少。多核处理器在频率不变的情况下通过并行运行多个任务来实现性能的提升，也使得 CPU 得到了充分的利用。

此外，在处理器其他资源上也有所突破。加利福尼亚大学提出了同时多线程处理器 SMT（Simultaneous Multi-Threading）[4]技术，它让多个任务共用处理器、Cache 和堆栈等资源，来提升资源的利用情况。

首先，数据共享的问题要充分的解决，虽然在单核系统中，也有进程、线程的并行执行，但是从其原理和实现上可以看出，它并不是实际意义上的并行操作，是通过时间片的轮训，让多个任务在一个时间段内同时运行，而多核系统中的并

行是真正的并行，系统中的多个处理器都是在同时执行的，即任意时刻有多个进程正在运行，这就要求对进程间共享数据的访问更加的谨慎，所以同步机制要重新考虑，改用更合理的锁机制。

其次，系统中的处理器增多了，该让哪些任务到哪个处理器上执行，才能使得系统的性能最大化。原先，系统中只有一个处理器，进程的分配不用考虑处理器的环节，只要根据其他的标准作出判断既可，但是现在又有一个非常重要的因素加入，原先的标准就必须作出调整以适应新的架构。并且在处理器间出现某些CPU 闲着、某些超载时，要进行适当的调整，让整个系统的 CPU 利用率最大化，而怎么去发现这些不平衡的现象，在出现时该如何调整时，才能使的执行的总时间最小，Cache 得到更多的利用。这将是本文探讨的重点，从调度上使系统的性能最优。

1.2 国内外发展现状

SMP 体系结构的发展极其的快速，它不仅被用于个人电脑，而且系统中处理器的数量也在不断增加，并且在图像和视频处理方面表现的极其出色。

2008 年美国休斯顿大学利用对子任务进行分层调度排队的算法，通过模拟实现，提出把运行队列进行分级可以解决 SMP 系统中的 CPU 归属问题[5]。

剑桥大学也提出了 SMP 当前运行队列的调度算法，并归结出进程进行切换的频率越高，SMP 系统所承担的压力就越大[6]。

华中电子科技大学也对 SMP 系统的调度策略进行了研究，提出根据进程的 CPU 亲和力进行任务的分配，可以到达很好的效果[7]。

梁根和郭小雪等通过对比队列中的任务数目、调用的快慢、进程进行切换的概率、CPU 的使用率、内存的使用率等指标及它们的搭配，归结出用运行队列的长度最合适[8]。

鞠九滨等提出 CPU、I/O 利用率比单独使用 CPU 运行队列的长队作为负载平衡系统的负载判断准则时，调度系统的性能要好，并且如果把它们组合起来一起考虑，效果更佳[9]。

电子科技大学的覃中通过对影响 SMP 调度性能因素的综合考虑和研究，归纳出了四元组<E，T，L，S>，来作为负载平衡的模型[10]。

北京邮电大学的胡丽聪等针对负载平衡问题，提出了基于动态反馈的一致性哈希负载均衡算法，实现了系统均衡、稳定、热变更的目的[11]。

哈尔滨理工大学的赵磊针对多核处理器提出了一种基于任务复制方式的分解DAG 任务图的方法，解决了核间通信开销大和根据处理器内核的个数进行调度的问题[12]。

华南理工大学的廖江苗对多核并行 B-link 树的索引进行了设计和研究，通过实验测试，证明了该算法在多核处理器系统中能带来性能上的提升[13]。

内蒙古大学的曹皓通过对调度系统进行动态模拟和分析，断定不管 CPU 核的数目是几个，都能进行负载均衡，并且不能盲目的追求核数的增加，随着调度粒度的增大，系统的响应时间也会增长[14]。

四川大学王正霞和刘晓洁等针对负载平衡的问题，设计出一种借助于 B+树的的反馈式平衡框架，并进行了模拟，发现该算法能够及时将负载进行平衡[15]。

河南大学张少辉通过分析并行程序设计中的并行问题，提出了一个利用 MPI 和 Windows 和 BP 神经网络，来对负载均衡进行预测，它能够及时有效的收集节点的负载情况，利用 BP 神经网络来推算节点未来某时的负载情况，为任务的分配和调度提供依据[16]。

上海交通大学的叶经纬从 CPU 的功耗着手,提出了四种基于温度的调度算法，利用进程的调度，使处理器的温度能够降下来，并让温度在核间分布的更加合理，实验表明，这样算法可以有效的使温度下降，功耗降低[17]。

虽然到现在，还没有特地为多核体系结构而研发的多核操作系统，也没有非常成熟的针对多核系统的调度机制，但 Linux 内核作为自由和开源的操作系统内核，并且非常的稳定、可靠，也容易进行移植和扩展，深受程序员的喜爱，并且它还能够很好的支持 SMP 系统，并且 Linux 内核还会根据处理器的硬件结构判断是不是能够实现 SMT 和 CMP，来决定如何更贴切的使用处理器资源。从 Linux kernel 2.6.23 开始 CFS（完全公平调度）[18]接替了 O（1）[19]调度，并且它的实现也不难，因此我们用 Linux 作为研究和改进的平台，对 x86 架构的调度系统进行研究。

1.3 论文主要工作

现在多核操作系统已经普遍的应用到了我们日常生活当中，并使得我们的生活更加丰富多彩，本论文试图通过剖析 Linux 内核的源码，掌握进程调度的工作原理及调度流程，并在此基础上作出改进。本文主要探讨了基于 Linux 内核的多处理器 SMP 调度系统的机制，主要的内容包含：

1. 研究了 SMP 和 SMT 的体系结构；
2. 介绍了一些典型的调度算法及相关的知识；
3. 剖析了 CFS 调度系统，并对 CFS 机制进行了研究；
4. 研究了 SMP 调度系统的机制，提出了一个调度模型<P，T，M，S>，并通过对其性能的分析，提出了针对 Linux SMP 调度的优化方案。

1.4 论文章节结构

本文的章节组织结构如下：

第一章，绪论。研究了课题的国内外背景，以及研究多核系统进程调度的意义，并通过研究国内外的发展情况，把握本文的研究方向。

第二章，调度系统理论基础。重点研究了 SMT 和 SMP 的体系结构，并介绍调度系统的一些基础知识和内核中一些重要的数据结构和宏操作。

第三章，CFS 调度系统剖析。对 CFS 调度的源码进行了解析，对 CFS 机制进行了研究。

第四章，SMP 调度系统研究。研究了 Linux 内核中的 SMP 调度系统，并概括出了它的调度流程，根据其原理概括出了一个调度模型<P，T，M，S>，以便于进一步的学习和研究，并通过对 SMP 系统性能的分析，并提出了一个优化方案，借助于 Linux 平台进行了实施、测试以及把对比的结果进行了比较和分析。

第五章，总结与展望。总结了一下课题，展望了将来可能的改进的方面。

第二章　调度系统理论基础

为了衡量处理器效率，目前提出了“每晶体管性能”的概念，即在芯片总功耗下每个晶体管的平均性能。从现在的发展趋势来看，多核处理器的结构设计是提高每晶体管性能的有效手段其中，普遍用到的技术有 SMT、CMP、CMT 和 SMP，下面来重点讲解一下 SMT 和 SMP 体系结构。

2.1 SMT

多线程处理器 SMT（simultaneous multithreading）的大体思想是：在每个周期中，负责指令发射的部件选择没有关联的指令，将其发射至各个对应的执行模块中，借此来提高功能部件的利用率，它的组织结构如图 2-1 所示。SMT 处理器对任务的调度方式与操作系统的方式有着明显的区别，操作系统中对任务的调度是由通过软件实现的，而 SMT 处理器对指令的调度是由硬件了来完成的。SMT 技术利用拷贝处理器上各结构上的信息，让处在相同处理器上的多个任务能够共同使用处理器资源，从而尽量地发射指令和超标量的处理等，使处理器资源尽量的被使用，并有助于降低出现访问数据时的 Cache 未命中的情况，它的流水线结构如表 2-1 所示。SMT 技术能够为快速运转的核心模块提供充足的数据进行处理，进而降低出现处理器闲置的情况。SMT 技术之所以备受青睐，是因为它对处理器中的核心设计修改很少，就能使性能提高很大一块。

SMT 结构的长处是：拥有多线程处理器，以及能够进行超标量处理，允许在

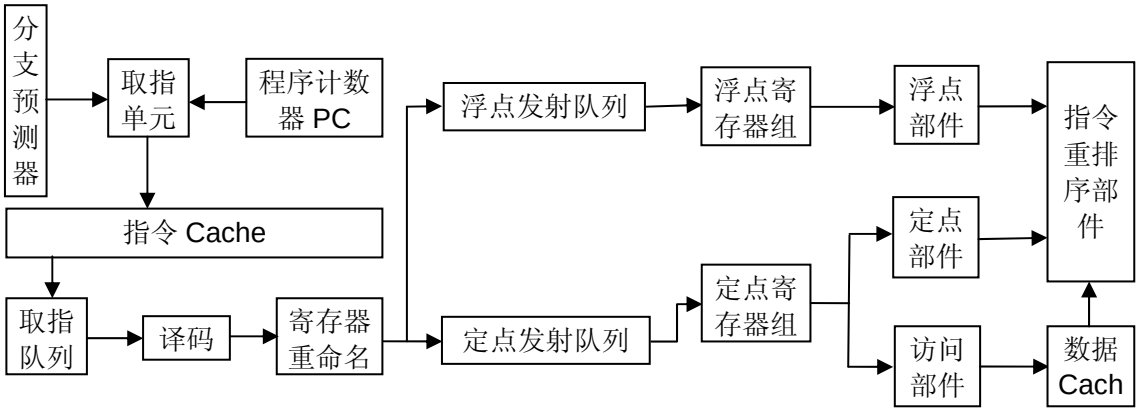


图 2-1 SMT 模块结构图

表 2-1 超标量流水线和 SMT 系统的流水线

（a）超标量处理的流水线

取指	译码	重命名	队列	寄存器读	执行	提交

（b）SMT 结构的流水线

取指	译码	重命名	队列	寄存器读	寄存器读	执行	写回	提交

单个周期内运行多条来自不同任务的指令，即在单个时钟周期内，SMT 可以通过软件的 TLP 和 ILP 两项技术来降低水平方向上的开销，增加对处理器资源的利用。不仅如此，SMT 还能够让来自不同任务的指令集合起来再进行发射，当然若只有一个任务是活动的，该任务就能够霸占所有可用的发射部件，这就最大限度的使用了发射模块，进而降低了垂直方向上的开销。

SMT 结构的短处是：一方面，随着指令发射宽度的增大，指令发射阶段变得比较复杂，电路上的延迟也就增大，进而限制了主频的提高；另一方面，多个任务共同使用一级 cache、分支预测以及 TLB 等，这就造成了竞争加剧、冲突不断，进而造成 Cache 缺失几率上升，分支预测也频频出错，影响了整体的性能。

在 SMT 体系框架中，当运行的任务数目比能够进行处理的数目大时，操作系统就会调度一部分去执行。因为在能够支持 SMT 的系统上，来自相同任务上的指令之间每时每刻都会争夺对硬件的使用，那么运行来自不任务上的指令应该就能够缓解这种压力的影响。一个调度系统就是负责从运行队列中，挑选出适宜的任务集合去同步运行。Snavely A 很早就对 SMT 系统中的调度进行了探索，他用“共生（Symbiotic）”这个名词来评价 SMT 系统上多个任务一并运行的可能性，并归结了针对于 SMT 系统中调度问题的算法：SOS(Sample Optimize Symbiosis)，它包括采样和共生两个阶段。采样时，从全部任务的各个子集合中采集运行状况信息，到了共生时，通过采集的信息来判断各个子集合的是不是那个最合适进行执行的子集合。实验表明，通过 SOS 算法，能够很快的找到最适宜进行运行的子集合，并能够使调度性能提升 17%。

但 SOS 算法忽略了进程的优先等级，为此，Sanvely 等又推出了一个新的算法，通过仿真，该算法能够减少 33%的系统响应时间。之后，Parekh 等又提出了基于任务敏感度的算法。它通过任务执行过程中的反馈来挑选进行运行的最合适的子集合，以求达到处理器的最大吞吐量。它还能够对性能和公平性进行兼顾，跟基于时间片的算法比较，利用 IPC 任务敏感度的算法在很小的硬件成本代价下，把加速比提升了 10%左右。

2.2 SMP

一个计算机系统上有多个 CPU，而各个 CPU 同步执行操作系统的一份拷贝，并且它们对总线、内存等资源是共用的，这种系统就是对称多处理机 SMP(Symmetrical Multi-Processing)。在 SMP 系统中，多个处理器共用总线、内存以及 I/O 等资源，并且它们也共同执行整个系统中的所有任务，利用这种方式来提升整体的运算处理性能，因此 SMP 系统也称作一致性内存访问 UMA（Uniform Memory Access）系统，而非一致性内存访问 NUMA（Nonuniform Memory Access）系统就没有此特性，系统中的 CPU 在与属于不同节点上的内存进行交互时，会有不同的开销和访问速度。尽管 SMP 系统中多个处理器，但用起来体会不到。而片上多核处理器 CMP(Chip multiprocessor)是另一种新的体系结构，它的基本思想是：利用简化超标量结构设计，把大规模的并行处理器中的 SMP（Symmetric multiprocessors）或 DSM（Distributed shared processor）节点集成到一个芯片上。SMP 系统大体的结构分布如图 2-2 所示。

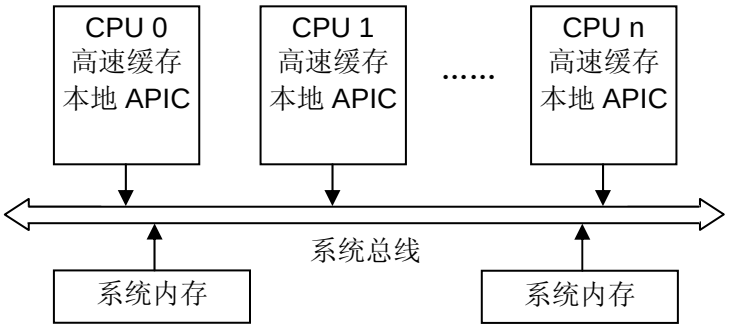


图 2-2 SMP 系统各模块分布图

SMP 处理机操作系统主要可以分为是主从式、独立监督式以及浮动监督式三大类系统。

主从式 OS 主要是由其中某个 CPU 记录并管理其他 CPU 的运行，当系统中有多个进程要执行时，主 CPU 要负责给它们调度。主从式 OS 的管理程序和主要的操作功能在主 CPU 上执行，来自从 CPU 的访问请求通过陷入递交给主 CPU，主 CPU 负责应答这些请求。主从式 OS 的不足之处是，一旦碰到极为严重的错误，整个系统可能会死机，需要重启才行。可见，主 CPU 的地位极为关键，一旦它出问题，整个系统就不能工作，即便是它的工作效率不高，也会连累整个系统，使整体的性能受损。

独立监督式 OS 跟有别于上边的那种方式，它的每个 CPU 分别都有属于自己

的控制中心。

浮动监督式 OS 在同一时刻有一个 CPU 是负责监管整个系统的主 CPU，但这个主 CPU 不是固定不变的，而是轮流来做主 CPU。可见这是一种很有效、很灵巧的处理方式，只是实现起来稍微繁琐一些。但它的优点足以让它应用在 SMP 系统的设计当中，也被用于紧耦合的多 CPU 架构中。

2.3 进程调度

进程作为现代计算机程序在运行时的抽象，是操作系统中的一个非常核心的概念，后来为了降低进程调度的系统开销，在前者的基础上又提出了线程的概念，它的出现，在很大程度上缩减了系统开销，提升了整体系统的效率，接下来就讲述一下线程和进程的概念以及相关知识。

2.3.1 基本概念

“进程”最先被 MIT 的 MULTICS 系统和 IBM 的 CTSS/360 系统所使用，是多道程序设计里的一个基本的概念，它常被看做是程序运行时的实例，用来充分描述程序执行到何种程度的数据结构的集合。而从操作内核的观点看，它被看做拥有系统资源（CPU 时间、内存、总线等）的最小实体。

进程有四个基本的组成部分：程序，专属的堆栈，进程的控制块 task_struct，独立、专有的内存区。这些都是必要条件，缺了一条便称不上是进程了，如若没有用户空间则就成了内核线程，如若有共享的用户空间则是用户线程。

内核态线程[19]和用户态线程的一个很重要的区别是没有自己的内存，它通过借用用户态的进程或线程的内存来用，它是在内核态生成的，不过对于我们使用来说它和进程区别不大，用起来也不必担心那些同步访问内存的操作，在切换时也不必非要写回内存操作。还可以使用它来实现传统的进程，如页面守护进程，网络守护进程等。在多核环境中，内核线程的使用真正的使并行操作成为现实。

但它也有自己的缺点，在生成、同步接下来的管理中，经常要使用系统调用，这无形之中给系统带来了负担。所以，线程间经常需要做切换和同步操作时，尽量避免使用内核线程。

用户线程则让线程可以在用户态生成，整个生命周期的操作基本都在用户态完成。它对内核来讲是透明了，系统的内核看不见它，所以不管系统是否不支持线程，用户都能够生成任意个线程。

用户线程是借助于线程库的来生成的，它利用抽象系统的内核来达到目的。线程执行需要的堆栈等都资源由线程库来监管，它的切换也不需要内核的参与，所以对系统的压力大大的降低了，这也使得操作起来比较快，这要是它的一大亮点。

在 Linux 内核源码中进程常被称为任务（task）或线程（thread），为了便于管理，内核使用进程描述符对进程自身和所做的事情进行描述，如描述进程的优先级、运行的状态、自身的内存空间、允许访问的哪些文件等，这便是我们常说的进程控制块。我们一般人认为，能够独立的被调度的各个任务都需要拥有自己的进程描述符。进程描述符在内核中用 task_struct 结构体来表示，其中包括了进程相关的全部内容，如进程的属性字段、指向其他结构体的指针变量等，进程描述符跟进程之间是一对一的关系，并且用标示符 PID 来标示进程。

其中，task_struct 中的 state 变量用来描述进程目前的状态，它集合了很多标志位，一种标志对应着一种状态，且它们是不可以在一个进程上同时存在，当进程已经准备好了，就差得到 CPU 进行执行时，那么它就进入了就绪态，即 TASK_RUNNING 状态；若它在执行过程中因为某个条件还没成立，它就进入了等待状态，如果这个状态可以被中断或是信号唤醒，那么它就是可中断的等待状态，否则就是不可中断的等待状态；如果进程被暂停运行，那么它就变成 TASK_STOPPED 状态；

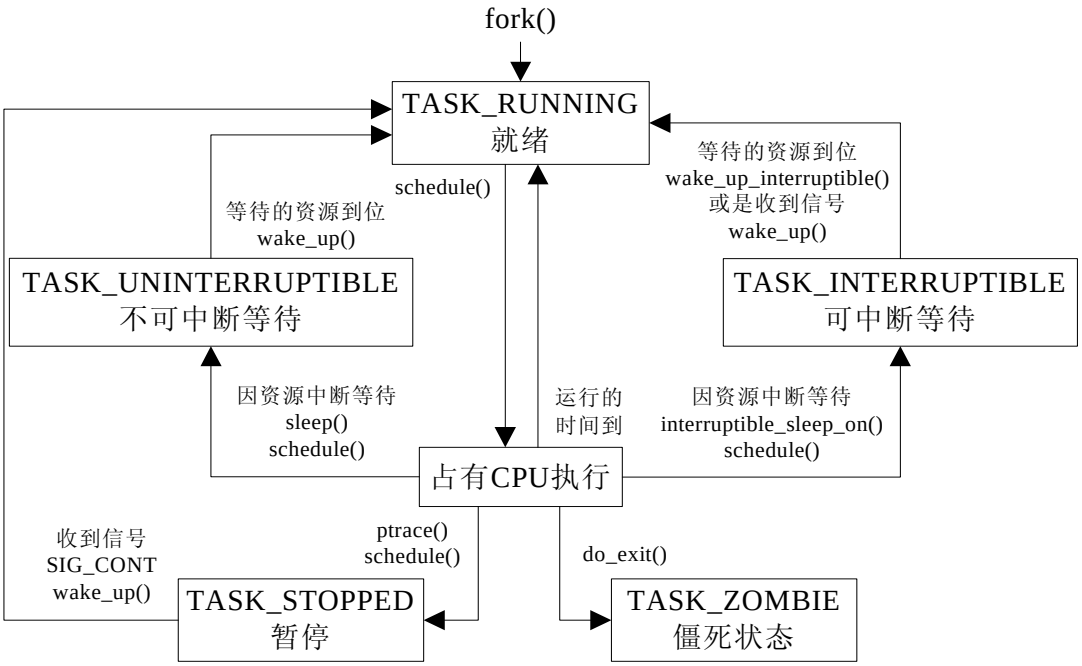


图 2-3 各个状态间来回转变的示意图

如果进程被调试或追踪程序监视，它停止运行，进入 TASK_TRACED 状态，很多信号这可以实现该操作；如果进程已经结束，但其父进程还没有通过系统调用收集它的信息，那么该进程就进入了 TASK_ZOMBIE 状态[21]。转换关系如图 2-3 所示。

2.3.2 调度策略

进程调度作为操作系统上的一个重要操作，它性能的好坏影响深远，所以设计一个性能卓越的调度系统非常的重要，它的性能是一个操作系统性能的重要衡量要素。要做好进程的调度，首先要了解进程的种类传统上，进程被分为两类：I/O 受限型（I/O-bound），即经常使用 I/O 设备，所以可能经常发生 I/O 中断，等待 I/O 设备执行完它的工作；还有一种是 CPU 受限型（CPU-bound），顾名思义，这样的进程需要很多的 CPU 资源去实现一些计算操作。

设计一个调度系统非常的繁琐，它不仅会影响系统实现的复杂性，还会由于难于实施而不得不在原来的方案中进行压缩，不论怎样，它应该能够满足下边的几点要求。

首先，交互式的任务处理；在现实使用中，这些任务需要经常与用户进行交互，为了不至于让用户觉得自己的请求被拖延，而是让各个用户都觉得系统在一心一意的处理它的请求，即使系统的负载很重时，也能让用户的请求的到尽快的处理，而不是觉察到有太明显的延迟。这类任务的代表有命令 shell，上网聊天软件，编辑类软件等。

其次，批处理的任务；批处理任务一般都作为后台任务来执行，不必与用户进行交互，对响应速度的要求不严格，要求的是平均速度，所以执行完一个任务所用的时间是一个经常被用到的标准。典型的批处理程序有汇编程序，科学计算，搜索引擎等。

最后，实时的任务；这类任务的执行对时间性要求最强，它不但要求任务运行的平均速度，而且还要求即时速度，不但要求有很高的交互性，还要求任务能够尽快的完成。在实际应用中，可预测性是对进程执行的重点关注对象。我们常见的实时任务有视频播放软件，机械、车床中的控制软件等。

此外，一个调度算法还要保障公平性的原则，即让全部的任务都能够得到执行，尽管会因为自身的紧急，所需的资源是否可用等因素的制约，它们的进度可能不同。并且要防止死锁，目前主要采用预防死锁的手段，通过破坏形成死锁的条件，来预防死锁，即破坏互斥、请求保持、环路等待、不可剥夺四个条件之中

的一个或多个。

为此，在设计一个调度算法时不应该忽视的方面有：

（1） 调度时机：在什么时候、什么情况下发生调度。

（2） 调度的策略：依据什么样的评判标准去选择并让其执行。

（3） 调度的方式：是“不可剥夺”的还是“可剥夺”的，当前正在运行的进程不释放所占用的 CPU 资源时，能不能迫使它对 CPU 资源进行释放，不让这个进程继续执行，而让其他的进程投入运行。如果可以迫使它释放资源停止运行，这在所有的时候都可以这么做吗，这么做是否安全等。

从经典的操作系统设计上看，一个成熟的调度算法需要达到几个看似彼此矛盾的目标：进程响应速度尽量快，后台批处理任务的吞吐量尽量大，尽量避免进程饥饿的情况，不同优先等级的进程间要尽量调和一些等。

Linux 的调度机制是利用了时间片的技术，多个进程轮换的相互交替执行，理论上讲，在任一段时间内可以有 N 个进程在同时执行，但在任意一个时刻只有一个进程在执行。倘若某个进程用完了自己的时间片，但尚未执行完毕，那么就需要进行切换。该技术依靠定时中断，并且对进程来说是看不见的，应用程序不必对这些进行考虑。

调度也根据进程的优先级进行分类，通过某种算法得出进程的当前的优先级，这个值决定了进程如何分配 CPU 时间。在 Linux 中，进程的优先级不是一成不变的，调度系统会监控进程的运行情况，并在一段时间或后根据它的情况进行优先级的更改，在一段时间内没有得到 CPU 的进程可以动态的提升它们的优先级，相反地，运行了一段时间的进程则动态的降低它们的优先级，以调和不同优先等级的进程都能得到执行。

Linux 2.6 及以后的版本都是支持抢占式的，这就意味着无论进程是内核态的还是用户态的，都很有可能被高优先级的进程所抢占。

普通进程都有各自的静态优先级，调度算法通过静态优先级来划分优先等级。Linux 内核使用从 100 至 139 四十等级来描述普通进程的优先级，其中 100 表示最高优先级，139 表示最低优先级。

新创建的进程一般继承它父进程的静态优先级，它本质决定了该进程的基本时间片的长短，静态优先级的值与优先等级成反比，级别高的时间片长。

普通的进程不但有静态优先级，而且还有动态优先级，它的取值介于 100 和 139 之间，调度程序依据它的大小来决定哪个进程投入运行，它的值也与优先级别

成反比，级别高的容易受 CPU 的青睐。

2.3.3 调度算法

在最初的操作系统中，调度算法比较简单：在每当要发生切换时，内核检查就运行队列链表，根据各个进程的优先级，选择最高的投入运行。但是，当就绪队列中的进程越多时，选择的过程要花费的时间也会随之增长。

Linux 2.6版本以后的调度算法复杂的多，现在可以比较好的解决了就绪队列中进程数量与算法时间复杂度的关系，并把时间复杂度控制在固定时间内，即便运行队列中的进程数量变多了，时间复杂度不会增加。同时也比较好的解决了也处理器数量的关系，每个CPU都维护着各自的运行队列。此外，现在的调度算法也能更合理地对待不同类型的进程，所以在负载很高的系统中，Linux 2.6中的响应时间明显比以前的快[22]。

包括Linux在内的多大系统中，进程一般按照一下几种调度算法进行调度：

（1） SCHED_FIFO：先进先出，多用于实时的进程，当进程被调度并得到CPU资源后，它的描述符被放在运行队列的当前位置，如果未出现更高优先级的实时进程要执行，则该进程持续占用CPU资源而不放弃，即便存在相同等级的可运行进程等待执行。

（2） SCHED_RR[23]：时间片轮循，多用于实时进程，每当一个进程的到CPU资源后，就让其进程描述符被插到运行队列的末尾，该算法可以让具有同等优先级的实时进程分配到比较合理的CPU时间。

（3） 最短进程：这是不可剥夺的策略，基本原则是所需CPU时间最短的进程优先得到调度。它的缺点是如果有持续不断地短进程，长进程会因为长时间得不到CPU而严重饥饿。并且没有剥夺机制的参与，所以在分时或事务处理时性能不高。

（4） 最短剩余时间：对上一个版本进行了改进，增添了剥夺机制。在此种调度算法下，距任务完成所需时间越短的进程越受调度程序的青睐，越容易获得分配到CPU。当就绪队列中新插入了一个进程时，其剩余时间可能比正在运行的进程还要少，只要该新进程一加入，调度系统就切换正在运行的进程并让新进程开始执行。跟上一个算法一样，调度程序计算进程的所需的CPU时间，并且容易造成长进程饿死的现象。

（5） 最高响应比：进程等待处理器的时间+进程进行服务时间，它们的和再

除以进程进行服务的时间就是该进程的响应比，它的基本原则是响应比越高的就绪进程越容易获得CPU的使用权，这个算法很有吸引力，因为它依据进程的年龄做出判断。开始时可能偏爱短作业（因为短作业的期待服务时间小，即分母小导致整个比值变大），而长进程因得不到服务致使等待处理器的时间增加，也使响应比值变大，最终也会使得自身的响应比很高，从而占用CPU资源。

（6） 反馈：它是利用基于剥夺的动态优先级机制实现的。某进程在首次转变到运行状态时，它被置于到最高优先级的队列，在执行结束下一次转变成就绪时，它又被置于第二高优先级的队列，总之，它的执行被剥夺一次，优先级就降一级并被置于相应的队列内，并且各个队列中一般都采用FIFO的策略，如果进程落入最末优先级的队列，它就循环往复地进出此队列，直至执行完毕。

2.4 Linux 内核链表

无论在教科书中还是项目中，我们对链表都很熟悉，但是在 Linux 中的链表与众不同，我们针对 2.6 内核中的链表[24]来讲解，链表在 Linux 内核中的实现如下边的代码所示：

```
struct list_head {      //定义了一个结构体。
    struct list_head *next, *prev;  //结构体的成员是该结构体类型的两个指针。
};
```

可以看出，此结构中定义了两个指向相同类型结构的指针：prev 指针和 next 指针，通过上一节的学习我们知道，内核中定义的链表是双向链接的，但是，我们注意到，该结构又与上一节中列举的不同，它没有定义数据域。

这正是 Linux 内核在设计和实现时巧妙、灵活的应用，如果我们在内核中也使用教科书中的经典定义，由于每个链表中数据域的数据类型只能是一种，而内核中可能要用到多种数据类型的链表，那么，我们只能为每种数据类型定义一个链表，这样就使得内核的设计不够灵活，不容易扩展，为此，内核骇客提出了引以为豪的、颠覆性的设计：不在链表中包括数据，而是将链表的节点包含到数据结构里边。

这样，只需在需要连接的数据结构中包含 struct list_head 成员，就可以这个数据结构的不同节点连接起来，而不需要考虑每个节点的数据类型。例如在

/include/linux/sched.h 定义的 task_struct 结构中就包括：

```
struct task_struct {
    ……
    //系统中全部进程通过它构建一个双向循环链表，其根节点是 init_task
    struct task_struct *next_task, *prev_task;
    struct list_head tasks;                    //全部进程的描述符通过它链接起来
    struct list_head local_pages;              //本地页面链表
    struct list_head thread_group;             //线程链表
    ……
};
```

可见，task_struct 结构被包含到了多个链表当中。因此，Linux 当中就不用定义很多种类型的链表，通过这个全能的模板就能定义所有的链表。Linux 简洁实用和极为标准的编程作风，通过这一点的到淋漓尽致地体现，它们的大体结构如图 2-4 所示。

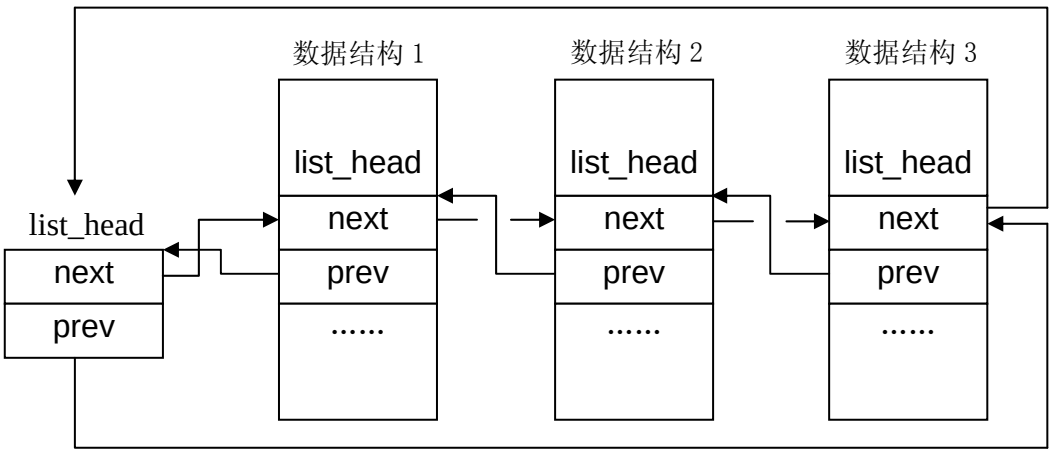


图 2-4 用 list_head 构建的双向循环链表

在具体实现中，Linux 仅仅定义了链表节点而没有专门去定义头节点，它是通过宏定义 LIST_HEAD()来完成的，它的大体操作相当于：

```
#define LIST_HEAD(name) struct list_head name ={ &(name), &(name) }
```

可见用 LIST_HEAD(list_name)定义一个 list_name 的头节点时，新变量 list_name 的类型就是 list_head，它用作新创建的链表头的占位符，是个哑元素。在一开始时链表的 next 指针和 prev 指针中都存放了链表自身的地址，这样就得到了一个空链表，Linux 内核用链表节点的指针 next 中存放的是不是自身的地址来辨别该链表是不是空链表。

此外，Linux 还可以用 INIT_LIST_HEAD 宏对一个链表进行一些基本的赋值操作，大体实现如下：

```
#define INIT_LIST_HEAD(ptr) do {(ptr)->next = (ptr); (ptr)->prev = (ptr);}while(0)
```

即让一个节点的前序和后续彼此相指。

我们用 INIT_LIST_HEAD(&run_list)就可以创建运行队列的链表了。

为了便于快速的操作，Linux 特地定义了几个宏操作，用了对链表进行遍历操作。为了弄明白这几个宏，先来搞清楚如何从链表中提取出正确的数据项。

因为 Linux 链表中只是记录了数据节点的 list_head 变量地址，为了访问链表中的所有者链表中的所有节点，Linux 定义了 list_entry(ptr,type,member)宏，它的返回值是类型为 type 的数据结构地址，其中 type 数据结构中包含 list_head 字段，ptr 中保存了 list_head 变量的地址，即指向存储在链表中的地址，type 表示数据节点的类型，member 是数据节点中定义的 list_head 的变量名，当要访问运行队列链表中的首个变量时，操作如下：

```
list_entry(current->tasks.next, struct task_struct, tasks);
```

list_entry 的使用不是特别难，它的实现如下：

```
#define list_entry(ptr, type, member) ({\
        const typeof( ((type *)0)->member ) *__mptr = (ptr);  \
        (type *)( (char *)__mptr - ((size_t) &((type *)0)->member));})
```

在这里要靠编译器原理的知识，首先算出成员在其 struct 中的偏移量，然后再用变量的地址求出该 struct 的地址值。

其中(type *)0)->member)操作不仅仅是用于链表中的一些计算，其中有趣的是在((type*) 0)->member 操作中先把数值 0 的类型设置为 type 类型的结构指针，然后访问其中的 member 成员，其中用到了 typeof()宏操作，用来得到 member 变量的数据类型；而宏 offsetof()得出的是 member 变量参照于该结构的偏移。

还有一个非常有用的宏就是 list_for_each()宏，其定义如下：

```
#define list_for_each(pos, head) \
    for (pos = (head)->next, prefetch(pos->next); pos != (head);
        pos = pos->next, prefetch(pos->next))
```

可以看出它实际上是利用一个循环实现的，其中输入参数 pos 作为循环的对象，并通过它得到 list_head 的地址，head 作为起始的链表头，沿着 next 的方向挨个访问，直至又回到起点（其中 prefetch()用作预取，来提高遍历速度）。

然而在通常的使用中，遍历操作时也可能会请求得到节点中的数据项，即 list_

for_each()和 list_entry()组合在一起进行操作。为此 Linux 又定义了一个 list_for_each_entry()宏，它跟 list_for_each()明显的不同，此时的 pos 是类型是不一样的，它所记录的是 list_head 的数据结构的地址，而不是 list_head 结构的地址。

2.5 current 进程剖析

在系统的运行过程中，经常需要知道现在是哪个进程在执行，为此，内核中定义了一个宏 current，它贯穿在内核的源码中，它的作用就是提供指向目前正在运行的进程描述符的地址。

Linux 把内核态的堆栈和线程描述符 thread_info 两个数据结构紧凑的存放到一段连续存储区域[25]，这样的好处是内核能够轻易地利用 esp 的值计算出当前 CPU 正在运行的进程。在 2.6 版本的内核中允许内核堆栈的大小可以在 4K 和 8K 两者间选择，其中此堆栈与用户态的进程所使用的不同，当内核态的进程访问内核的数据时要使用内核态的堆栈。

可见，系统空间的堆栈不同于用户空间的堆栈那样在运行时临时分配，而是一开始就设定好的。由于堆栈的尺寸固定且较小，内核要采取一些额外的措施防止溢出，如不能让中断服务程序、设备驱动程序等中的函数嵌套过深，函数中的局部变量不能使用过多等。内核堆栈大小为 4K 的示意图如图 2-5 所示。

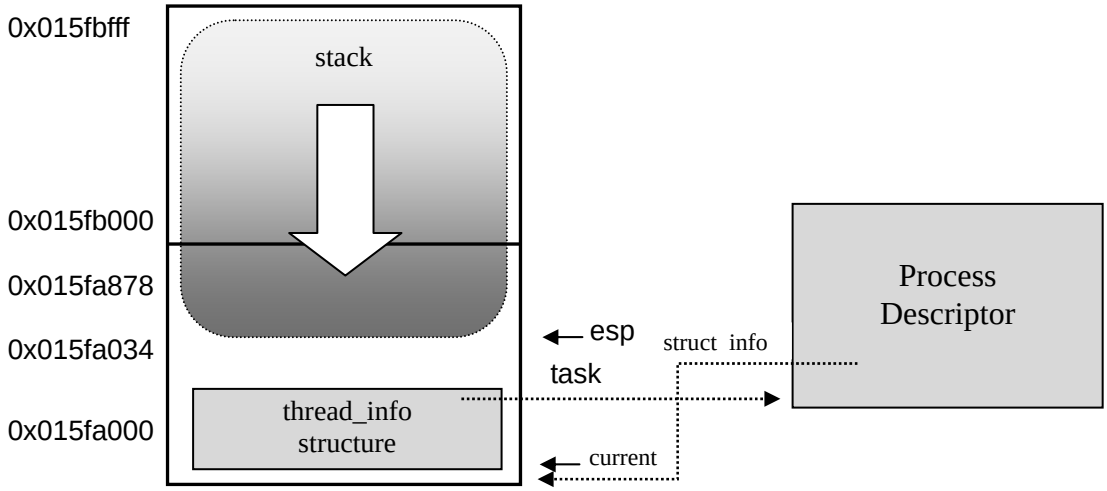


图 2-5 thread_info 结构图

实现此内核配置的源代码位于 thread_info.h 头文件中：

```
#ifdef CONFIG_4KSTACKS
#define THREAD_SIZE              (4096)
```

```
#else
#define THREAD_SIZE                (8192)
```

其中 THREAD_SIZE 大小的不同会直接影响 current 宏的实现，我们来剖析一下 current 宏操作的实现[26]。

```
#define current get_current()  /*即把一个函数赋值给了 current，并且它不传入任
                                何的参数。*/
//把 get_current 定义为静态的内联函数。
static inline struct task_struct * get_current(void)
{
    //此处要有到 thread_info 结构体。
    return current_thread_info()->task;      //返回一个结构体的一个指针变量
}

//定义的函数类型是结构体指针型。
static inline struct thread_info *current_thread_info(void)
{
    //声明了一个 thread_info 结构体的指针。
    struct thread_info *ti;
    //使用汇编语言给 ti 赋值。
    __asm__("andl %%esp,%0; ":"=r" (ti) : "" (~(THREAD_SIZE - 1)));
    return ti;      //把结构体指针当成返回值返回
}
```

而 thread_info 结构的定义又如下：

```
struct thread_info {
        struct task_struct      *task;    //存放着相应进程 task_struct 的地址。
        struct exec_domain      *exec_domain; //一个进程的执行域。
        ... ..
}
```

可见，利用 CONFIG_4KSTACKS 的不同，当 THREAD_SIZE 为 8K 时，屏蔽掉内核栈地址的低 13 位，当 THREAD_SIZE 为 4K 时，则屏蔽掉内核栈地址的低

12 位，从而得到 thread_info 的基地址。而 task 字段又在 thread_info 结构体中的偏移量为 0，因此该基地址所指向的空间中就包含了当前 CPU 正在运行进程的描述符。

2.6 switch_to 宏剖析

在系统当中任务的数目一般都是比处理它们的 CPU 要多，所以不能让一个任务长期占用着 CPU，在必要的时候要进行切换操作。

虽然每个进程都有各自的内存区，然而它们却要使用数量比进程少的 CPU 资源，所以在进程发生切换时，系统要对之前进程所用的一些寄存器的值记录下来。这些值称为硬件上下文，它包括了进程在执行时所需的全部信息。Linux 把硬件上下文分为两部分存放，其中进程的描述符中有一个 thread_struct 类型的 thread 字段指向的结构保存了该进程大部分的硬件上下文，eax 和 ebx 等中的值则保存在内核态堆栈中[27]。

进程的切换一般都是通过 schedule()函数来实现的，而从实现的本质上讲，一个进程的切换主要有两部分组成：

1. 切换页的全局目录，开始一段新的内存空间；这部分内容主要涉及内存管理，本文在此不做过多的介绍。
2. 切换进程的硬件上下文和内核态的堆栈，即切换进程运行所需的所有信息；这部分主要由 switch_to 宏执行。

switch_to 宏是一个与硬件关系紧密的例程，它主要完成堆栈的切换，它的定义如图 2-6 所示。

这段程序虽然不长，但是却实现了重要的操作。可以看出，该宏有三个参数，第 6 行语句把当前 CPU 的 ESP，即当前进程的系统空间堆栈的指针存入 prev->thread.esp 中，接着，又把 ESP 寄存器的值设置为受到调度的新进程 next 的堆栈指针，现在 CPU 的堆栈指针指向新进程的堆栈了，即完成了堆栈的切换。

假设有两个进程 A 和 B，prev 存放进程 A 的地址，next 存放进程 B 的地址，在本次切换的过程中 A 就是被切出的进程，B 为被切入的进程，在运行第 7 行语句之前使用的是 A 的堆栈，但当运行完第 7 行语句后就开始使用 B 的堆栈了，即当前进程是 B 而不再是 A 了，因为上边我们已经分析了，current 进程的实现实际上就是根据 CPU 当前的堆栈指针 ESP 计算得出的。第 4、5 两行 push 进 A 的系统

```
1   #define switch_to(prev, next, last)                     \
2   do {                                                    \
3       unsigned long ebx, ecx, edx, esi, edi;              \
4       asm volatile("pushfl\n\t"       /* save    flags */  \
5                    "pushl %%ebp\n\t"      /* save    EBP    */   \
6                    "movl %%esp,%[prev_sp]\n\t"   /* save    ESP    */ \
7                    "movl %[next_sp],%%esp\n\t"   /* restore ESP    */ \
8                    "movl $1f,%[prev_ip]\n\t"   /* save    EIP    */ \
9                    "pushl %[next_ip]\n\t"   /* restore EIP    */   \
10                   __switch_canary                         \
11                   "jmp __switch_to\n"     /* regparm call   */   \
12                   "1:\t"                          \
13                   "popl %%ebp\n\t"        /* restore EBP    */   \
14                   "popfl\n"               /* restore flags */\
15                   : [prev_sp] "=m" (prev->thread.sp),     \
16                     [prev_ip] "=m" (prev->thread.ip),     \
17                     "=a" (last),                          \
18                       "=b" (ebx), "=c" (ecx), "=d" (edx),     \
19                       "=S" (esi), "=D" (edi)                  \
20                       __switch_canary_oparam                  \
21                   : [next_sp]   "m" (next->thread.sp),    \
22                     [next_ip]   "m" (next->thread.ip),    \
23                       [prev]        "a" (prev),           \
24                       [next]        "d" (next)            \
25                       __switch_canary_iparam                \
26  } while (0)
```

图 2-6 Swich_to 的源码实现

堆栈，而第 13、14 两行却从进程 B 的堆栈 pop 回来了，实际上第 13、14 两行是在恢复新切入的进程 B 在之前被换出时 push 进系统堆栈的内容。

现在的进程还不能执行，显然切换还没有完成。第 8 行把标号为“1”所在的地址保存到 prev->thread.eip 中，即保存的是第 13 行 pop 指令所在的地址，把它作为进程 A 下一次被切入时的返回地址。接着，又把 next->thread.eip 压栈，再执行一个__switch_to()函数。当 CPU 运行到__switch_to()函数里的 ret 指令时，因为是利用 jmp 指令跳转过去的，最后入栈的 next->thread.eip 将作为返回的地址，又因为每个进程被换出时都执行了这里的第 8 行，这就意味着每个进程在被恢复调度时都是从第 13 行开始执行的。但有一种例外，就是新创建的进程在第一次执行时，并没有经历一次被换出，即没有执行过这里的第 4 到第 8 行，所以要将其 task_struct 中的 thread.eip 预先设置为 ret_from_fork，最终返回到了用户空间中。

在进程的切换过程中，涉及到的进程是三个而不是两个，所以要用到第三个

参数 last。假设内核切进程 A 而切入进程 B 执行，此时 prev 指向 A 而 next 指向 B。随后内核恢复执行进程 A，那么必须挂起当前正在执行的进程 C，而此时的进程 C 可能不再是进程 B 了，于是 prev 指向 C 而 next 指向 A，以这样的参数运行 swith_to 宏，当进程 A 恢复执行时，会找到自己原先保存的堆栈，于是 prev 还是指向 A，next 指向 B，此时的内核就没有任何关于进程 C 的引用了，所以 last 参数作为一个输出参数，用来记录进程 C，并将 C 的地址保存到内存当中。

2.7 本章小结

本章首先讲述了 SMT 和 SMP 多核技术，接着介绍了进程、线程的基础知识，以及调度算法、调度策略，最后剖析了 Linux 源码中的几个基本的结构和宏操作，为下一步的源码研究做好准备。

第三章　CFS 调度系统剖析

作为自由软件和开发源码的典型代表，Linux 受到了全球程序员和用户的追捧，得到了丰富而快速的发展，本课题就是以 Linux 2.6.36 内核版本为基础展开讨论的。一个操作系统的主要包括：进程管理、内存管理、文件管理、设备管理几大部分，其中进程的管理和调度直接决定了系统的性能和效率，本章我们就开始着重介绍 Linux 2.6.36 里的几个重要的数据结构和函数，并对调度系统的机制做深入地剖析。由于 Linux 内核对进程和线程没有明确的加以区分，有时还把它们统称为任务，所以在本文里出现的“进程”、“任务”和“线程”都表示相同的含义。

3.1 CFS 基础

从 Linux kernel 2.6.23 开始，引入了由 Ingo Molnar 提出的完全公平调度 CFS[28]，以优异的性能，替换掉了 O(1)调度，它的目的是为了让进程能够更加公平地、合理地使用 CPU。它的设计初衷就是：在真实的硬件基础上模拟一个更准确的、更理想的多任务 CPU，这就意味着 CPU 能以同样的处理能力去运行每个任务，即如果系统中两个进程在运行，那么它们各占用一半的 CPU 资源（在不考虑优先级的情况下）。

在设计 CFS 调度器时，关于实时进程的调度没怎么修改，而批处理任务跟普通进程的调度进行了较大的修改，主要体现在三个方面：模块化的接口、调度器和组调度。

模块化的接口设计：CFS 调度系统使用了模块化的设计方式，可以有多种的调度算法注册成当中的一个小模块进行运作，因此，系统中就可以存在多种不同的调度机制，即便将来想添加新的算法，也无需修改调度系统核心部分部分实现，只需有针对性的修改相应的函数接口即可。在目前的 Linux 系统中，CFS 调度系统配备了 CFS 和实时调度两种机制，用来应对普通进程跟实时进程的管理和调度，它们的实现代码分别在 kernel/sched_fair.c 和 kernel/sched_rt.c 中。

调度器：对于交互式任务和批处理任务，CFS 使用了新的调度策略设计了调度系统，它采用了非常简单的算法来实现，并且不用考虑进程睡了多久，不再对进程的种类进行划分等等，CFS 希望根据对 CPU 资源的需求程度来选择要运行的

进程，来确保进程的到 CPU 资源的公平性。

组调度[29]：引入组调度是对用户进行划分，以保证用户之间能公平地合理地使用 CPU 资源。

其中编译选项 CONFIG_GROUP_SCHED 用作 CFS 组调度，用于开启 CFS 组调度的功能。

3.1.1 CFS 设计原理

完全公平调度算法的设计宗旨就是让系统中的所有进程能够公平的使用 CPU 资源，但这里所说的公平并不是指调度器把 CPU 时间平均地分配给每个任务，如果那样，则会造成紧急任务以及权重高的进程得不到及时响应，系统的能力也大大下降。并且 CFS 关重的是一段时间的公平性，随着时间的推移，它能够更好的让多个任务的到兼顾。

CFS 使用了虚拟运行时间（virtual runtime）来判断该让哪个进程去运行。在发生调度时[30]，CFS 总是选择虚拟运行时间小的任务，即选择执行时间最短的任务运行，以此来平衡各个进程的执行时间，在后面的源码的解析中，本文将陆续的对它进行细致的介绍。

虚拟运行时间在 CFS 中是一个重要的判断标准，调度系统利用它来做出判断，决定哪个任务执行，CFS 的公平性也是通过它来贯彻实行的，一个任务执行的时间越短，就逐渐提升它的优先等级，即使在一开始时某个任务的优先级很低，它在等待了一段时间后，也能得到 CPU 资源进行执行。

CFS 为了区别对待不同优先等级的进程，在计算进程的虚拟运行时间时把进程的优先级也考虑在内，实现方法是：让进程的虚拟时间增长与优先等级成反比，即让高优先级的任务的虚拟时间的增长速度小，相反，低优先级的任务的虚拟时间增长速度大，这就兼顾了公平，也对紧急情况不同的任务进行了区别的对待，不至于一味的追求公平而使得紧急的任务得不到及时的处理。

CFS 与 O(1)调度系统不同之处还体现在运行队列的管理上，CFS 使用红黑树算法来管理系统中进程，红黑树把任务的虚拟运行时间看出参照标准，并根据这个值来决定该进程在红黑树中的位置，在安排时，让红黑树左边节点上的任务比红黑树右边上任务的虚拟运行时间小。随着时间的推移，通过不断的更新虚拟运行时间，让变大的任务向右挪，这样就让原来虚拟时间大的任务也能分得 CPU 资源，这就保证整个系统正常地运行。并且进程也不像以前那样根据时间片维护在

运行队列中，而是维护在红黑树中，通过结构体 struct cfs_rq 来保存的，它的主要成员如表 3-1 所示。

表 3-1 cfs_rq 结构体核心成员变量及注解表

成员变量	主要作用及含义
struct load_weight load;	记录着本队列的负载情况。
unsigned long nr_running;	队列上可运行的总进程数。
u64 min_vruntime;	队列中最小虚拟时间的值。
struct rb_root tasks_timeline;	存放着根节点的地址。
struct rb_node *rb_leftmost;	存放着红黑树中最左边的节点的地址。
struct list_head tasks;	用于构成双向循环链表
struct sched_entity *curr, *next, *last;	指向相应的运行实体
当开启编译选项 CONFIG_FAIR_GROUP_SCHED 时	
struct rq *rq;	存放着跟该 cfs_rq 相对应的运行队列的地址。
struct task_group *tg;	指向此运行队列的调度组。

总之，CFS 调度系统先选择当前系统中虚拟运行时间小的进程运行一段时间，并把运行的时间根据该进程的优先级换算成虚拟运行时间，并把它累计到该进程的总虚拟运行时间中，倘若该进程总的虚拟运行时间不再是最小的，那么会有其他的任务跑到红黑树的最左边上，所以在发生调度时，它就会被选中去占用 CPU 资源，进而发生抢占。

3.1.2 CFS 组调度

试想，当前系统有两个用户 A 和 B，A 运行了 3 个进程，A 运行了 7 个进程，假设它们的优先级一样大，调度系统能够使得每个进程占用 10%的 CPU 时间，但是用户 A 只分得了 30%而 B 却分得了 70%，倘若 B 陆续有新的任务添加进来，就会造成 A 所分得的 CPU 时间越来越少，对于 A 而言，有失公平性。组调度就是为了解决这一问题而提出的，用来用户之间占用 CPU 的公平性。Linux 内核使用结构体 task_group 来描述组调度。如果需要发生调度，CFS 会先从各个调度组之间做出判断，再来针对某个调度组进行判断。对此常用的策略有 SCHED_BATCH 和 SCHED_NORMAL 两种。

3.1.3 红黑树简介

红黑树[31]是节点带有颜色属性的平衡二叉树，并能平衡索引，所以它的性能要优于一般的二叉树，它的时间复杂度是 O(log n)（n 指树中节点的数目）。具体的操作规则如下：

（1）树中的结点要不是红色要不就是黑色，没有其他的选择；

（2）树中存在一个红结点，那么它的全部孩子必都是黑的；

（3）树中的全部叶子都是黑色；

（4）从树的根节点向叶节点的所以路径中，所包括的黑节点数必须相同。

3.2 进程在 CFS 中的表示

在前面我们已经讲解过 task_struct 结构体，它是一个进程在内核态的具体描述，记录着进程本身息息相关的重要信息，譬如该进程号、所使用的内存地址空间等等，然而我们在阅读该结构时发现与 CFS 调度系统相关的几项，如表 3-2 所示。

表 3-2 task_struct 结构体核心成员变量及注解表

成员变量	主要作用及含义
volatile long state;	记录着进程当前的状态；
unsigned int flags;	进程的标志位；
int lock_depth;	大内核锁的深度；
int prio, static_prio, normal_prio;	进程的动、静态优先等级；
struct sched_class *sched_class;	该进程所使用的调度类；
struct sched_entity se;	该进程所对应的调度实体；
unsigned int policy;	进程的调度类型
cpumask_t cpus_allowed;	描述哪些 CPU 能执行该进程
struct mm_struct *mm, *active_mm;	进程及内核线程的内存区的地址
pid_t pid;	该进程的进程号

并且我们看出，与先前的 O(1)调度系统相比，现在的结构体里边没有再利用 struct prio_array 来管理各个任务，并添加了两个结构体作为成员变量，作为调度实体的 sched_entity 以及作为调度类的 sched_class。

sched_entity 囊括了进行调度全部的相关内容。CFS 调度器并不是直接跟进程进行交互，而是跟调度实体交互，这样可以便于实现组调度：把所有的进程放在

不同的组中，CFS 先确保这些组间的公平，然后再来考虑组内的各个任务。该结构体中与本文相关的成员如表 3-3 所示。

表 3-3 sched_entity 结构体核心成员变量及注解表

成员变量	主要作用及含义
struct load_weight load;	记录着负载，是平衡操作的依据；
struct rb_node run_node;	用来描述红黑树的节点，以便把实体放到红黑树中；
unsigned int on_rq;	用来判断是否在 CPU 队列中；
u64 sum_exec_runtime;	记录着总的运行时间；
u64 vruntime;	保存着该 entity 对应进程的总虚拟运行时间。

3.3 CFS 模块化设计

CFS 调度系统使用了模块化的实现方法，这并不是意味着将调度系统划分成许多的模块进行加载，而是指在编写代码上的模块化，这也有利于跟其他调度模块进行并行处理，也使得调度系统便于扩展。

其中 CFS 中模块化最典型的代表就是调度类，在 Linux kernel 用 sched_class 结构体来描述。它使用模块化的设计思想，对调度策略进行了封装，并为 Linux 执行进程调度给出了必要的方法接口函数，这样就可以根据不同的系统的需要，编写各异的、相适应的函数接口，通过在初始化时对它们进行赋值，就可以直接调用了，sched_class 结构体如表 3-4 所示。

表 3-4 sched_class 结构体核心成员变量及注解表

成员变量	主要作用及含义
void (*enqueue_task)(struct rq *rq, struct task_struct *p, int wakeup);	将进入运行状态的实体放入红黑树中，并把节点计数器加 1。
void (*dequeue_task)(struct rq *rq, struct task_struct *p, int sleep);	把退出运行状态的实体从红黑树中移除，并把节点计数器减 1。
struct task_struct * (*pick_next_task) (struct rq *rq);	该函数用于挑选将要运行的任务。
void (*put_prev_task)(struct rq *rq, struct task_struct *p);	存放着现在运行的任务，如果已经执行完了就退出队列，否则再放到红黑树中。
void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);	时钟中断通过它进行任务的查看和调度。

如此一来，我们只要按照系统具体的情况，编写相应的函数实现，就可以给调度系统提供针对某一操作的具体实现，当把 CFS 运用到其他体系结构的系统中时，我们只需按照该系统的要求，改写调度类的实现，就能够将 CFS 应用到相应的系统当中，这就能在很大程度上提升了可扩展空间。

3.4 CFS 代码剖析

以上我们介绍了与 CFS 调度系统相关的几个非常重要的结构体，下面本文以 Linux kernel 的调度过程着手进行分析 CFS 的源代码，借此来搞明白 CFS 的实现过程。

在 Linux 中进程发生切换时常会出现，它们主要通过两个步骤来完成：第一步，通过时钟中断来触发更新检查机制，看看当前的进程是不是需要发生切换；第二步，利用 schedule()来完成最后的切换工作。

3.4.1 更新与查看

在 Linux 系统中，通过时钟中断引发进程调度经常发生，因为它是伴随着时钟滴答运行的，所以具有运行的机会比较高，它主要是利用 scheduler_tick()来实现的，那么本节就从此函数开始剖析，沿着它的流程，分析 CFS 调度系统在代码层的实现。

每当一个时钟到来时，负责时钟中断的处理函数 timer_interrupt()就会调用函数 do_timer_interrupt_hook()，do_timer_interrupt_hook()又会调用函数 do_timer()和 update_process_times()，其中 update_process_times()主要负责更新进程时用到的一些与时间相关的字段，其中最为重要的就是调用函数 scheduler_tick()更新时间片的剩余节拍数，该函数主要负责管理内核中与整个系统调度相关的统计信息，并负责激活当前进程调度类的周期性调度方法，其流程如图 3-1 所示。

该函数主要是做了一些与时间更新的工作，并获取了当前 CPU、运行队列，再获取运行队列当前正在执行的 current 进程的地址信息，进而调用了该进程 sched_class 中的 task_tick()，我们在上面讲到，它是 CFS 系统中的模块化实现的一个方法，那么接下来探究它在 CFS 调度系统中是通过哪个函数与之对应的。我们发现在 Linux/kernel 中的 sched_init()函数有这样一个赋值操作：

```
current->sched_class = &fair_sched_class;
```

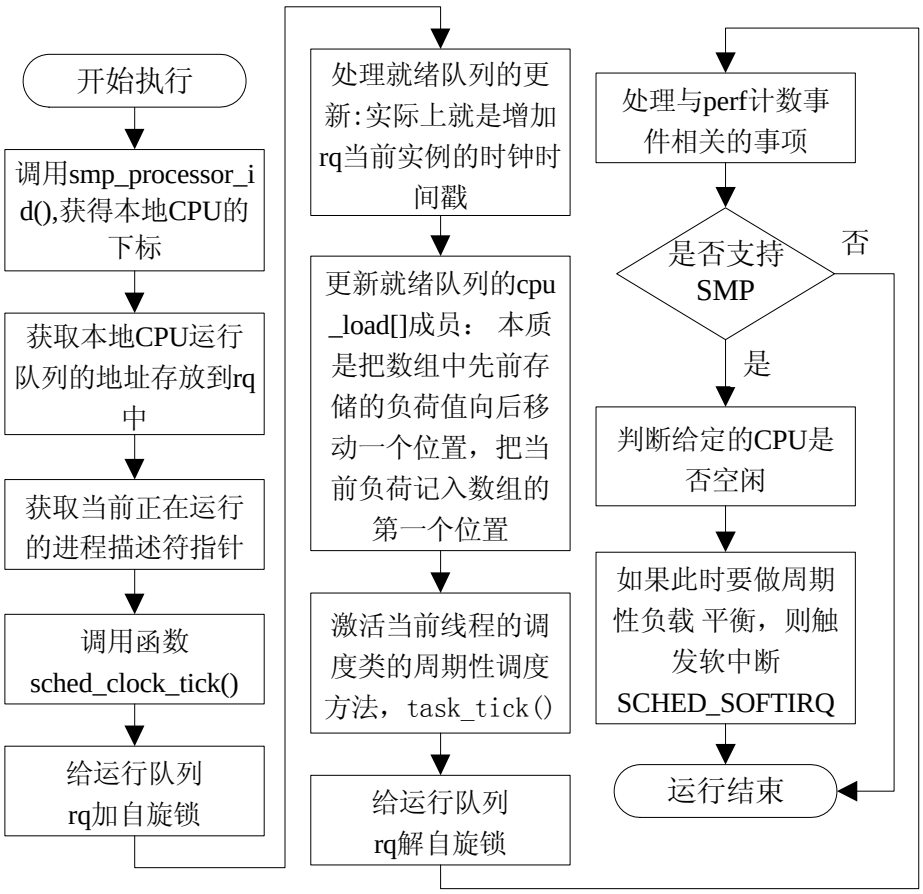


图 3-1 根据源码画出的 scheduler_tick()流程图

即在调度初始化时，给当前运行的进程的调度类进行了赋值，而 fair_sched_class 的几个与本文相关的操作如下：

```
static const struct sched_class fair_sched_class = {
    .pick_next_task = pick_next_task_fair,
    .put_prev_task = put_prev_task_fair,
    .task_tick = task_tick_fair,
    ……
};
```

如此以来，在 CFS 调度系统中用 task_tick_fair 来为接口 task_tick 提供具体的操作[32]，它的核心代码如下：

```
struct cfs_rq *cfs_rq;//声明一个结构体。
    struct sched_entity *se = &curr->se;//获取当前的实体。
    for_each_sched_entity(se) {//对结构体进行遍历。
        cfs_rq = cfs_rq_of(se);//得到实体对应的队列。
```

```
entity_tick(cfs_rq, se, queued);
……
```

此函数的功能和流程都不复杂，首先把当前进程的实体存放到局部变量 se 中，接着再遍历所有的进程实体，在遍历过程中，把每个实体中与红黑树相关信息取出并存放在局部变量 cfs_rq 结构体中，最后，用局部变量 cfs_rq、se 等作为参数调用函数 entity_tick()。而 entity_tick()的工作流程主要是利用三个核心函数来实现的，一个是负责更新当前的 cfs_rq 和进程的信息 update_curr()函数，一个是 resched_task()，最后一个是负责检查是否当前需要发生抢占的函数 check_preempt_tick()。

函数 update_curr()负责对时间点从上次时钟滴答以来的更新工作，它的核心函数是__update_curr()，并且在最后更新该进程的开始运行时间，它的主要实现过程如下，

```
{
    ……
    delta_exec = (unsigned long)(now - curr->exec_start);
    ……
    __update_curr(cfs_rq, curr, delta_exec);//进行一些标记的更新工作。
    curr->exec_start = now;//获得现在的时间戳。
    ……
}
```

其中 delta_exec 用来记录从最后一次修改负载记录后当前任务所占用的运行总时间，即计算当前进程的执行时间。

函数__update_curr()是整个 CFS 虚拟运行时间计算的真正核心，它根据当前可运行进程总数对运行时间进行加权计算，计算出该进程的虚拟运行时间，并更新一些变量，其核心源代码如下：

```
{
    ……
    //记录当前任务一共的物理执行时间。
    curr->sum_exec_runtime += delta_exec;//主要是进行累加的操作
    //用优先级和 delta_exec 来计算 delta_exec_weighted 以用来更新 vruntime
    delta_exec_weighted = calc_delta_fair(delta_exec, curr);
    //更新进程的虚拟运行时间
    curr->vruntime += delta_exec_weighted;
```

```
    update_min_vruntime(cfs_rq);
}
```

可以看出，任务所使用的 CPU 时间存放到对应实体的 sum_exec_runtime 中，而该任务的虚拟运行时间是通过它运算得出的。此外该函数还对其他的一些变量进行了更新操作。

一个进程的虚拟时间的计算就是通过函数 calc_delta_fair()来完成的，该函数会首先判断进程的优先级是否为 0，倘若为 0 就把 delta_exec 作为返回值，否则把函数 calc_delta_mine()的结果作为返回值，它的参数分别是当前进程的运行时间 delta_exec、该任务的指针 curr 跟系统默认的权值 NICE_0_LOAD，它的具体执行就是运算：delta_exec_weighted =delta_exec * NICE_0_LOAD / curr->load.weight。其中 curr->load.weight 指的是该任务的权重，它就是表示 nice 对应的值，并且 nice 跟权重值成反比。

根据 Linux 的实现原理，nice 可以取从-20 到 19 之间的数字，并且它跟它相应的 weight 值有一定的对应原则：当 nice 增加 1 时，与它相应的 weight 就变为原来的 90%左右，根据 Linux 系统默认的标准，当 nice 等于 19 时，它相应的 weight 值就等于 15，而当 nice 等于-20，它相应的 weight 值就等于 88761。

据此，我们可以得出一个结论：在多个任务运行了相等的墙上时间时，优先级高的任务的 delta_exec_weighted 比优先级低的任务的要小，并且该值会被累加到该进程的虚拟运行时间中，通过前面的学习我们已经知道 CFS 就是通过这个变量进行判断的，而这种原则会得出优先级最大的进程会被放到 rb_tree 的左边，以便在下一次选择时被选中运行。

当函数 update_curr()执行完毕后，接着会判断输入参数 queued，而它在调用 task_tick()时传入的是 0，所以不会执行 resched_task()。

最后 entity_tick()还调用了 check_preempt_tick()来查看当前的进程是不是有必要被一个新的任务所抢占，它的核心操作是：

```
{
    ……
    //得出进程在当前的负载情况下允许运行时间片的大小
    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    //倘若该进程已运行的物理时间多于上边计算出的时间片
    if (delta_exec > ideal_runtime) {
```

```
            //标记该进程，需要发送切换。
            resched_task(rq_of(cfs_rq)->curr);
            ……
        }
        ……
        /*倘若当前 CPU 上的可运行进程数多于 1，则还要计算出当前正在运行进
          *程的虚拟运行时间跟下一个要运行的进程的虚拟运行时间的差，并将该
          *值与当前负载状况下的允许运行时间片比较，如果大，则也需将可抢占
          *标志置位*/
        if (cfs_rq->nr_running > 1) {
            struct sched_entity *se = __pick_next_entity(cfs_rq);
            s64 delta = curr->vruntime - se->vruntime;
            if (delta > ideal_runtime)
                resched_task(rq_of(cfs_rq)->curr);
        }
    }
```

3.4.2 schedule()分析

调度程序主要由 schedule()函数实现，它主要通过两种途径实现：直接调用和延迟调用。

直接调度：当 current 因为不能获得必要的请求时，系统将阻塞该进程，此时一般采取直接调用的方式，首先将 current 进程转移到相应的等待队列中，再把 current 进程的 state 状态字段设置为 TASK_UNINTERRUPTIBLE 或者 TASK_INTERRUPTIBLE，最后调用函数 schedule()。

延迟调度：通过把 current 的 TIF_NEED_RESCHED 标志位置为 1，而系统在检查该标志时，来决定是否发生切换，所以系统会在不久后调用 schedule()函数。常见的使用该方式的场合有：

1. 当 current 用完了自己的 CPU 时间片时，通过 scheduler_tick()设置该标志。
2. 当一个优先级比 current 的优先级高的进程被唤醒时，函数 try_to_wake_up()设置 TIF_NEED_RESCHED 标志。

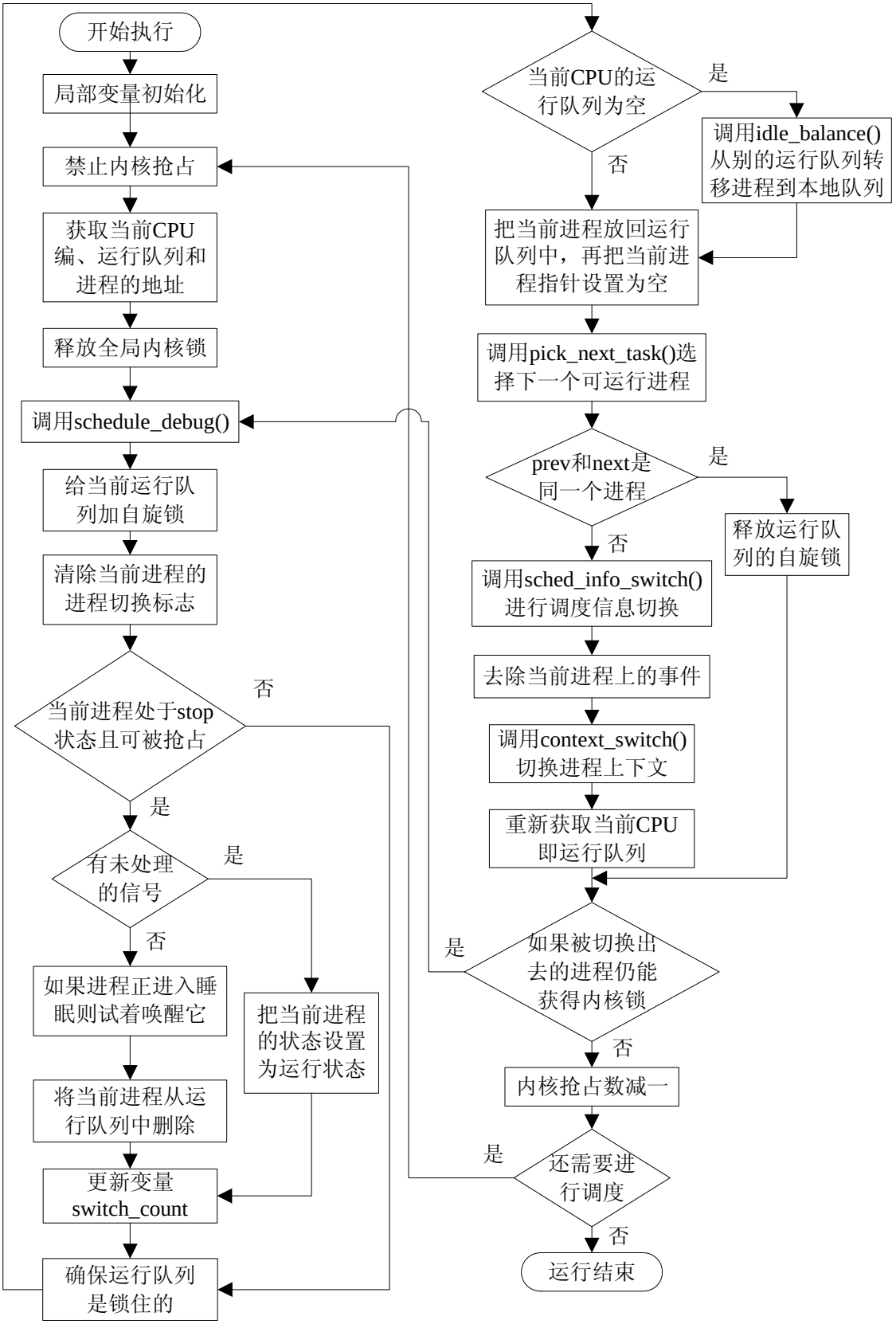


图 3-2 根据源码画出的 schedule()函数流程图

3. 当调用 sched_setscheduler()函数时。

schedule()函数的主要工作流程是先从将该进程退出相应的运行队列，然后计算、更新调度实体的相关调度信息，再根据进程的虚拟运行时间把当前进程重新插到相应的调度运行队列中（对于实时调度，则插到对应优先级队列的队尾），最后从运行队列中选择要运行的下一个进程，发生切换。当切换后，本次调度就算结束了，此时系统中的 current 进程就是切换后新的进程了。它的流程如图 3-2 所示。

由于采用了模块化设计，schedule()函数中两个重要函数 put_prev_task()和 pick ext_task()，对应于 CFS 算法的模块实现分别是函数 put_prev_task_fair()和 pick_next _task_fair()。

3.4.3 与 schedule()相关的函数分析

schedule()函数调用 put_prev_entity()进行处理，主要是为了将任务再插入到相应的队列中，毕竟有的时候，实体会从运行队列脱离出来。其中把一个进程实体插入红黑树是通过调用函数__enqueue_entity()来完成的，__enqueue_entity()函数的工作是对系统中的调度实体进行遍历，并在遍历过程中对每个进程实体插入到红黑树中的相应位置。它的核心代码是：

```
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    s64 key = se->vruntime - cfs_rq->min_vruntime;
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (key < entity_key(cfs_rq, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    if (leftmost)
```

```
            cfs_rq->rb_leftmost = &se->run_node;
        rb_link_node(&se->run_node, parent, link);
        rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
    }
```

代码分析：该函数首先通过一个 while 循环在红黑树中找到该进程实体合适的位置，如果是树中最左边的位置，则把它保存到 cfs_rq->rb_leftmost 成员变量中，以方便以后得查找，最后在对该节点执行插入操作，这样就把该实体插到在第一步中找到的位置中了。

当现在我们已经看出，CFS 调度系统算法的重点是选择虚拟运行时间最短的任务。运行队列用红黑树的方式管理，其节点的键值就是进程的虚拟运行时间，CFS 调度器所选的下一个进程，就是所有进程中虚拟运行时间最小的那个，即树中最左侧的叶子节点。这就是通过函数 pick_next_task_fair()完成的，实质的工作是通过 pick_next_entity()函数实现的，有趣的是，该函数不必遍历整个树，因为该值在插入树时已经被缓存在 cfs_rq 结构体的字段 rb_leftmost 当中。

3.4.4 概括 CFS 处理流程

通过剖析，我们发现 O(1)调度中的运行队列数组、进程类型的区分、基于优先级的时间片分配等没有了，而增加了模块化的设计框架、组调度等，每个进程分配到的时间不再是一个基于进程优先级的常数，而是一个依据系统运行情况的一个变数—虚拟运行时间，CFS 用它来追踪每个实体中的进程，它的计算方法是：

$$\text{虚拟运行时间} += \frac{\text{进程运行的墙上时间}}{\text{进程的权重}} \times \text{NICE\_0\_LOAD}$$

公式里的 NICE_0_LOAD 表示系统默认的权重大小，即当 nice 等于 0 时。CFS 调度系统总是让虚拟运行时间最短的任务投入执行，由于算式中进程的权重处在分母上，权重大的进程 vruntime 的增加速度就小，这样就区别对待了优先级不一样的任务。

通过剖析我们发现，CFS 的处理流程主要分为两个部分，一个是对进程虚拟时间的更新，并在必要时标记切换标准，它是通过 scheduler_tick()触发的；另一部分就是实际的切换操作，它是通过 schedule()函数实现的。

如果低优先级的进程陷入了阻塞，其虚拟时间也会变得比其他进程要小，从而也能够得到执行，使得非互动性的任务不至于在很长的时间内都得不到执行。

为了能够在众多进程中快速的选中合适的进程，CFS 用红黑树来管理系统中的任务，把每个任务作为树的节点、以虚拟运行时间的大小作为排序依据挂到红黑树中，并随着 vruntime 的改变而做出适当的调整。

文献[33]对 CFS 调度和 O(1)调度做了关于公平性和交互性的测试，得出的结论是：在不牺牲交互性的前提下，CFS 很好地做到了公平性，而且 CFS 没有采用复杂的算法去区分一个进程是不是交互性的，所以 CFS 比 O(1)调度系统更有效率。

3.5 本章小结

本章内容的重点是通过内核源码的剖析，从代码实现的角度分析了完全公平调度 CFS 实现的基本原理，并对 CFS 相关的函数进行了详细的剖析，对较长的函数据其源码画出了流程图，借此来更加全面、细致的认识 CFS，对下边的研究做准备。

第四章　SMP 调度研究

4.1 SMP 调度基础

在第一章我们已经讲过，SMP 是指对称多处理器[34]，即系统中所有的 CPU 通过总线相连并共同使用内存和外设，彼此对称，不存在谁是主 CPU 谁是从 CPU。为了降低对内存访问的冲突，SMP 的硬件做了相应的改进，使得每个 CPU 都有自己的高速缓存，并有相应的硬件支持它们之间的通信。CFS 调度系统对 SMP 的支持得到了进一步提高，现在的调度系统能在保持效率的情况下进行多个 CPU 的负载平衡操作。通过在编译前配置内核时打开 CONFIG_SMP 选项，就可以让内核把源码中的关于 SMP 的代码一起编译，其中关于 SMP 的代码是被放在#ifdef CONFIG_SMP 跟#endif 之间。

因为 CFS 采用了模块化设计，所以所有的关于 SMP 的操作都被做成了函数接口，放在调度类结构体 sched_class 中，其中针对 CFS 的赋值如下所示：

```
static const struct sched_class fair_sched_class = {
#ifdef CONFIG_SMP
    .select_task_rq       = select_task_rq_fair, //负责选择进程所在的运行队列。
    .rq_online             = rq_online_fair,
    .rq_offline            = rq_offline_fair,
    .task_waking           = task_waking_fair,
#endif
};
```

接下来将主要探讨 CFS 是怎样与 SMP 调度系统进行配合工作的，为了使平衡算法获得最好的效率，首先要弄清楚系统中 CPU 的结构，让我们先来了解一下什么是调度域。

4.1.1 调度域分析

Linux 始终坚持采用 SMP 的架构，即内核对其中的每个 CPU 都没有偏袒。但不同的多处理器架构的机器也具有各自的风格，所以调度系统的实现要随着硬件的不同而有所区别。常见的多处理器有三种类型：

1. 标准多处理器系统：该系统中的所有 RAM 集被所有 CPU 共享使用，这种多处理器的体系架构是最普通的一种。
2. 超线程：一个支持超线程的微处理器能够立刻执行几个线程，它是由 Intel 提出的，支持超线程的一个物理 CPU 可以被看成多个不同的逻辑 CPU。
3. NUMA：把 RAM 和 CPU 以节点为单位分组，当 CPU 与本节点内的 RAM 进行交互时，几乎不存在竞争，所以反应非常的迅速，而与其他节点的 RAM 进行交互时，响应的时间就没有这么短了[35]。

目前高端的系统中常把这些基本的架构类型组合起来使用，如包含两个支持超线程的 CPU 可被看成四个逻辑 CPU，这样它就既是一个 NUMA 系统，又是 SMP 系统，每个物理 CPU 可以由多个核组成，各个物理核又可以通过 SMT 之类的技术实现虚拟 CPU 技术，在这样的系统中，逻辑处理器成了最基本的单元，任务的切换也是在逻辑 CPU 间进行的。而调度系统的任务不仅是让 CPU 资源得到尽可能的使用，而且还是保证每个 CPU 的资源得到均衡的使用，应该避免让有的 CPU 超负荷执行任务，而有的处理器却处于空闲状态，让进程在忙的处理器上排队，却还有空闲的 CPU。然而，要让每个 CPU 上的负载情况相差很小时，需要付出额外的开销，即使在系统启动时每个 CPU 上的负载情况是差不多的，然而有的任务执行的时间短有的执行时间长，并且新增加的任务不一定会分配到哪个 CPU 上执行，所以在一段时间后，CPU 上的负载情况就相差较多了，那么就要对这些情况进行查看，当发现严重失衡时，为了它们趋于平衡，就要把位于两个不同物理 CPU 上的进程进行负载的转移，然而这又会让 Cache 中的数据因过期而被覆盖掉，利用率不高，并且大量的负载平衡操作会给 CPU 带来负担，为此 Linux 2.6 设计并实现了调度域来解决这个问题，在 linux 中描述一个调度域的数据结构如表 4-1 所示。

表 4-1 sched_domain 结构体核心成员变量及注解表

成员变量	主要作用及含义
struct sched_domain *parent;	指向上一次调度域
struct sched_domain *child;	指向孩子调度域
struct sched_group *groups;	调度域所包含的调度组
unsigned long span[0];	记录调度域中的所有 CPU
enum sched_domain_level level;	该调度域所在的级别
unsigned long last_balance;	前一次平衡操作的时间戳
unsigned int balance_interval;	记录负载平衡间的时间间隔

结构体 sched_group 表示调度域中的每个组，结构体 sched_domain 中的 groups 字段指向该调度域的所有调度组构成的链表的首元素。若该调度域存在父调度域，其地址存放在 parent 字段，同理，孩子调度域的地址存放在 child 字段中，且系统中的全部物理 CPU 的 sched_domain 结构体的地址都存放到了每 CPU 变量的结构体 phys_domains 中。

调度域[36]就是把具有相同调度策略和属性 CPU 集合起来，即在一个系统当中可以采用不同的调度策略，也就是每个调度域可以选择使用一种自己的调度系统而不影响其他 CPU 的调度方式。调度域又根据超线程、对称多处理机等划分成多个级别，组成一个树状结构，形成分层的组织形式，并通过 parent 指针来组建这些层次结构，如图 4-1 所示：

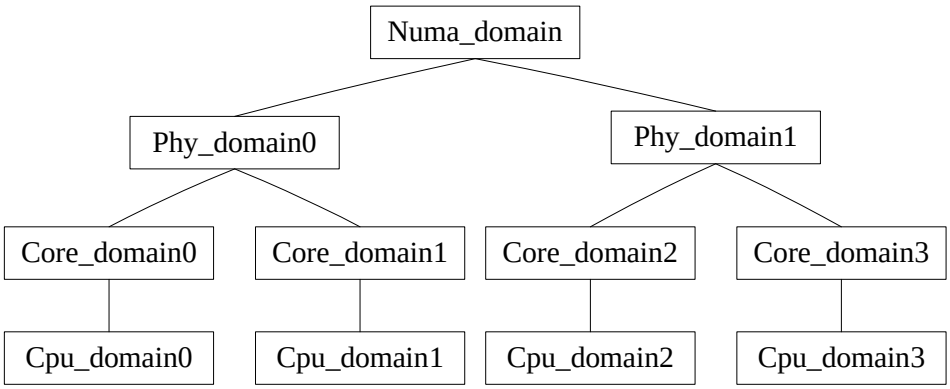


图 4-1 Scheduling Domains 原理图

其中系统中的每个 CPU 都有一个基本的调度域，即处在最下层的调度域，这些基本的调度域在 Linux 中可以用宏 cpu_sched_domain()跟 this_sched_domain()来访问。调度域的层次结构从基本的调度域开始组建，利用调度域结构体中的 parent 成员变量来连接到上一级的调度域。当某个调度域的 parent 指针等于 NULL 时，说明该调度域是最高级别的调度域了。

调度域中都包括一个或者几个处理器，并把它存放在结构体成员 span 的数组中。每个 sched_domain 结构体中的 span 变量一定是它的子调度域中 span 变量的超集合。处于最上层的调度域也就管理到了系统中全部的处理器。调度域结构体中的 span 表明了在哪些 CPU 间做负载的平衡。sched_domain 结构体中还有一个 groups 字段，即把调度域划分成互不交叉的调度组[37]，在 Linux 中用 struct sched_ groups 来描述调度组，它的核心成员如表 4-2 所示。

可见，通过它的 next 成员变量指针构建成循环链表。并且发生在调度域上的负载均衡操作是针对这些调度组之间的。并且把每个调度组中的负载量规定为此

表 4-2 sched_group 结构体核心成员变量及注解表

成员变量	主要作用及含义
struct sched_group *next;	用于构建循环链表
unsigned int cpu_power, cpu_power_orig;	CPU 的功率
unsigned long cpumask[0];	用于标记调度组所包含的 CPU

组中全部 CPU 上的负载，如果有些组之间的负载不平衡时，就在它们之间进行适当的任务转移操作。

4.1.2 同步机制及每 CPU 变量

当执行的结果取决于两个或者两个以上的交叉控制路径时，就可能出现竞争；倘若内核的控制路径对内核的某个数据结构操作时被挂起，则其他的控制路径就不能再对该被挂起的数据结构进行操作，否则会出现数据不一致性；当多个进程同时访问一个共享公共数据时，也会出现数据不一致性；要解决这些问题就要对这些数据进行同步。

同步机制是由计算机先驱 Dijkstra 设计推出的[38]，我们有时也会把同步机制叫做“锁”机制。在实际的应用中，同步的任务是维护临界值的互斥访问。在访问临界资源的时候，先去进行申请该临界资源的使用许可，如果得到申请许可，则锁定该临界资源并进行访问或修改；如果没有得到许可，则表明其他的某个进程正持有该临界资源的权限，此时该进程有两种选择：一是释放之前得到的 CPU 资源，并把自己阻塞挂起，等到所需的临界资源的许可能够重新可以去申请成功为止；二是不释放自己所使用的 CPU 资源，并不断地去查看能否成功申请到该临界资源的许可，也就是忙等。可见这两种处理的方式不同，所以适用的场合也有区别，各有所长：第一种自己释放了 CPU 资源，那么别的进程就可以去使用了，然而会使得发生任务的切换，给系统带来一定的压力和负担；另一种没有进行切换操作，但是却占用了 CPU 资源而没有做实质性的工作。所以当等待执行临界资源的时间比进行切换的时间短时，即等待的时间非常短时，应当使用第一种策略，否则让出 CPU 资源并挂起、等待。

在多处理器系统中，“忙等”机制通过自旋锁来实现的，当某个进程检测到自旋锁被其他进程占用时，就不停地“自旋”，即执行循环指令直至等待的锁打开。

还有一种同步机制叫做信号量，它本质上是一个的计数器，当进程试图访问某个数据结构前先要查看该结构对应的信号量。一个信号量可以被看做一个对象，

其组成包括一个整数、一个用于进程等待的链表及两个原子性方法：加减一操作。

系统中每个敏感的结构都一个信号量，初始值为某个整数值。当某个进程想要访问这个数据结构时，就在相应的信号量上执行减一操作，倘若结果不小于 0，则允许该进程访问数据结构，若不然则把该进程阻塞并插入到等待该信号量的链表中。当其他的进程对该信号执行加一后，唤醒阻塞在该信号量、位于阻塞队列中的第一个或几个进程。

目前最简洁、重要的同步手段是将逻辑上独立的内核变量声明成每 CPU 变量，它是 Linux 2.6 内核的一个新特性。它实际上是一些结构体的数组，各个处理器分别占用其中的一个元素，在处理器对各自的数组元素修改时不需要加锁，但是不允许操作别的处理器上的元素。这样每个 CPU 就使用各自专有的拷贝，为不同 CPU 间的并发提供了保护，对频繁更新的变量非常适用。

该方案最明显的缺点是不能对异步访问提供保护。如果一个处理器被另外一个抢占，则很可能会造成数据的不安全，所以在使用每 CPU 变量时，要保证关掉内核抢占。

根据研究显示，锁的“粒度”大小会对 SMP 调度系统的可扩展性有很多影响[39]。锁的“粒度”的大小是指该锁锁定范围的大小。假如系统中锁的粒度较大，在设计实现时算法会容易一些，也便于后期的维护，缺点是发生资源竞争的概率比较大，随着 SMP 系统的处理器的增多竞争的瓶颈会越来越明显，系统的并行度也随之大幅度下跌；假如系统中锁的粒度较小，系统中 CPU 间竞争就少一些，即使处理器数量增多竞争也不会随之大幅度上涨，如此一来可以方便的进行扩展，缺点是完成这样的算法非常难，并且不容易进行实施。因此要设计一个 SMP 调度系统，应统筹安排和规划，在保证系统的并行的同时，还要让其方便的进行扩展，并把算法设计不至于太难，源码的实施也控制在可接受的范围内，保证系统的可靠性。

4.1.3 局部性原理

程序的局部性原理[40]是指 CPU 存取数据或指令时，被访问的内容趋向于集中在一小块连续的区域，一般有三种不同的类型：时间的局部性，最近一段时间内访问过的数据、指令被再次访问几率很大；空间的局部性，现在和不久的将来访问的信息在地址空间里是相邻或相近的；顺序局部性：因为除了跳转外大多数的指令和数据的访问是按顺序进行的，指令执行的顺序性和数组存放的连续性最终

导致了顺序局部性。这个理论在计算机科学的发展过程中一直发挥着重要作用，不仅减少了开销还得到了很高的利用率和性能。

4.2 SMP 系统负载平衡剖析

SMP 系统中每个处理器上的负载是变化的，即每个 CPU 上的任务量不是固定不变的，而是随着时间在动态的改变，可能会有一些处理器超载而其他处理器的负荷却是闲置的情况。这无疑在很大程度上影响了系统的整体性能，用户也会得到的响应速度是难以接受的，为此我们设计把重载处理器上的任务挪到相对空闲或轻负载的处理器上，而这个过程称为负载平衡。

在 Linux 系统中，任务会被自动的分配到各个不同的 CPU 上去，并且尽量保持各个处理器间负载的均衡。那么，SMP 调度系统是怎么将多个 CPU 之间的负载均匀分配。作为一个成熟的负载均衡算法，关键在于做到：

1、查看系统中负载情况和作出调整的时机；

2、进行负载分配时采用的算法和准则。

4.2.1 何时查看

首先让我们来看一下什么时间去查看并调整系统中的负载分配。作为一个有明显时间标志的系统，我们首先能想到的最简单的方法就是每隔一段时间，就查看一次当前系统中各个处理器之间的负载是否平衡，如果不平衡就作出相应的调整，让其达到平衡。

或者在每生产一个进程或消亡一个进程时进行查看。然而一个进程从被创建到执行完成并退出，中间要经历不止一次的阻塞等待、睡眠挂起等，而发生这些状态转换时，对处理器的负载是有影响的，因此不能在这些时间点进行负载状况的查看，毕竟我们关注的核心是有哪些处理器上处理的进程多，哪些处理器空闲。

可见，这些做法都不能满足要求，Linux 2.6.36 中利用了时钟中断来触发查看负载的方法，通过调用函数 scheduler_tick()开始的，其中核心代码是：

```
#ifdef CONFIG_SMP
    rq->idle_at_tick = idle_cpu(cpu); //判断目前给定的 cpu 是否是空置的
    trigger_load_balance(rq, cpu);
#endif
```

又调用了函数 trigger_load_balance()，核心操作是：

if (time_after_eq(jiffies, rq->next_balance) && likely(!on_null_domain(cpu)))

raise_softirq(SCHED_SOFTIRQ);

可以看出 trigger_load_balance()激活了软终断 SCHED_SOFTIRQ，该软中断通过函数 sched_init()注册了下面的操作：

open_softirq(SCHED_SOFTIRQ, run_rebalance_domains);

即注册了中断处理函数 run_rebalance_domains()，它又调用 rebalance_domains()函数，它的流程如图 4-2 所示。

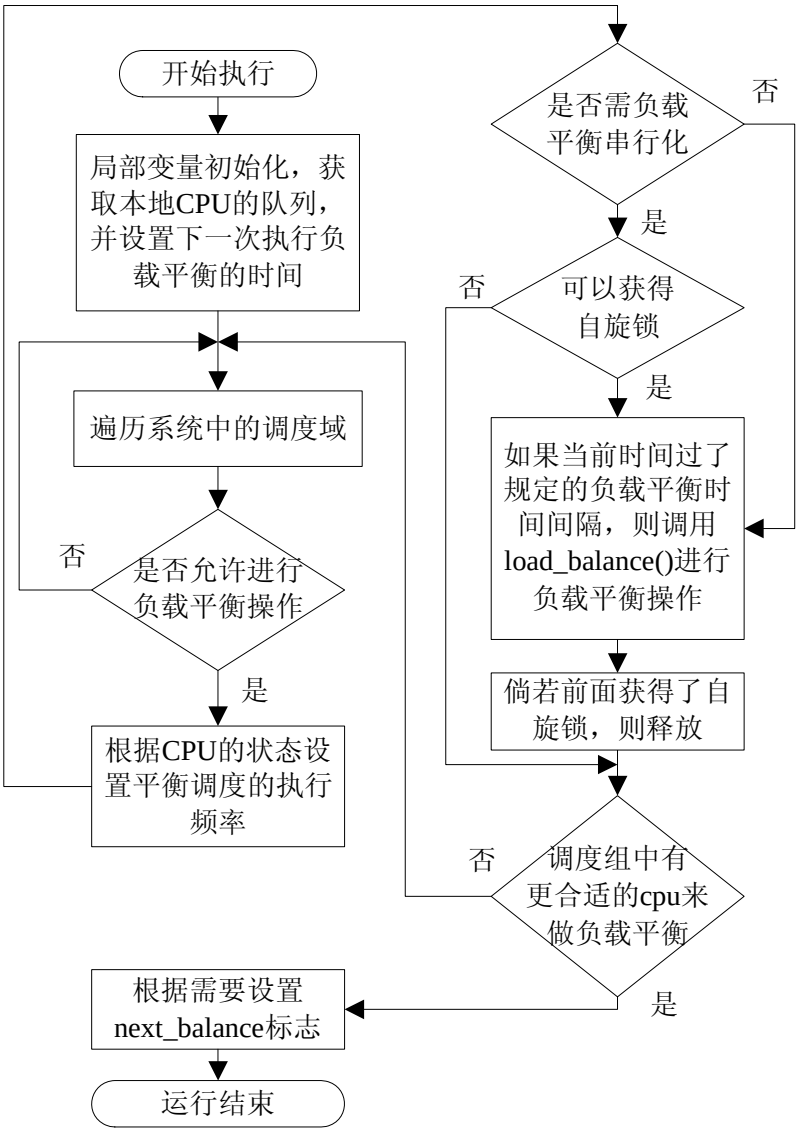


图 4-2 根据源码画出的 rebalance_domains()流程图

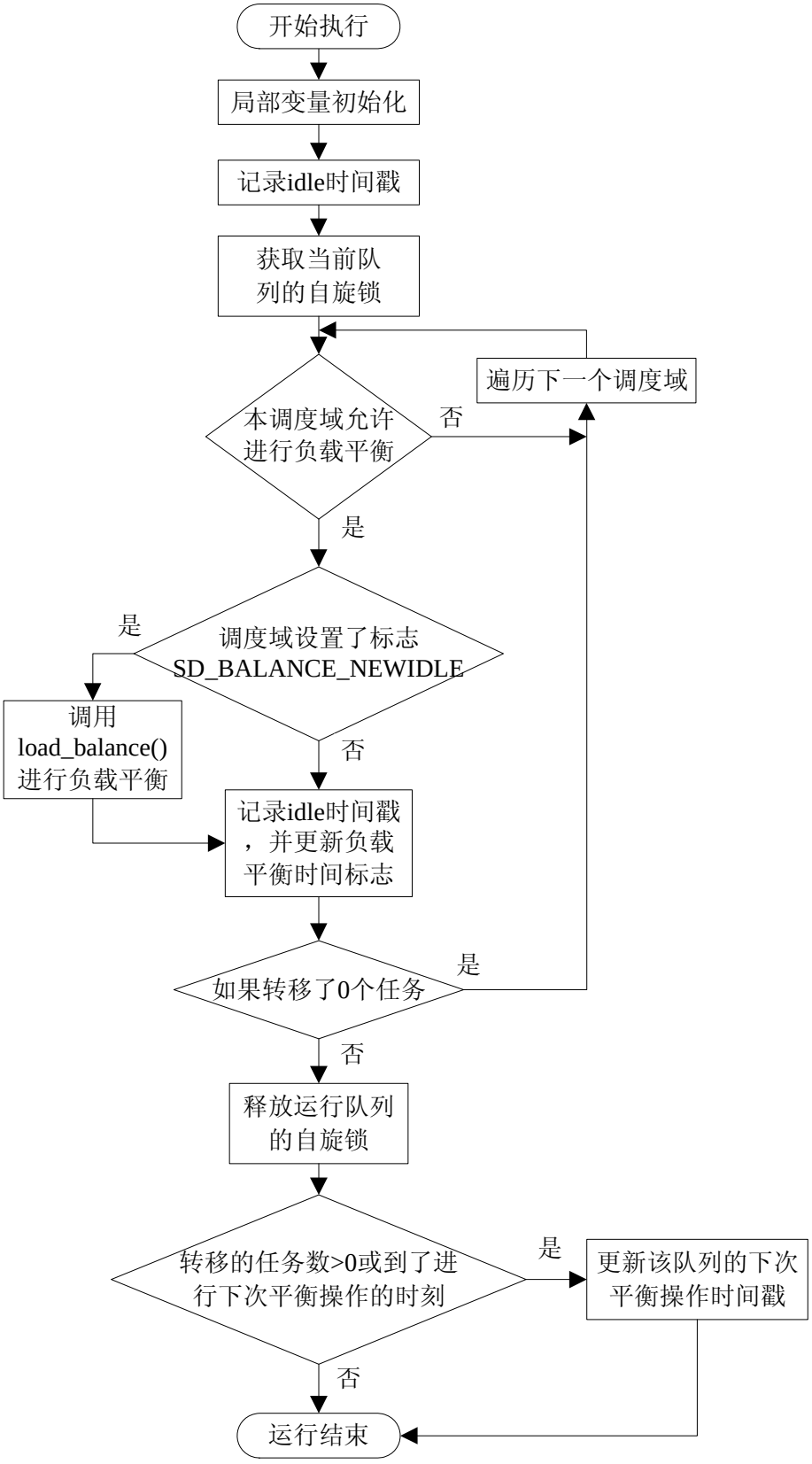


图 4-3 根据源码画出的 idle_balance()流程图

该函数检查每个调度域，并在必要时对负载平衡操作做准备。它利用for_each_domain()遍历系统中的每个调度域，并依据调度域的级别，由下往上的方式进行检查，这主要是为了提升处理 Cache 的命中率，因为子调度域一般运行在一个 CPU 或者通过超线程技术模拟出的多个处理器个上，这些处理器要访问的大多数据都共享在 L2 Cache 中，为了提升它的命中率而不至于使得 Cache 中的数据因长时间没有被访问而被新近从内存读进来的数据覆盖掉、造成失效，就非常的有必要让这些负载尽可能的运行在一个 CPU 上。

如果不平衡就执行负载平衡操作，进行任务的转移，并且在级别不一样的调度间转移进程的代价是不一样的，级别也高代价越大。rebalance_domains()依据给定调度域的 CPU 繁忙情况，判断是否要调用 load_balance()进行负载均衡，并且在给定 CPU 繁忙不同的情况下，load_balance()的执行频率会有所调整。

并且在函数 schedule()(流程如图 3-2 所示)中，如果当前的 CPU 状态是空闲的，它会直接调用函数 idle_balance()，它的流程图如图 4-3 所示。

可见，它调用函数 load_balance()时，传递的 CPU 状态参数是 CPU_NEWLY_IDLE，它与其他类型的 load_balance()的不同之处主要体现在：

1、它只有在 CPU 空闲时立即调用，不会对下次调用的时间间隔的进行更改，并且为接下来的时间中断进行的负载平衡设置一个小的时间阀值，以达到快速觉察不平衡状态。

2、在这种状态下，只进行少量的任务转移，能保障 CPU 上有任务在执行即可。

3、它通过把另外的 CPU 上“拉”任务至本地的队列中。

不过它们最终也都通过调用函数 load_balance()来完成负载平衡。

透过上面的分析可以看出 Linux kernel 利用了软中断，通过 CPU 时钟滴答来触发查看当前系统中负载的时机。但是查看负载或作出负载调整的频率是怎样的，是不是进行负载平衡的频率就等于调用 scheduler_tick()的频率，通过阅读函数 rebalance_domains()可以看出，调用的频率是根据 CPU 的繁忙情况而定的，所以频率是可变的。

所以，我们通过对调度系统的分析，最后归结出，在 Linux kernel 中有两条路径来查看当前系统中的负载情况，并决定是否进行负载平衡，如图 4-4 所示。

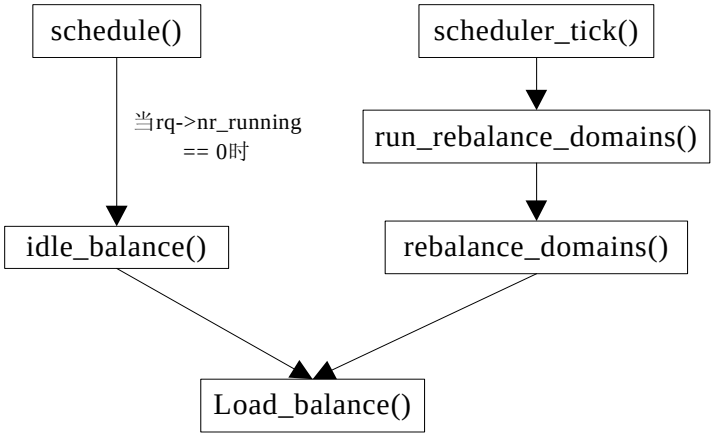


图 4-4 进行负载平衡的两条路径

在 Linux kernel 中，进行负载平衡的频率是在函数 rebalance_domains()进行的，该函数的大体流程是，首先声明一个长整形变量 interval 并初始化为 1，用它来记录时间。在遍历各个调度域的过程中，该函数会检查当前给定的 CPU 状态是否是 CPU_IDLE 状态，即是否处于空闲态，如果不是，则让 interval 乘以一个因子，即让 interval 增大数倍，如果系统的当前时间比上次进行平衡的时间加上 interval 还要大时，则调用 load_balance()函数。

通过这种方法，在当 idle 的标志位等于 SCHED_IDLE 时，即当前给定的 CPU 处理器的状态是空闲，interval 会比较小，那么 SMP 调度系统就会用较高的调用频繁(大概几个时钟滴答)来执行函数 load_balance()，如若不然，即当前给定的 CPU 的状态是忙碌的，那么 SMP 调度系统就会用较低的调用频繁(大概 0.1 秒左右)执行 load_balance()。

4.2.2 如何调整

接下来，开始讨论怎么进行负载的分配的问题。既然系统中调度域之间或着某个调度域内的各个组之间或者某个组内的 CPU 之间出现了负载不平衡的状况时，我们设想，如果把最忙碌的 CPU 上的任务转移一个到最清闲的 CPU 上，如果还不够平衡，那么就再转移一个，如此循环往复，直至达到均衡为止。

虽然看上去能够满足负载平衡的要求，但是还是先看一下 Linux kernel 中是怎么实现的。在上一节时，我们已经指出 Linux 的负载平衡是通过 load_balance()实现的，接下来就看看它的具体实现。

load_balance()函数负责查看是否调度域处于明显的不均衡状态，通过把最繁忙的组中的一些任务转移到本地的队列来减轻不平衡状况，load_balance()函数的

处理流程如图 4-5 所示。

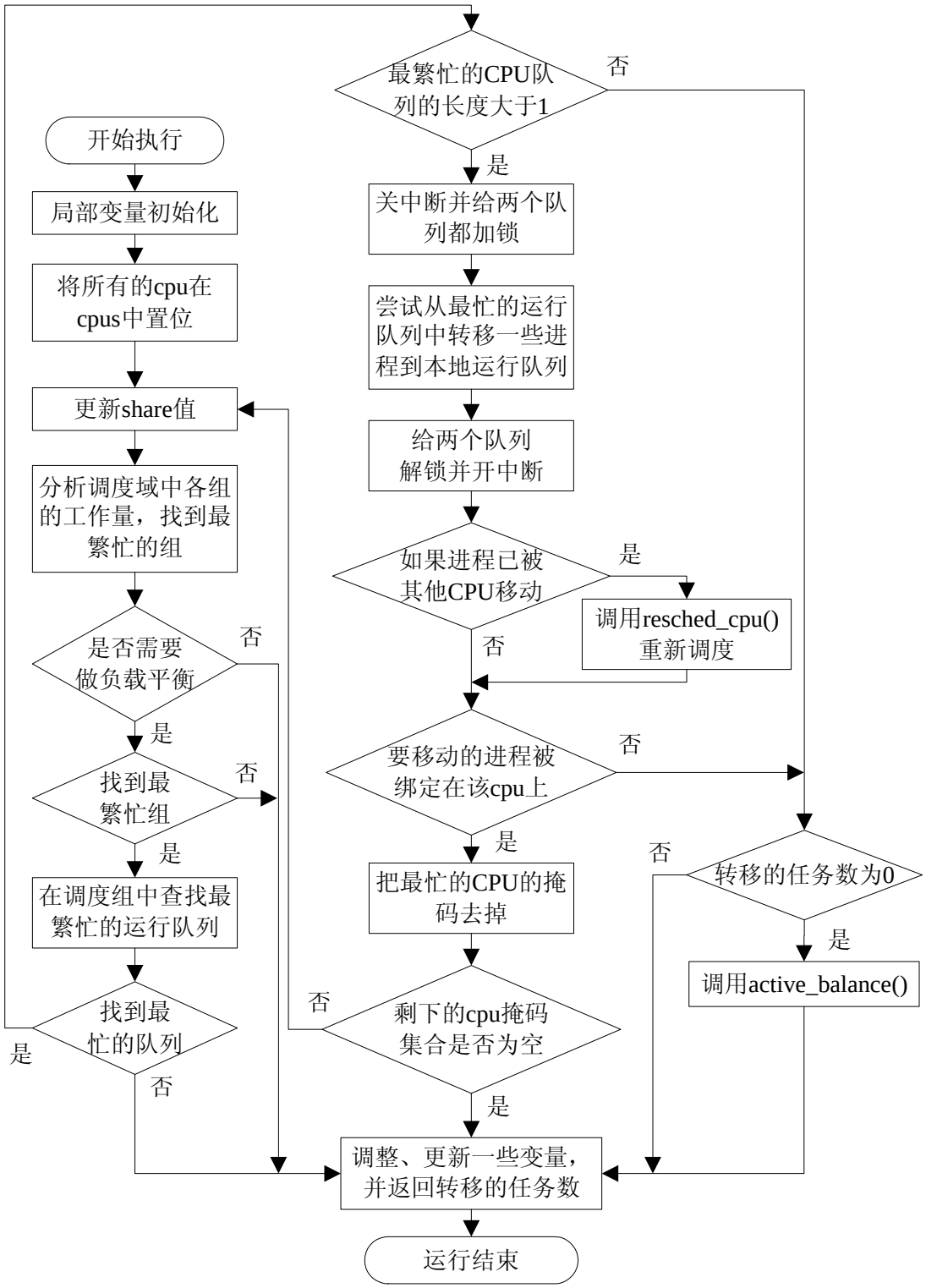


图 4-5 根据源码画出的 load_balance()流程图

不难看出，该函数的主要流程就是先在调度域中找到最忙的调度组，其中判

断的根据是该队列的负载的大小，是在函数 find_busiest_group()中进行的，在 Linux 中有一个数据结构 sb_lb_stats，它保存了一个调度域内的一些用来进行平衡操作的一些数据，如表 4-3 所示。

表 4-3 sd_lb_stats 结构体核心成员变量及注解表

成员变量	主要作用及含义
struct sched_group *busiest;	指向该域中最忙忙碌的组
struct sched_group *this;	该调度域中当前的那个调度组的指针
unsigned long total_load;	调度域中所有 CPU 上的总负载
unsigned long avg_load;	调度域中的平均负载
unsigned long this_load;	当前调度组的负载
unsigned long this_load_per_task;	当前调度组的平均负载
unsigned long this_nr_running;	当前调度组内运行队列中进程的总数
unsigned long max_load;	最忙的组的负载量
unsigned long busiest_load_per_task;	最忙的组中平均每个任务的负载量
unsigned long busiest_nr_running;	最忙的组中所有运行队列中进程的个数

其中，busiest 指向本调度域内最忙的组，而 this_*变量用于表示当前 CPU 的运行队列，决定选择调度域中最忙的组的参照标准是该组内所有 CPU 上负载(load)的和，找到组中找到忙的运行队列的参照标准是该 CPU 运行队列的长度，即负载，并且 load 值越大就表示越忙。在平衡的过程中，通过比较当前队列与以前记录的 busiest 的负载情况，及时更新这些变量，让 busiest 始终指向域内最忙的一组，以便于查找。

在比较负载大小的过程中，当发现当前运行的 CPU 所在的组中 busiest 为空时，或者当前正在运行的 CPU 队列就是最忙的时，或者当前 CPU 队列的负载不小于本组内的平均负载时，或者不平衡的额度不大时，都会返回 NULL 值，即组组之间不需要进行平衡。不然，就根据不平衡额度的大小做出选择，当最忙的组的负载小于该调度域的平均负载时，只需要进行小范围的负载平衡；当要转移的任务量小于每个进程的平均负载时，也进行小范围的负载平衡。通常转移的量不是很大，是防止某些处理器从忙碌状态变成 idle 状态。当一个调度域的总负载小于或者等于该组内进程的平均负载时，不需要进行负载平衡。当找到最忙的组后，再从最忙的组中找出最忙的 CPU，它是利用 find_busiest_queue()实现的。它遍历该组中的所有 CPU 队列，经过依次比较各个队列的负载，找到最忙的那个队列。

最后一步环节就是任务的转移了，代码层的实现是用函数 move_tasks()实现的，它负责把进程从源队列转移到本地队列。首先检查，如果该任务的负载重量的一半比要进行转移的任务量还要大，则跳过该任务，不转移它，接着利用函数 can_migrate_task()检查进程是不是能够被移动。当该进程正在运行，或者本地 CPU 不包含在进程描述符的 cpus_allowed 字段中，或者进程此时是高缓存命中率的时，直接跳过该进程，即不转移该进程。

当发现有进程能够被转移时，调用 pull_task()完成最后的出源端队列，入本地队列的操作，如果新被转移进来的任务具有更高的优先级，则置位调度的标志位 TIF_NEED_RESCHED，以便将来抢占。

从 Linux 2.6 开始，提出了 CPU 亲和力机制的概念[41]，用户可以在用户空间通过编写程序，指定一个进程在哪个或者哪几个 CPU(所用的系统是多核的)上运行，这样可以使得有亲近关系的进程能够运行在同一处理器上，进而能够高效的使用 Cache，提升 Cache 的利用率，加快程序执行的速度。然而这种方式是静态的，如果指定的 CPU 负载已经很忙，还给它增加负载时，将得不到很好的收效。所有，这个方式缺乏灵活性。

因为在不同 CPU 间转移负载时是有代价的，所以不应当忽视系统缓存的失效、能源的损失等因素，正是这些因素的会影响函数执行并最终影响调度的效率。到底选择哪些进程进行转移代价才能降到最小，经过分析可以看出 Linux 使用了如下方案：第一，利用运行队列的负载量（runqueue 的长度）大小为依据，决定需要进行平衡操作的调度组；第二，被迁移的进程没有正在运行，如果被转移的进程正在运行，那么发生迁移时还要发生进程切换，造成一次上下文切换，而且刚刚被读到 Cache 中的数据没来得及使用就可能被覆盖掉；第三，转移的目的 CPU 可以利用进程描述符的 cpus_allowed 字段进行标注，使进程转移时有一定的方向性；第三，如果被转移的进程在源端 CPU 上有很高的缓存使用率，则尽可能去避免这样的进程，间接提升了 Cache 利用率。

可以看出，这和我们在本节开头提出的方法是类似的，不过在 Linux 中的实现考虑的更加全面、细致。

4.2.3 概括 SMP 调度流程

通过对源码的剖析，概括出的流程图如图 4-6 所示。

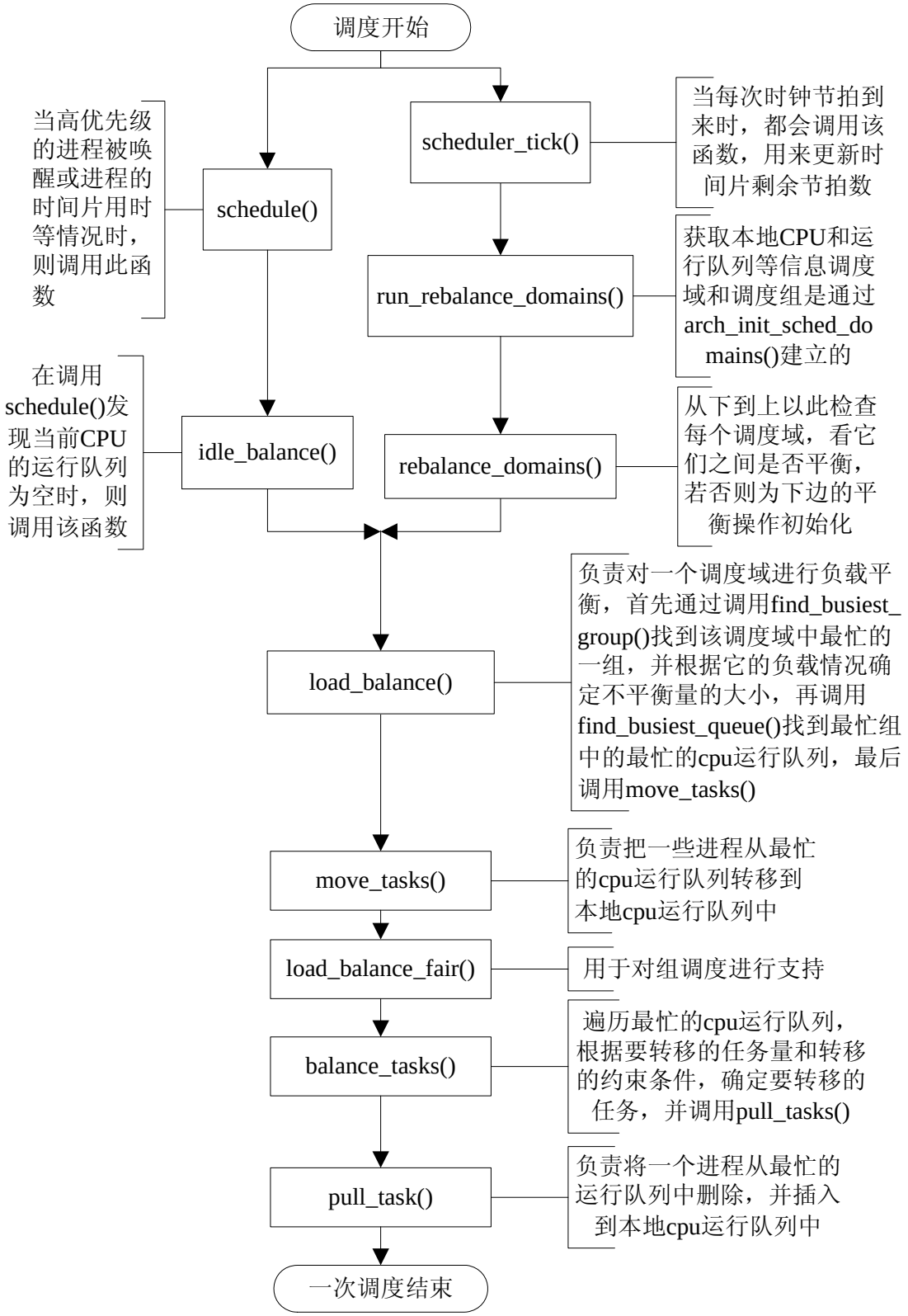


图 4-6 概括出的 SMP 调度处理流程图

4.3 SMP 调度模型改进

通过 4.2 节对 SMP 源码的分析，并深入分析了各种负载均衡调度策略，文献[10]综合考虑了各个调度要素提出了<E，T，L，S>调度模型，其中分别表示系统环境、任务属性、负载评价和调度策略，但通过分析和本文将就提出的优化方案发现，该模型的考虑和设计不够全面，

一个合适的调度模型，无论对调度系统的研究还是对它进行学习，都有很重要的指导意义，但是如果模型本身设计不全面，可能会对使用此模型的人造成误解，本文结合对 SMP 调度的剖析，对原有的模型进行了扩充和改进。

首先，处理器作为一个重要的因素没有被单独列出，而系统环境这个要素表述过于笼统。

其次，文献[48]的研究表明用资源利用率即把处理器利用率和 I/O 利用率进行综合考虑，会得到更好地性能，并且现在的调度的研究领域还提出了基于温度感知而进行调度[17]，可见任务的转移不能仅考虑任务的负载，而该调度模型的调度依据用 Load（负载评价）表示，针对的只是负载，并且它只被用来评价处理器的状态，而在进行任务的转移时还须确定转移哪些任务以及它们的去处。

最后，该模型也没有考虑进程跟处理单元的关联关系，即运行队列是怎么进行维护的，并且它对约束的条件概括的不全面。

因此本文基于以上三点，对 Linux 内核的负载均衡调度模型进行了改进，并用四维向量<P，T，M，S>表示。其中：

P(Processor)：指的是系统中的处理器；

T(Task)：指的是系统的任务集；

M(Migrate)：指的是系统进行转移的依据，即转移因子；

S(Strategy)：指的是系统的负载均衡策略。

4.3.1 Processor

多核系统的处理器要素主要体现在三个方面：PA，SD 和 SG。

处理器架构(Processor architecture)：它用来描述处理器的架构及拓扑结构，因为不同的架构对负载平衡算法的影响是不同的。

根据组成多核系统的处理器是否是完全对称的，可以把多处理器系统分为同构的和异构的。其中，同构多核系统多用通用的处理器组成，多个处理器执行相同的任务，而异构多核系统除有通用处理器作为控制外，还集成了 DSP、ASIC、

VLIW 等来处理某些特殊的应用。

调度域 SD(Scheduling Domain)：调度域把具有相同调度策略和属性 CPU 集合起来，即在一个系统当中可以采用不同的调度策略，也就是每个调度域可以选择使用一种自己的调度系统而不影响其他 CPU 的调度方式，即把系统中的 CPU 进行了第一次划分。

调度组 SG(Scheduling Group)：每个调度域又可以划分为多个真子集，即调度组，这是对 CPU 的第二次划分，负载平衡操作就是在这些调度组间进行的。

4.3.2 Task

用户提交的任务的不同主要体现在 TT(Task Type)、TP(Task Processor)和 TC(Task Constraints)三个方面。TT 表示任务的类型，Linux 系统把任务大体分为普通进程和实时进程两类，而且对待他们的策略也是不同的。并且用来被作为转移因子的要素都是通过对任务的熟悉进行提取得到的，如负载大小，I/O 利用率，内存使用率等。TC 表示任务的约束条件，结构特征及性能需求等。

TP 是用来描述进程和处理单元间的关联关系，运行就绪的进程一般都被放到 CPU 运行队列中，而在 Linux 中的 SMP 系统中有多个处理单元，那么运行队列是集中管理、在处理单元要执行进程时再分配，还是预先就对进程进行分配，即把所有的运行就绪的进程挂到各个 CPU 运行队列中。

通过下边关于锁的机制的讲解中，我们也会看到，为了提高并行执行的效率，应该降低锁的粒度，应当采用分散管理，其中 Linux 就是给每个处理单元都维护着一个运行队列。

在 Linux 中的 TT 表现在把进程分为实时进程和普通进程，对实时进程和普通进程是分在不同的模块中进行处理的，并且约束条件是实时进程要优于普通进程进行执行。

4.3.3 Migrate

一个多核系统要进行负载的转移，必须有所依据，如负载大小，CPU 的温度高低等，可以使用一个五维向量(I，F，T，S，D)来表示，即转移的指标(Migrate Index)，用于计算转移急切程度的转移函数(Migrate Function)，作出决策时需要达到的转移阈值(Migrate Threshold)，反映系统自身的转移状态(Migrate State)，用于调节转移状态的状态修正因子(Migrate State Correction Factor)。

1、转移指标

一个系统转移指标的选取对于分析系统的负载平衡极为重要。转移指标是对系统中某个转移要素进行量化后的结果。选择不同的转移指标会直接影响负载均衡的策略和效果。通过下面的章节的分析可知，重要的转移指标有：运行队列的长度、CPU 利用率、内存的使用量、I/O 使用率、系统的平均响应时间，CPU 的温度等。根据实际的情况，可以选择某一个或者某几个指标组合起来使用。

Linux 中使用 CPU 运行队列的长度(nr_running)和任务的负载加权值作为转移指标，负载的大小是由调度策略和优先级决定的。

在选择转移指标时应注意以下几个方面：

1) 指标容易获取，这有利于频繁测量以及时反映系统的负载和 CPU 运行情况；
2) 能真实的反映系统上的负载和 CPU 运行情况；
3) 当同时使用多个转移指标时，对它们的获取应相互独立而不彼此干扰。

其中 Linux 内核考虑到了获取难度的问题，最后采用了把运行队列的长度作为系统转移指标，即把运行队列中的进程的负载加权的总和，作为转移指标。

2、转移函数

设 R1，R2，R3 为实数集，则转移函数如下：

$$f = f(R1, R2, R3\ldots)$$

如果把运行队列的长度 Nl 和内存的使用量 Nm 作为转移指标的话，那么此系统转移函数的实现如下：

$$f = f(Nl, Nm)$$

Linux 中的平衡单位是组，即当一个组内的负载超出平衡时才会触发平衡操作，因此负载函数的定义不是依赖单个 CPU 运行队列的长度，而是整个调度组所有运行队列的 nr_running 之和与 weighted_load 之和，设 n 是组拥有的 CPU 数目，则转移函数为：

$$f(nr\_running, weighted\_load) = \{(sum\_nr\_running, sum\_weighted\_load) \mid sum\_nr\_running = \sum_{i=1}^{n} \text{rq->nr\_running}, \quad sum\_weighted\_load = \sum_{i=1}^{n} \text{weighted\_cpuload(i)}\}$$

3、转移阈值

转移的阈值可以采用一对数< a1 , a2 > 来表示，其中 a1，a2∈R，且 0 < a1 ≤ a2。通过此转移阀值与利用转移函数得出的结果比较，而得出当前系统所处的转移状态。其中 Linux 内核在查找最忙的组时用 avg_load 表示阀值 a1，用 group_capacity 来表示该调度组的计算能力即 a2，在查找最忙的 CPU 队列时用 max_load 表示 a2。

4、转移状态

转移状态一般是通过转移阀值划分的，通常 CPU 的转移状态可分为空闲(idle)，轻量(light)、适量(suitable)和过量(over)四种状态，所以转移状态的集合为{空闲，轻量，适量，过量}。

一个系统的转移状态，是通过转移函数计算的结果与转移阀值进行比较而得出的，该模型通过转移的阀值来对转移的状态进行了划分，其关系如下：

$$S = \begin{cases} \text{空闲}, & f(I) = 0 \\ \text{轻量}, & f(I) \in (0, a1] \\ \text{适量}, & f(I) \in (a1, a2] \\ \text{过量}, & f(I) \in (a2, \infty) \end{cases}$$

5、转移状态修正因子

转移状态修正因子是用来修正系统的转移阈值，它可以根据系统中转移状态的情况而进行动态的调整，若让 C ≠ 0，即实现一种自适应的转移平衡调度策略。

4.3.4 Strategy

均衡的策略可以通过一个二维向量(ST，SE，SC)来描述，其中 ST(Strategy Type)是指均衡策略的类型。均衡策略通常分为静态和动态两种，静态均衡策略通过现有的状态和之前的经验，预先把所以的任务合理地安排到各个处理器上，运行时不再对进程进行重新分配。该策略的优点是可以容易地实现最优负载均衡，但在实际环境中的负载情况是动态变化的且比较复杂，静态的进行处理不能很好的适应。

动态均衡策略是通过实时的收集、处理并分析系统的运行状态信息，在运行的过程中动态地将任务转移到合适的处理器上，完成不同处理间的负载平衡。该均衡通常可以及时地根据系统中任务的分布，来作出平衡的决策，并能获得好的性能，更加灵活有效。但是它的实现也复杂，收集、分析状态信息需要系统的开

销，Linux 内核就采用了此种方式。

SE(Strategy Element)指策略的要素，其中动态均衡策略的主要包括六点，

1) 以某种频率和方式检测系统运行过程中的实时状态；
2) 根据转移指标、转移函数、转移阀值、转移状态等判断当前的处理器状态；
3) 确定进行任务调度转移的源点；
4) 根据转移指标、转移函数、转移阀值、转移状态等分析运行队列中的任务的属性；
5) 确定要进行转移的任务；
6) 确定进行任务转移调度的目标点；

Linux 内核的负载平衡算法是基于调度组的。当某调度域中的某个调度组的负

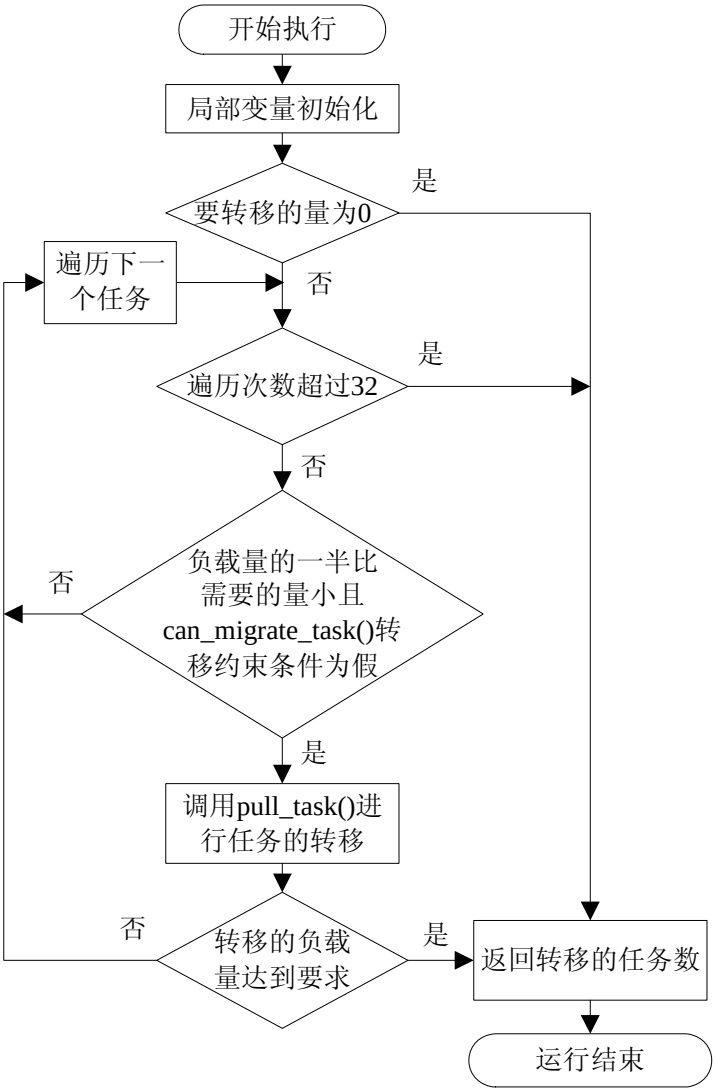


图 4-7 balance_tasks()流程图

载与该调度域的另一个调度组的负载量只差小于一定阀值时，才会进行负载的平衡。Linux 内核触发平衡操作的时机有：

1) 当 schedule()函数检测到某 CPU 进入 ilde 状态时，会调用 idle_balance()函数进行任务的转移，如 4.2.1 节分析。
2) 时钟中断会周期性地触发检查每个调度域的各个调度组是否处于负载平衡，如果不平衡就调用 load_balance()函数进行负载平衡，如 4.2.1 节分析。

SC(Strategy Constraints)是指平衡策略的约束。在 Linux 内核中，主要有三个方面，一是要考虑 CPU 的亲和力(affinity)，二是不转移正在运行的进程，三是不转移 Cache 命中率高的进程，该约束是通过函数 can_migrate_task()来实现，调用过程如流程图 4-7 所示。

4.3.5 SMP 系统的几个重要环节

作为计算机的四大子系统之一，调度系统的设计直接关系到系统的整体性能，因此在设计 SMP 调度系统时，要使得资源得到最大化的利用，系统的响应时间得到缩短，要把多处理器系统的性能得到最大化发挥和利用。

SMP 服务器的最显著的特点就是共享，其共享的范围包括了几乎系统中的所有主要资源，如 CPU、内存、总线、I/O 等等。而恰恰因为这些特征，造成了 SMP 服务器的扩展方面的缺陷：因为多个设备共享，SMP 性能扩张的瓶颈就限制在了共享的环节上，如 2.2 节中图 2-2 所示，假如每个 CPU 都要访问内存，内存就会因为竞争而成为瓶颈;又由于所有的 CPU 都是通过总线来访问内存或外部设备的，那么这些 CPU 也会竞争使用总线。不但如此，服务器系统越大，处理器越多，与内存进行交互时竞争总线就会恶化，内存访问的冲突也很不乐观，所以会造成资源利用率不高，CPU 得不到充分发挥。

到底如何才能从软件方面提升 SMP 系统的性能，通过熟悉了调度系统和分析了前人的研究基础上[42]，总结出的观点是：进行负载平衡比不进行负载平衡的性能要好，用做判断是否要进行负载平衡操作的状态信息得是现在的且不能太多。因此，SMP 调度系统的改进可以从负载平衡算法着手，即设计出要对哪些任务进行负载平衡，什么时间执行，转移多少，从哪个 CPU 转移到哪个 CPU 上。

另一方面，通过 4.1 节的分析我们看到，调度系统的设计跟锁、每 CPU 变量及程序的局部性原理都有密切的关系。

4.3.5.1 锁机制

在 4.1 节已经提到，SMP 系统中可能有多个任务并行运行，如果这么任务之间存在一些共享的资源或数据，那么对它们的访问就需要加锁进行同步。但是如果用粒度大的锁去保护资源，就会使得该锁可能成为竞争的对象，并随着竞争该锁的任务数的增多而使并行的瓶颈更加严重。因此要尽量采用小粒度的锁，提高 SMP 系统的并行能力。

另外一方面，如果多个进程有共享资源，并且它们位于同一个 CPU 上，当其中进程 X 访问该资源时，必须先去申请保护该资源的锁，倘若不能成功时则阻塞等待，说明该资源正被另外一个进程 Y 使用，当 Y 用后之后进程 X 接着执行，因为当 Y 执行时，缓存中已经存放了有关于共享资源的数据，这样 X 就可以直接使用，使得高速缓存中的数据可以多次利用，即提升了高速缓存的利用率，但是如果这两个进程不在同一个处理器上，当 Y 运行结束后，高速缓存的数据很可能过期而失效，白白浪费掉。

因此要尽量使用小粒度的锁，并让征用同一种锁的任务安排在同一个处理器中。

4.3.5.2 总线饱和程度

目前，对称多处理机的使用是非常普遍的，因为它既可以以很高的计算性能作为单独的系统使用，也可以作为构建计算机集群的组件，而且在花同样的价格时，SMP 系统所获得的性能要多于其他体系结构的系统。

然而，SMP 系统对资源的访问是共享的，各个 CPU、内存、外部设备都是通过总线连接在一起的，但连接处理器到内存的总线带宽是有限的，且不容易进行扩展，当多个处理器访问内存或外部设备时，都要使用总线，进而就造成了访问的冲突。

在文献[43]中，作者通过实验数据表明，总线带宽消耗型的应用程序可以使总线带宽的饱和度减少到 3 折，更糟糕的是一些程序的运行速度会变慢 2%到 55%。基于这样的实验发现，该文提出了两种基于总线带宽需求的调度策略 QWG 和 LQG。

LQG(Latest Quantum Gang) 和 QWG (Quanta Window Gang)两种调度策略很像一个伙伴调度策略，LQG 在每个最近一次的调度量结束之后，利用程序提供的信息为正在运行的任务更新总线的交易率，而最新的总线交易率是利用最近由应用程序引起的总线交易次数除以调度量。其中可用总线交易率(ABTR)用来表示在某次调度过程中可供分配的总线带宽，而系统总线交易率(SBTR)是一个常数，表示

该系统最大的交易率。

通过计算每次调度后的可用总线的交易率，进而算出可用总线交易率与未分配的处理器的比值：

$$ABTR_{/\text{proc}} = \frac{ABTR}{Unallocated\ \ processors}$$

只要有处理器是可用的，对于每个要执行的任务来说，匹配度是可以计算的：

$$\text{Fit}ness = \frac{1000}{1+|\ ABTR_{/\ proc} * Application\ Threads - BTR_{lastest}\ |}$$

当 $BTR_{lastest}$ 越接近 $ABTR_{/\ proc} * Application\ Threads$ 时，任务就越适合进行调度，即利用最新的量子总线交易率作出决策。而 QWG 不同的是它不使用最新的量子作依据，而是使用过去的一个量子窗口内的总线交易率。

通过实施在普通用户级别的 CPU 管理器上，实验结果显示，这两个策略能够比标准的 Linux 调度系统在性能上平均提升 26%，并且 QWG 策略比 LQG 策略更加的稳定，即使在边缘的应用上 QWG 也有很好的表现，而 LQG 显得过于敏感。

4.3.5.3 目的 CPU 的选择

多处理器系统通常有两种维护可运行队列的方式，一种是整个系统内只维护一个运行队列，另一种是把系统中的任务预先分配给各个 CPU，让每个 CPU 独自维护自己的运行队列。

在单处理器和简单的 SMP 系统中，通常只维护一个可运行进程队列，这种设计的好处是复杂度和实现难度都很小，并且也便于维护，因为有进程在运行队列中，就不会出现 CPU 空闲的情况，而且只有一个队列，不会进行负载平衡操作。但随着系统中 CPU 数量的增多，这种方式的缺点就变得不可忍受。因为所以处理器共享运行队列，在每个 CPU 操作该运行队列时，为了数据安全必须进行同步，这就需要给运行队列加锁，而该锁会锁住该 CPU 之外的所以处理器，也就使得锁的粒度过大，严重制约了进程的并行执行，成为了限制 SMP 系统性能的瓶颈。

不但如此，该方式还忽视了运行时的局部性理论。队列中进程的第一次执行和下一次执行可能距离的时间很长，队列中的任务越多可能时间越长，这就会使 Cache 中的数据过期，并且第一次执行与以后的执行不一定是在同一个 CPU 上进行的，这就把 Cache 中的数据直接浪费了，高速缓存的命中率也就大打折扣，而这对系统的整体性能而言，是不可忽视的[44]。

如果把运行队列预先分组，让每个处理器维护各自的队列，而不需要考虑其他队列中的进程，于此以来就缩小了锁的范围，避免了不同 CPU 间对运行队列的竞争，即保证了程序支持的并行度。而且 CPU 之间更加对立，不管 SMP 系统中 CPU 的数目有多少个，都不会造成激烈的竞争。由于每个任务都在固定的处理器上运行，很好的体现了局部性理论。

但是这种方式也不是一劳永逸的，因为有的任务执行的快，有的任务执行的慢，在以后的运行过程中难免会出现运行队列不一样长的情况，即造成各个 CPU 的负载不平衡，这就需要我们设计一套相应的负载平衡操作，以避免出现负载的严重不平衡。

整体来说，把任务预先分组的方式对 SMP 系统而言是利大于弊的，它即体现了程序的局部性原理，又提高了进程执行的并行度，虽然要考虑平衡问题，但是整体来说还是不错的选择。现在的 Linux 也是这么实现的，这也从实践的角度证明了我们的分析是正确的。

此外，在分配哪些任务到哪些 CPU 上执行，也是有区别的，在上一节中也提到，有些任务直接有共享资源，它们会进行频繁的核间通信，我们对此了做了分析，让它们分配到一个处理器上，会使得缓存的利用率高，但是如果如此进行安排，很可能会造成负载不和谐的状况。所以我们考虑让 CPU 也进行分组，也就是类似于调度域的概念，让联系紧密的任务分配到同一个调度域的同一个组中，不仅如此，因为使用了超线程技术，一个 CPU 可以模拟出 n 个逻辑的 CPU，而这些逻辑的处理器之间共享高速缓存等一些资源，我们把联系紧密的任务分配到这些处理器上，就不会造成严重的缓存失效问题[45]。

4.3.5.4 转移指标的选取

我们知道，当系统内一个域中某组的负载比该同域内另外某组的负载量多很多时时，就要进行负载平衡，把高负载 CPU 上的一部分进程转移到另外一个，而应该选择转移哪个或者哪些进程进行转移，应该选择什么样的策略作为依据做出决定，我们需要去分析。

在 4.2 节中已经对 Linux 中的 SMP 进行了分析，它采取的方案选择那些还未在 CPU 上运行的进程进行转移，并且可以通过设置 task_struct 中的的 cpu_allowed 变量来标记要转移到哪些 CPU 上，这样就可以降低转移的代价，但是还可以进一步缩减。假如我转移的进程的生命周期很短，转移该进程的时间与该进程的生命周期长度相当时，我们说转移这一类的进程是不合算的，因为直接执行该进程所

花的时候为 t，而把它先进行转移，再让它执行完毕所花的时间就是 2t，这就浪费了一半的 CPU 时间。因此，在选择时要选择生命周期较长的进程进行调度。

文献[46]通过利用进程的生命周期的长度，来进行负载平衡，它提出转移“老“的进程效果比较好，它利用大量实验，表明转移生命周期较长的进程能够弥补转移带来的消耗，并提出了转移代价的公式：

$$p = c + \frac{s}{b}$$

其中 c 是指不可避免的转移花费，s 指任务内存区的多少，b 指传送的宽度。因此，只要进程 a 还要运行的时间 t(a) > p，那么转移该进程的合适的。它又通过结合移进和移出任务的 CPU 上的进程数目 x 和 y，归纳出了转移进程的最小运行间是：

$$t(mi) = \frac{c + \frac{s}{b}}{x - y}$$

因此，只要一个进程剩余的运行时间大于 t(mi)，那么该进程就合适作为被转移的对象。

它还依据统计学的思想给某任务已使用的处理器时间 L 超过 1 秒时的生命长度(t)分布形式：

$$f\{t > L\} = L^n$$

n 近似为-1，并根据系统的不同而在-0.8 至-1.3 间波动，而当进程已使用的 CPU 时间 L 小于 1 秒时，此进程还能运行 L 的几率就大于 1/2，即多数情况下剩余生命周期会大于 L。更进一步，L 大于 p 时，即 $L > c + s/b$ 时，该进程就适合转移。

再者，有些进程的自身的地址空间较大，占用的资源较大，那么在转移时花费的代价自然就大，而且在访问时，涉及到的内存页面多，会增加访问内存的冲突，所以尽量不要选择这样的进程。通过阅读发现，基于公平的任务调度负载平衡的算法[47]，能够在一定程度上改善系统的现状。它通过对比队列中的任务的多少、调用的快慢、进程切换的发生次数、CPU 的使用情况、闲置内存的多少及负载的平均值等要素及它们的与或组合，得出最好的指标是运行队列的长度，因此现有的系统的负载指标大多使用运行队列的长度。但文献[48]的研究指出，把资源利用率作为参照标准，即处理器的利用率和 I/O 设备的利用率，会更加的鲜明，并且这种方式更加便于资源的利用率以及系统响应速度的提升，不仅如此，该文献还通过实验表明同时考虑了资源利用率跟运行队列长度时，会得到更好的性能。

最后，从进程间的关联程度来考虑。我们知道，有些进程之间有共享变量、共享存储区，进程之间要进行通信等，使得进程之间藕断丝连，倘若这样的进程放到不同的 CPU 上执行，会造成因访问共享资源而发生冲突，并且使得相同的数据存放到了多个 CPU 的 Cache 中，不利于提升高速缓存的命中率，还可能会使缓存的数据频频过期而被覆盖掉。这对于同一个进程的多个不同线程的情况比较普遍，所以尽量让这些任务放到给定的处理器上执行，对此，Linux 提出了 CPU 亲和力的技术，通过编程实现可将一个进程在指定的 CPU 上执行[49]。

文献[50]利用 CPU 利用率和内存利用率作为负载因子，对调度算法进行了重新设计。

4.4 基于唤醒信号的优化方案

通过 4.2 节对进程调度的研究，了解了调度的流程和基本原理，Linux 内核提供了支持 SMP、SMT 的编译选项，用户可根据需求对其打开或关闭。Linux 会在系统启动时得到处理器的拓扑结构，并根据此结构进行调度域的划分，此外，Linux 内核为每一个处理单元维护一个运行队列，以此来进一步降低调度系统中锁的粒度。

从实际的应用反馈发现，Linux 调度在 SMP 方面的性能表现还是很优异的，它的独立的运行队列设计，既能方便的进行扩展，同时也充分利用了各个处理单元的高速缓存，但是对任务分开处理也有不好的方面：调度系统在需要时会在处理器间进行负载平衡，即把任务从一个运行队列转移到另外一个，而负载平衡就意味着一定程度的 Cache 失效(本文的研究是基于系统中的处理器是有 Cache 的)。

Cache 就是根据局部性原理而设计的。Cache，即高速缓存存储器，其作用就是为了缓解 CPU 和内存在存取速度上越来越大的差距，不让 CPU 白白的在等内存的反应。Cache 的容量不大，通常只有主存储器的几百分之一，但存取速度远远高于内存，甚至接近 CPU 的存取速度。当 CPU 要存取数据或指令时，根据程序局部性原理，正在使用的主存储器某一单元邻近的那些单元将被用到的可能性很大。因而，当中央处理器存取主存储器某一单元时，计算机硬件就自动地将包括该单元在内的那一组单元内容调入高速缓冲存储器，中央处理器即将读取的主存储器单元很可能就在刚刚调入到高速缓冲存储器的那一组单元内。

对高速缓存利用，内核采取了一些解决措施，如优先在同等级别的调度域间进行负载平衡、不对高 Cache 命中率的任务进行转移、不对正在运行的进程进行

转移等，在很大程度上避免了高速缓存的失效，但通过对内核的剖析可以看出，当多个进程间有共享的系统资源，但它们运行于不同的处理器上，那么每当一个进程第一次访问该共享资源时，都会造成一定程度的高速缓存失效，但是如果让这些进程在同一处理器上运行，那么在访问共享资源时高速缓存命中率就会在一定程度上得到提高。因此本文将以尽量减少 CPU 高速缓存失效为目标，提出一个优化方案并进行实践。

4.4.1 SMP 算法设计

针对上面的分析，本小节将结合 4.3 节提出的调度模型，进行的算法设计。通过分析，我们了解了影响 SMP 系统进程调度的主要因素，并且主要的焦点落在任务的分布和负载平衡算法上，为了提高系统的资源利用率和整体性能，充分利用高速缓存，在进行任务的转移时，要考多种要素，而标准的 Linux 内核中的 SMP 在进行任务的转移时，没有对运行队列中的任务属性的不同进行区别的对待，所以要想进一步提升高速缓存的利用率，主要是通过对转移的任务进行选择，让共享资源多的进程尽量分配到同一个处理器上执行，通过本文提出的调度模型可知，要在选择转移的对象时，添加一个转移因子，即设计一个针对提升 Cache 利用率的转移因子。

1、转移指标

要想让共享资源多的进程分配到一起执行，即特征是共享资源，而我们知道为了保护共享资源的安全性，在访问这些资源时要先取得保护该资源的锁，所以可以通过利用该锁来把多个耦合度高的进程关联起来，所以转移指标设置为保护进程间共享资源的锁。

2、转移函数

该函数的功能主要是用来确定任务转移的目的端，既然想让共享同一把锁的进程运行在同一个处理单元上，那么需要首先知道第一个持有锁的进程在哪个处理器上执行，并对征用该锁的进程进行标记赋值，记录下要转移到的处理器，即函数为：

$$f(\text{Task}_{\text{持有某指定锁}})=n\text{（最佳的处理器编号）。}$$

3、转移阀值

结合原有调度系统的基础上，对某些进程和 CPU 进行最佳匹配，当要进行任务的转移时，利用各个 CPU 的编号作为阀值，当转移函数得到的结果等于某个阀值时，就把该任务优先转移到对应的 CPU 上执行。

经过这样的优化以后，进程间共享资源多的进程会被尽量安排到同一个处理上执行，这会带来如下的好处，首先，处理器的 Cache 会得到重复利用，从而提高 Cache 命中率；其次，竞争同一把锁的进程运行在同一个处理器上，会加快锁的获取速度，从而提高了执行的速度；再次，进程间进行通信时可以直接通过读取 Cache 来完成，从而在一定程度上降低了总线了的占用率。因此，让前一个进程释放了锁唤醒的同时通知竞争该锁的进程，让它们尽量转移到唤醒进程运行的处理器上执行，故本文提出了一种基于唤醒信号反馈的优化方案，来对 SMP 调度系统进行优化。

为此，我们需要一种能被内核支持的用户态锁，所幸的是现在的 Linux 支持一种快速的用户态锁定机制 futex[51]，它是一种轻量级的用户态同步方面。为了实现本文的优化，先介绍一下 futex 锁。

4.4.2 FUTEX 锁机制

在传统的 Unix 中，进程间的同步机制是通过对内核对象的操作来实现的，并且这些操作对同步它的进程是有影响的。但研究表明，存在大量的同步是没有竞争关系，不如某个进程 c 从进入互斥区至退出该互斥区期间，通常没有其他的进程去竞争，但是进程 c 在陷入内核后和在退出时都要检查有没有竞争，这就造成了没有必要的系统开销，Futex(fast userspace mutex 的缩写)的诞生就是为此而来的[52]。futex 是用户态跟内核态的混合同步方法，是一种快速的用户态互斥锁，通过用户态的锁跟信号量组成的，它可以被用作不同进程间的共享内存，不论进程还是线程都能使用，因为一个进程的多个线程的虚拟空间是共享的，futex 变量可以通过虚拟地址来唯一识别，而进程则通过 mmap()申请一块共享内存区间。

不但如此，futex 的操作是在用户态进行，只有在必要的情况下才调用系统调用，陷入内核，这样就减少了不必要的开销。

在使用时，首先需要同步的进程声明一个 futex 变量并放在它们的共享内存中，当进程要进入或退出互斥区时，首先需要对 futex 执行原子性的减一操作，此时再查看 futex 的值：

1. 如果 futex 变成 0，则说明 futex 锁可以使用，竞争不存在，申请占用该锁的进程得到该锁，而不必陷入内核；
2. 如果 futex 变成了一个负数，则说明此时 futex 锁不可用，有竞争存在，该进程要通过系统调用陷入内核，并把此进程阻塞在该锁相应的等待队列

中；

相应的在释放该锁时，就对变量 futex 执行原子性的加 1 操作，再检查变量 futex 的值：

1. 如果此时 futex 大于 0，则说明这段时间内没有其他进程来征用该锁，即不存在竞争，该进程正常执行；
2. 如果此时 futex 不大于 0，则说明这段时间内有进程来征用该锁，即存在竞争，则需要调用系统调用 sys_futex 陷入内核，去唤醒一个或几个等待使用该锁的进程。

4.4.3 基于反馈的优化方案

所谓的反馈是指利用 CPU 采集负载信息，再根据这些信息来影响动态负载平衡算法的一种机制[53]，我们通过 4.2 节的分析可知，Linux 的调度系统在分配任务时没有考虑到各个任务之间的联系，如果将两个需要经常进行通信的进程分配到不同的 CPU 上，就在一定程度上降低了高速缓冲区的利用率，而且在进行负载平衡进行任务转移时，也需要考虑这方面的影响。

为了让持有相同锁的进程运行在同一个处理器上，当第一个获得该锁并使用完之后，在唤醒的同时也把持有该锁的进程呼唤到它的处理器上来，所以本文提出的利用唤醒信号反馈的调度优化方案的大体模型如图 4-8 所示，调度系统在第一次执行时按照原来的调度系统流程进行，但在调度完成后，开始采集一些信息，反馈模块对这些信息进行处理后，反馈给调度系统，进而影响调度系统的执行流程，从而达到优化的目的。

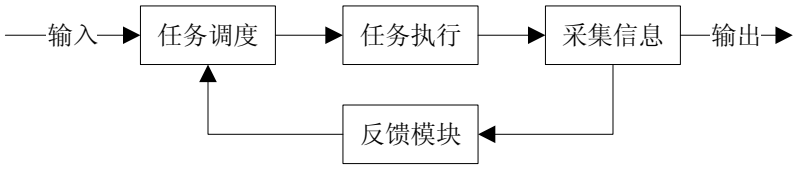


图 4-8 反馈的模型图

4.4.4 优化算法实施及流程

设想，如果有一个进程 a 持有某种锁 S 被分配到 CPU 0 上执行，现在又有一个使用该锁的进程 b 要执行，为了不让 CPU 间因锁 S 而竞争，并充分利用高速缓存，只要 CPU 0 上的负载没有严重超标，分配到 CPU 0 上最合适。

在多个进程征用同一个 futex 锁时，它们之间存在着联系，要对该 futex 锁进

行竞争，在运行的过程中要使用共同的数据对象，为了充分利用这些在高速缓存中的数据，所以当第一个征用该锁的进程运行完成后，在释放锁的同时获取它所在的 CPU 编号，当存在等待此锁的队列时，就让队列中的任务记录当前 CPU 的编号，以便于将来对把该任务转移到此处理器上去执行。

因为 SMP 系统中的 CPU 是维护在一个数组中，所以只要知道数组的下标就能唯一确定该 CPU。而要标示一个进程最合适运行的 CPU，需要添加一个整型的属性字段 cpuindex。因为它是标示进程的属性，所以将它添加到进程控制块 task_struct 中，它的数据数据结构如表 3-2 所示。

在创建进程时，就把 cpuindex 属性初始化成-1。当系统运行到进程 a，并发现它需要申请使用完 futex 锁后，当释放该锁时，如果发现有竞争存在，就通过函数接口 task_cpu()来获得当前的 CPU 编号（数组的下标值）并记录在它的 cpuindex 中，再把它作为参数传递给唤醒函数，让它在唤醒进程的同时把 CPU 的编号也一起传过去，并把被唤醒的进程的 cpuindex 字段设置为传递的 CPU 编号，释放 futex 锁并唤醒等待该锁的进程。

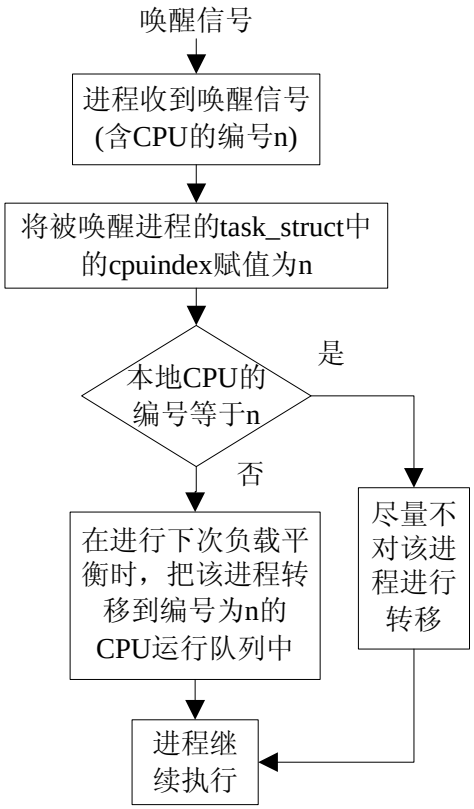


图 4-9 被唤醒进程所做的处理

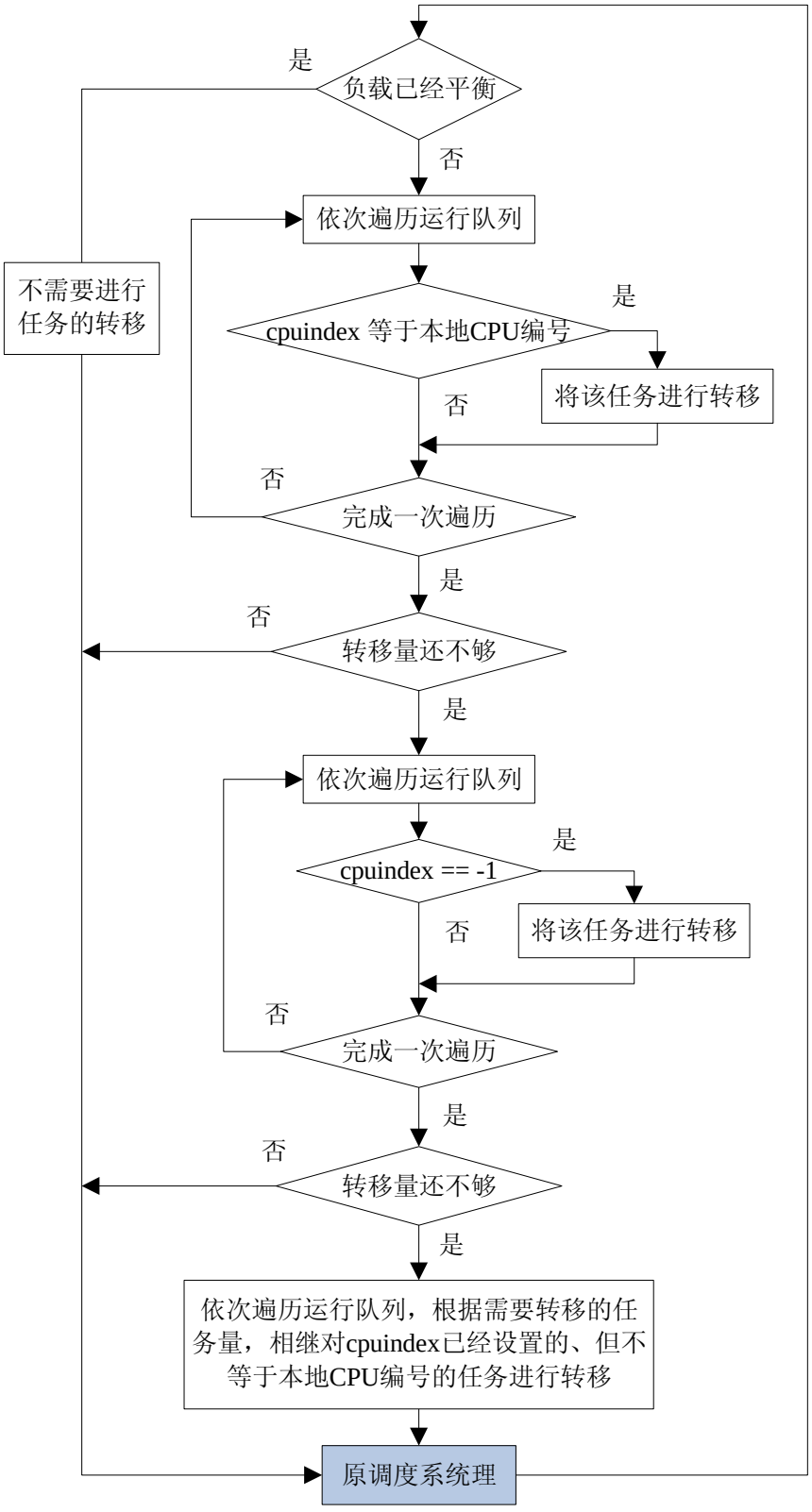


图 4-10 在转移时对任务处理的流程简图

此时，如果被唤醒的进程所在的 CPU 等于它的 task_struct 中的 cpuindex 变量，则不需要进行转移操作；否则，就尽量把被唤醒的进程转移到编号为 cpuindex 的 CPU 上去。对被唤醒进程所做的处理的流程如图 4-9 所示，但如果目的 CPU 的转移状态决定了它不能作为转移的目的处理器，则转移失败，若被唤醒的进程没有转移成功，则按照标准内核的操作流程执行，暂不起优化的作用。

利用此优化方案后，当调度系统查看当前的负载状况时，如果发现有负载不平衡时，会通过调用 load_balance()进行负载平衡。首先它会找到调度域中最忙的一个组，再从最忙的组中找到负载最忙的处理器，并选择其中的一些负载，把它们转移到本地处理器上来。在选择被转移的负载时，在遍历的过程中，会检查该任务的 cpuindex，看它是否为-1，可能遇到的情况有下边几种方式：

1. 若某任务的 cpuindex 不等于-1，而与当前所在的 CPU 的编号不相等，且与当前 CPU 的编号也相等，则转移此进程；
2. 若某任务的 cpuindex 等于-1，则调度系统仍按原来的顺序执行，该进程不需要优化执行流程；
3. 若某任务的 cpuindex 不等于-1，并且与当前所在的 CPU 的编号相等，则该进程将不被选为被转移的对象；
4. 若某任务的 cpuindex 不等于-1，而与当前所在的 CPU 的编号不相等，且与当前 CPU 的编号也不相等，则不优先考虑转移此进程；
5. 若没能转移一个进程且 CPU 上的负载还是最忙的，则不考虑次优化方案，按调度系统原来的流程执行。

使用了优化方案的任务转移优先顺序如流程图如图 4-8 所示，其中不带背景颜色的部分是指原调度系统的工作流程，带背景颜色的部分为新添加的部分。

4.4.5 核心代码修改与实现

根据上文的设计，本文对内核源码进行修改的地方主要有三处：进程控制表 task_struct、Linux 内核调度系统的部分代码和 Futex 锁的进行唤醒部分的代码。

首先，需要对进程的 task_struct 结构体进行修改，是它能够记录所在 CPU 的编号，因为这种修改是基于 SMP 架构的，所以把该成员变量放在#ifdef CONFIG_SMP 和#endif 之间，如下所示：

```
struct task_struct {
    volatile long state;    //进程当前的状态。
```

```
    int cupindex;      //记录该进程当前 CPU 的编号。
    cpumask_t cpus_allowed;      //记录着可以运行该进程 CPU 掩码
    ……
};
```

其次，该方案还涉及到 futex 锁机制的修改，让它在发送唤醒信号的同时，把 CPU 的编号一起发给被阻塞的进程，因为该方案的实施只涉及了唤醒过程的改变，所以只对唤醒操作的系统调用进行修改即可。

在 Linux 内核中为 futex 锁提供的系统调用为：

```
asmlinkage long sys_futex(u32 __user *uaddr, int op, u32 val,
                struct timespec __user *utime, u32 __user *uaddr2, u32 val3);
```

而 sys_futex()函数是通过宏 SYSCALL_DEFINE6 进行扩展得到的，具体实现如下所示：

```
SYSCALL_DEFINE6(futex, u32 __user *, uaddr, int, op, u32, val,
            struct timespec __user *, utime, u32 __user *, uaddr2, u32, val3);
```

其中，参数 op 有多个选项，通过判断的值等于哪个宏值，而做出不同的操作，其中两个与本文相关的选项是 FUTEX_WAIT 和 FUTEX_WAKE，分别表示此次系统调用是等待还是唤醒，并最终通过 do_futex()进行统一处理的。

根据本文的需要，让唤醒的信号携带 CPU 的编号信息，所以需要重写 Futex 锁针对 FUTEX_WAKE 实现的代码，本文对其用函数 futex_wake()来实现，其核心修改代码如图 4-11 所示。

当唤醒进程调用完 futex_wake()后，被放入等待该锁的进程的 cpuindex 字段就设置好了最佳的执行该进程的 CPU。

最后，要修改 SMP 调度系统中的一些代码。

由图 4-4 可以清晰的看到，标准 Linux 内核中的每次平衡操作都要调用 load_balance()，而正在进行任务转移的是在函数 move_task()中实现的，如图 4-3 所示。所以我们将判断 cpuindex 的部分放到 move_task()中，它又调用了 load_balance_fair()，接着又调用了 balance_tasks()函数，它遍历了最忙队列中的进程，从任务的负载大小和缓冲区利用情况看看是否合适把一个进程转移到本地运行队列中来，如果合适就调用 pull_task()进行转移，如 4.2.2 节的讲解，所以要修改 balance_tasks()的函数实现。

任务的转移是在函数 balance_tasks()中确定的，所以要实施该解决方案，可以把条件的判断放到该函数中，首先把 cpuindex 等于本地 CPU 编号的任务进行转移，

```
static int futex_wake(u32 __user *uaddr, int fshared, int nr_wake, u32 bitset)
{
    /*用传入的地址 uaddr 填充 key 所对应的数据结构  */
    ret = get_futex_key(uaddr, fshared, &key);
    if (unlikely(ret != 0))/*  未找到该地址所对应的 Futex 锁*/
        goto out;
    hb = hash_futex(&key); /*根据 key 得到相应的等待队列*/
    spin_lock(&hb->lock);
    head = &hb->chain; /*得到等待队列的头结点*/
    cpu_no = task_cpu(current); /*获得唤醒进程的 CPU 编号  */
    plist_for_each_entry_safe(this, next, head, list) {/*对等待队列进行遍历*/
        if (match_futex (&this->key, &key)) {/*进行锁的匹配*/
            if (this->pi_state || this->rt_waiter) {
                ret = -EINVAL;
                break;
            }
            if (!(this->bitset & bitset))
                continue;
            this->task.cpuindex = cpu_no;/*给被唤醒进程的 cpuindex 赋值*/
            wake_futex(this);/*进行唤醒操作*/
            if (++ret >= nr_wake)
                break;
        }
    }
}
```

图 4-11 futex_wake()函数的核心实现

再考虑转移 cpuindex 等于-1 的进程，最后若对上边两类的进程转移完后还没有达到要求时，就对 cpuindex 不等于本地 CPU 编号的进程进行转移，重写后的 balance _tasks()函数流程图如图 4-12 所示。

通过这些流程，使得有共享资源多的进程尽可能的被分配到同一个处理单元上执行，以便避免出现共享资源的多个进程运行在不同的处理器上的情况，从而在一定程度上提升了高速缓存利用率。

最后，需要在进程创建时添加对 task_struct 中 cpuindex 字段的初始化，即就它初始化为-1，因为进程的创建都是最终通过 do_fork()来实现的，所以此步骤添加在此函数中。

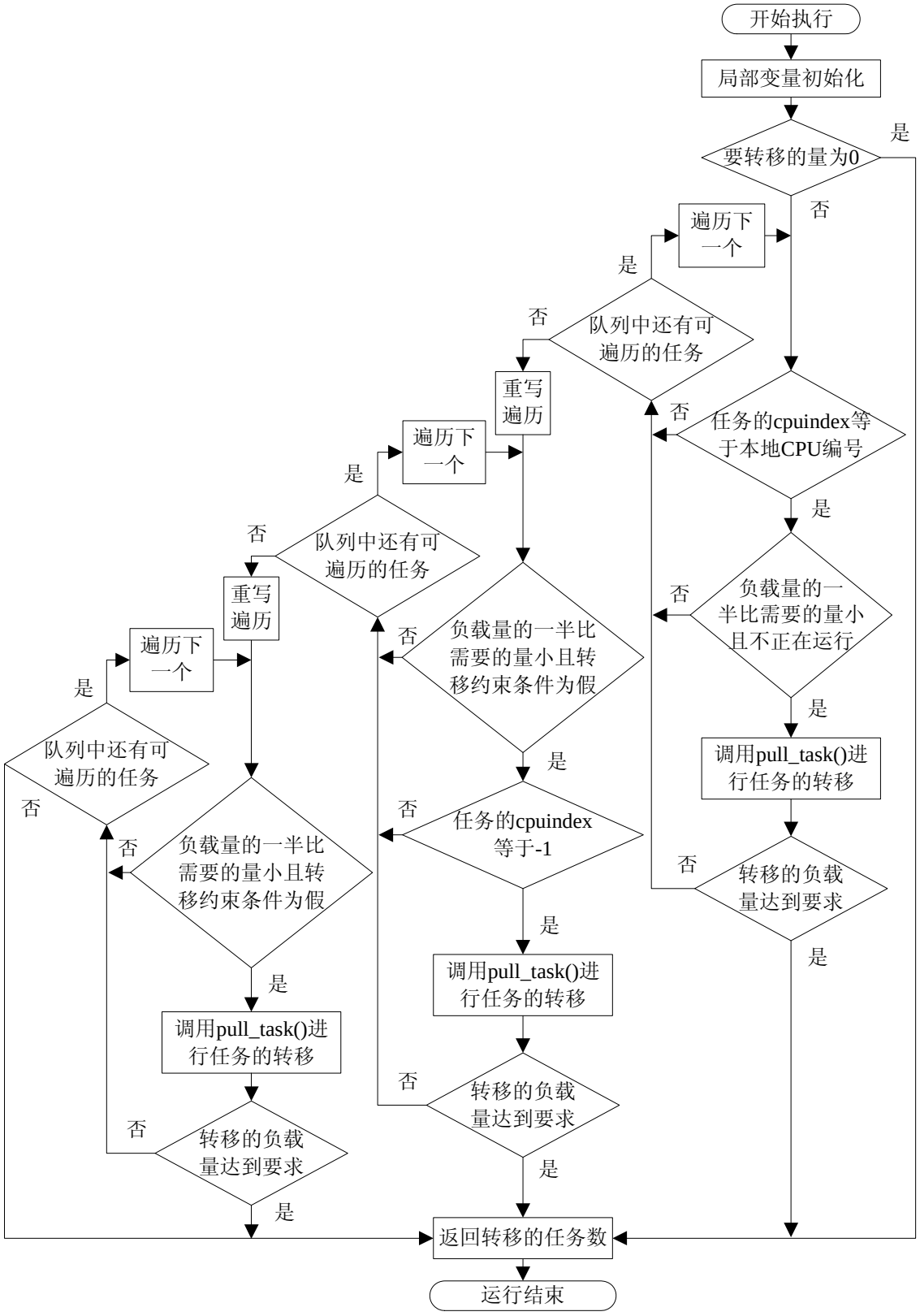


图 4-12 具有优化功能的 balance_tasks()流程图

4.4.6 对比测试

一般 SMP 系统用于大型的服务器、计算机集群等，但是现在用来做测试的设备条件有限，只能做一些简单的测试，来反映优化方案的性能，实验设备的配置如表 4-4 所示。

表 4-4 测试环境表

硬件的类型	硬件的指标
CPU 的数目	2
CPU 的频率	2.5GHz
CPU 型号	Xeon E5420
一级缓存	256KB
二级缓存	12MB
内存	2G
Linx 版本	2.6.36

因为上我们提成的基于唤醒信号反馈的优化系统是在有些任务之间存在共享的情况下，提升了高速缓存的利用率，因此，为了测试出优化方案的性能，我们设计的测试程序应该从能够反映高速缓存利用率上着手，但是很难对直接对高速缓存的利用率进行测量。

试想：如果同样的程序在同样的硬件设备上运行，CPU 的频率是一样的，内存是一样的，缓存是一样的等等，而唯独调度系统不一样，那么分别在这两个调度系统上运行同样的程序，如果运行的时间不一样，则说明调度的效率不一样，而且运行时间短的进程说明调度系统的效率高，所以，测量两组程序在同样的硬件条件，不同的调度系统的情况下，进行测试对比。

设计的测试程序情形如下：我们用多个进程竞争共享资源的方式来实现测试程序，我们设置每组进程有 4 个，让这 4 个进程共享一段独立的缓存区，用 Futex 锁进行保护，并对缓冲区的大小进行控制，当多组进程在访问各自的共享缓存区时，会重复利用 CPU 的高速缓存，这样，此优化方案就发挥优势了。本文在系统中同时运行 $n\times 4$ 个这样的进程，n 可以根据我们的需要来设置，主程序通过函数 gettimeofday()分别获取开始和结束的时间。为了对比明显，使用多组不同大小的共享内存来作比较，并进行多次测试取平均值。

为了使得实验数据的对比更加明显一些，我们把 n 分别取 10 和 20，设置的共享缓冲区分别取 512KB，1MB，2MB，4MB，8MB，进行测试，结果分别如图 4-13、

图 4-14 所示。

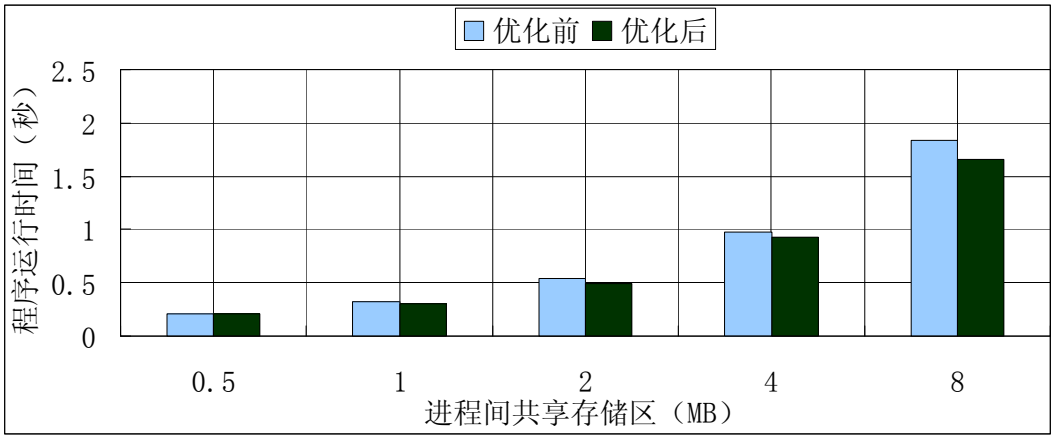


图 4-13 10 组进程共享缓存时的运行时间对比图

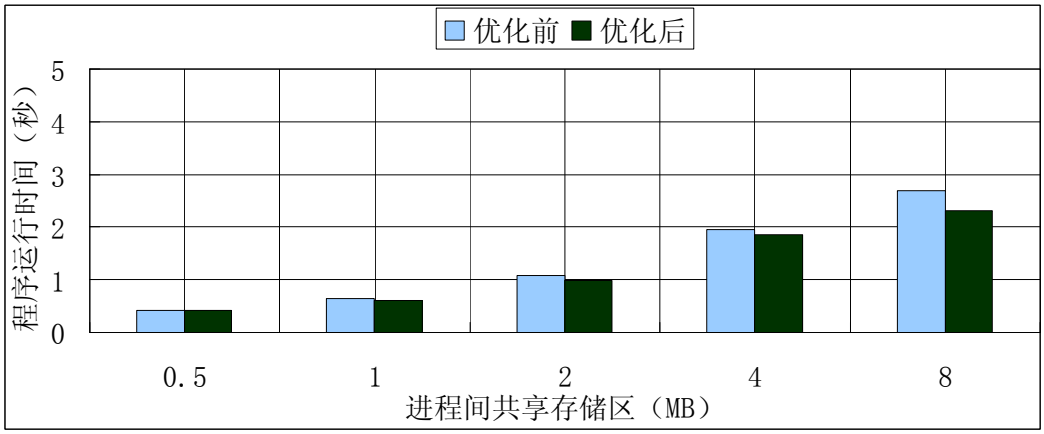


图 4-14 20 组进程共享缓存时的运行时间对比图

4.4.7 测试结果分析

通过比较图 4-13 和图 4-14 的数据可以看出，使用了优化方案的 Linux 系统上运行相同的程序时,程序执行所需要的时间比标准的 Linux 系统上的所需时间要短一点, 即系统所花费的时间要少，本文通过运行多次取的平均值，尽量避免了个别误差的影响，可以看出，使用了优化方案的调度系统比 Linux 2.6.36 标准的调度系统的执行总时间要短。

但是，从图 4-13 和图 4-14 还可以看出，在同等的任务强度下，当缓冲区变大的时，程序运行完成所节省的时间有所增加，但与缓冲区的增加量并不成比例，并没有缓冲区的增加量来的明显，而且在任务量加大时，即由10×4 增加到 20×4 时，在共享相同大小的缓冲区情况下，节省的时间差不多，也不存在比例关系，下面就从这两个方面，尝试分析这样状况的分析。

首先来讨论分析，在系统执行相同的负载量的情况下，当缓存区变大时，程序运行完成时所节省的时间虽然有所增多，但与缓冲区的增加量不成比例。

1. 随着进程间共享的缓冲区变大，进程与进程之间进行联系的数据增多，进程与进程之间共享的数据多了，那么第一个进程在 CPU 高速缓存中存放的数据，而这些数据中被用于其他具有共享关系的进程的比例就增加了，即高速缓存中能够被重复利用的数据比例增加了，如此一来，当具有共享关系的进程在该 CPU 上运行时，就会使得高速缓存的命中的机会比以前多，即高速缓存的利用率得到了提升，进而说明调度系统的效率增加了，最终会使得总的运行时间变少，这是必然的。
2. 另外，我们知道，处理器的高速缓存的大小是一定的，在本次实验中，一级缓存只有 256KB，二级缓存的大小为 12MB，当进程间的共享缓冲区增大时，我们选取的共享缓冲区比较大，这样就不能使共享的数据都被存放在处理器的 Cache 当中，在运行过程中，当 Cache 被用完时，Cache 中中的某些数据会被其他数据重新覆盖掉，所以即使让共享的数据再多，因而不会明显地提升 Cache 的命中率，对调度系统的效率也不会有很大的提升。因此，运行时间的缩短量与进程共享缓冲区的增加量不成比例。

其次，再来讨论在共享缓冲区相等的情况下，当任务量增加时，程序运行完成所需的时间基本相同，这可能有三个方面会影响这个情况。

1. 当任务量增加时，就容易让系统中陷入不平衡的状态，然而，当前的实验环境中只有 2 个处理器，任务被分配到合适的处理器上的概率是 1/2，即不管系统的任务有多少，它们有一半的机会不需要进行优化转移。因此，即使任务增多，优化方案有 1/2 的概率得不到发挥。
2. 因为每个任务除了运行在调度系统中外，还需要其他的操作，如申请内存空间，初始化 futex 锁等等操作，而且优化方案也会有一部分系统开销，当这样的任务按倍数增长时，这些方面的花费也会按照倍数增长，不但如此，由于系统的资源是有限的，随着任务的增加，在申请资源时发生的冲突会增多，这就使得花费可能更多。而且优化方案的执行与会随着任务的增加而增多。因此，虽然优化方案节省了一点时间，但被这些额外的花费而消耗掉。
3. 跟上面第二种分析的情况一样，处理器的高速缓存大小一定，随着任务数的增多，使得 Cache 失效的机率也增加了，从而不利于提升高速缓存的利用率，进而影响了整个程序的执行效率，使得运行时间加长了。

最后由于我们的实验环境的简陋，CPU 的数目不是很多，而且我们设计的测试程序可能会比现实应用中的情况要简单一些，致使测试不能更加贴切的反映出优化方案在实战中的效率。但是，从测试的结果可以说明优化方案是正确的。而且根据优化方案的设计思想，当环境中的处理器比较多，并且任务负载也比较大时，效果会更好一些。

本文的优化方案只有在征用锁时才会触发，且借助一种锁的争用情况进行了处理，但是实际应用中，进程从创建到消亡可能不只要用到一种锁，一个进程与多个进程都有共享资源，从而造成进程间是网状关系，不便于对它们进行分组。如何让所有的锁都能够考虑到，对这些具有网状结构关系的进程作出合理的安排，是更一层的优化方向。

4.5 本章小结

该章首先介绍了关于 SMP 调度系统的一些概念，并对 Linux kernel 中关于 SMP 的实现作了分析，接着根据分析和结合这方面的文献资料，改进了调度系统的模型并用<P，T，M，S>表示，以便于进一步的学习和研究，并结合对 SMP 调度性能的分析，提成了基于唤醒信号反馈的调度优化方案，并进行了设计、实现以及测试和分析。

第五章　总结与展望

应用需求的日益提升，推动了计算机技术的快速发展，从巨型机到微型机，从单核到多核，无不经历了革命性的跨越。然而需求没有停止，尤其是当前多核操作系统还尚不成熟，经典的几大操作系统都不能在核数很多的情况下完美胜任，所以，这里潜伏着机遇和挑战。而 Linux 平台作为一个开源的操作系统，对研究、学习和进行实验给予了不错的选择，该论文就是在剖析 Linux 内核源码的基础上，重点研究了 CFS 调度系统和 SMP 调度系统，在此基础上提出了自己的改进想法，并在最后提出、设计、并修改实现了一个优化方案。

Linux 也在经历着变革，从最早的 1.0 版本发展到现在的最新的 3.2.11，每个版本的发布都见证了它的成长，从最近的功能很简单的小内核，发展到现在的成熟、稳定、功能完善的操作系统，展现了它强大的生命力和卓越的研发途径。

所以以一种崇敬的心情阅读了 Linux 内核的源代码，并最后把焦点放到了进程的调度上来。调度系统从原来的 O(1)调度到现在的 CFS 调度，有不少的改变，采用了新的理论和算法，不再对进程的种类进行区分，这不仅使设计更加的简单，而且使得运行的性能更好，对每个进程更加的公平。通过对 CFS 源代码的解析，使我们对进程的调度有了更加细致、全面的认识。

我国在计算机技术的研发和创新上一直落后国外，无论是在大型机还是在个人电脑、小型机方面，我们一直在后面追赶，而现在多核系统领域还没有被抢占，而 SMP 系统的应用需求很大，现在的多核个人电脑已经在我们身边普及，而大型的服务器也要求能够及时的响应，来满足网络上发来的各种请求，所以这是一个很好的机会，我们应该加大研发力度，在这个方面有所突破。为此，本文针对 SMP 调度系统做了深入的研究，从它的基本知识入手，充分体会它的设计理念，掌握了 SMP 系统是如何把系统中的 CPU 划分调度域，来实现在一个系统中使用不同的调度策略，以及 SMP 系统的同步机制来保障多个 CPU 安全的并行执行，并按照进行查看负载的时机和负载调整的策略两条线索研究了 SMP 调度系统的源码，把握它查看负载情况的时机和应对不平衡时所做的策略，在此基础上，提出了本文的猜想，并结合在大量阅读文献，对已有的调度系统的模型进行了改进并用<P，T，M，S>来表示，以便于进一步的学习和研究，并结合对 SMP 调度性能的分析，对可以改进的地方进行了深入的研究和分析，最后，依据上边所在的研究，提出

并设计了基于唤醒信号反馈的 SMP 调度系统的优化方案，并以 Linux 内核为基础进行了源码修改和实现，通过做实验得出的结果来验证方案的正确性，对结果的一些方面做出了分析。

到现在为止，已经有很多的前辈对 SMP 调度系统和负载平衡的算法进行了研究，也取得了丰富的研究成果，然而随着永无止境的应用需求的不断提高，硬件性能的日渐升级，使得 SMP 系统依然有着很大的挑战，还有很大的空间等着我们去开拓，去创新，还有一些方面值得我们去思考，去探索：

1、大多的 SMP 系统会定期的执行，通过遍历完系统中所有的调度域来判断系统是不是处于负载不平衡状态并作出相应的调整，但是如果系统中 CPU 的很多，负载的很重，遍历一次需要很长的时间，所以 Linux 在检查的负载平衡上可以提出更好的方法，如折半查找，使用哈希表等等。

2、不论在分配进程还是负载的平衡时，应当充分考虑进程之间的耦合度，例如一个家族的进程或有亲缘关系的线程安排到一起执行，应该能够提高 Cache 的命中率。

3、虽然目前对负载转移的因子有很多的研究，如运行队列长度、资源的利用率、CPU 温度等，虽然有些确实因素确实能反映去处理器的负载情况，而且在模拟实验中也得到了很好的效果，但是实现起来又有点复杂，对这些因素如何去进行权衡并付诸实践，着值得我们去思考。

4、本文的优化方案只是针对一类锁进行了优化，而在现实应用中，进程之间的关系是复杂的，它们之间的联络不是简单的线性关系，如何对这类的进程进行分配和负载平衡等。

致 谢

本文的最后，我要感谢我的导师顾小丰老师。学习 Linux 内核的一年多来，使我感受到了一流编程大师的风采，也深刻的认识到了自己的浅薄与不足。从选题到调研老师都给我很细致的指导和意见，到最后的论文撰写和审查时，老师又给我认真的指点和一遍遍的查阅，老师那为人和蔼、平易近人的形象让我非常的感动。不仅如此，老师为我的学习也给予了很好的环境和严格的要求，还对我经济不太好的家境给予了关怀和帮助。

我要感谢我的师兄和同学们，他们在生活上陪伴着我度过了美好的三年时光，并在我撰写论文时给了我很多帮助，在代码实现过程中，也帮我克服了重重困难。

我要感谢学校，给我们提供了非常好的休息、生活、运动、读书的环境，让我在不知不觉中度过了我最快乐的一段时光，也让我在快乐中得到了成长，得到了提高。

我要感谢我的爸妈，是他们养育了我，含辛茹苦，默默地疼爱着我到现在。

参考文献

[1] 张琦，许胤龙. 多核系统中的程序性能优化研究：[博士学位论文]，安徽：中国科学技术大学, 2010

[2] Tan, I.K.T. Chai, I. Poo Kuan Hoong. Pthreads Performance Characteristics on Shared Cache CMP, Private Cache CMP and SMP[J]. Computer Engineering and Applications (ICCEA), 2010 Second International, 2010: 186-191

[3] 李锋涛，郑晓曦. 基于 SMP 结构的 linux 内核进程调度的研究[J]. 数字技术与应用, 2011, (11): 89-90

[4] Hiroshi Inoue and Toshio Nakatani . Performance of multi-process and multi-thread processing on multi-core SMT processors[J]. Workload Characterization(IISWC), 2010 IEEE International Symposium on, 2010:1-10

[5] Wancy, Betti. Scheduling Strategy of Subtask Decomposed. Advanced Information Networking and Applications, 2008(5): 383-386

[6] J. Tuya, A.Corrales. A pragmatic task design approach based on a Ward/Mellor real-time structured specification[M]. SpringerBerlin, 2008

[7] 韩建军，李庆化. 同构计算环境中一种新的静态任务调度算法. 华中科技大学，2005(3): 145-167

[8] 梁根，郭小雪，秦勇.基于公平调度算法的分布式系统负载均衡研究[J]. 计算机工程与设计，2008.3,29(6): 1362-1363

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

[10] 覃中，李毅. 基于多核系统的线程调度:[硕士学位论文]. 成都：电子科技大学, 2009

[11] 胡丽聪，徐雅静，徐慧民. 基于动态反馈的一致性哈希负载均衡算法[J]. 微电子学与计算机，2012: 29(1): 177-180

[12] 赵磊. 适应多核处理器的任务调度研究：[硕士学位论文]，哈尔滨：哈尔滨理工大学, 2010

[13] 廖江苗，陈虎. 多核处理器上的并行 B+树索引算法研究与实现：[硕士学位论文]. 广州：华南理工大学, 2010

[14] 曹皓. 多核处理器体系结构下 Linux 调度机制的研究：[硕士学位论文]. 呼和浩特：内蒙古大学, 2011

[15] 王正霞，刘晓洁，梁刚. 基于 B+树快速调优的反馈式负载平衡算法[J]. 计算机应用, 2011, 31 (3): 609-612

[16] 张少辉，马骏. 基于 BP 算法的动态负载平衡预测：[硕士学位论文]. 开封：河南大学, 2009

[17] 叶经纬，祝永新.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

[18] 刘婷，王华军，王光辉. 基于 Linux 内核的 CFS 调度算法研究[J]. 电脑与电信, 2010, (1): 61-63

[19] 宫团基，夏克俭. 查找、插入及删除时间为 O(1)的方法研究[J]. 计算机工程与设计, 2010, 31 (9): 2097-2100

[20] Park, J. Lim, H. Kim, H. Development of kernel thread Web accelerator[J]. Electronics Letters. IEEE, 2002, 38(13): 672-673

[21] 葛君,郑凤婷. Linux 系统中进程调度策略[J]. 湖北：桑丘职业技术学院学报,2009, (5): 42-44

[22] 邹治锋. 基于 Linux 进程调度算法的改进与实现:[硕士学位论文]. 无锡：江南大学, 2006

[23] 杨静，李炜，万峰松，吴建国. Linux 内核 2.6 进程调度分析与改进[J]. 计算机技术与发展, 2009, 19(7): 105-107

[24] 张颖慈，吴跃. 数据结构在操作系统进程调度中的应用研究:[硕士学位论文]. 成都：电子科技大学, 2009

[25] Robert Love. Linux Kernel Development, Third Edition[M]. Sams, 2007

[26] 张荣亮，余敏. Linux 操作系统内核分析与研究:[硕士学位论文]. 南昌：江西师范大学, 2007

[27] 毛得操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社, 2001

[28] Jacek Kobus, Rafa Szklarski. Completely Fair Schedule and its tuning[EB/OL]. http://www. fizyka. umk.pl/~jkob/prace-mag/cfs-tuning.pdf

[29] 使用完全公平调度程序（CFS）进行多任务处理[EB/OL]. http://www. ibm.com/developerworks /cn/linux/l-cfs/

[30] 张桂兰，王飞超. Linux 内核完全公平调度器的分析及模拟[J]. 中国科技信息, 2009, (04): 134-137

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L, Rivest and Clifford Stein. Introduction to Algorithms, Second Edition[M]. The MIT Press, 2001

[32] 杜慧江，王云光. Linux 内核 2.6.24 的 CFS 调度器分析[J]. 计算机应用与软件, 2010, 27(2): 166-168

[33] Wong, C.S. Tan, I.K.T.Kumari, R.D. Lam, J.W. Fun, W. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers.[J].IEEE. Information Technology, 2008. ITSim 2008. International Symposium on, 2010: 88-96

[34] 高世杰. 基于 Linux 的对称多处理机系统结构研究[D]. 武汉大学：计算机系统结构, 2002

[35] McCurdy, C. Vetter, J. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms[J]. Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on,2010: 87-96

[36] Suresh Siddha ,Venkatesh Pallipadi. Chip Multi Processing aware Linux Kernel Scheduler[R]. Ottawa:On-tario Canada, 2006, 2: 329-338

[37] 施文佳，杨斌. 对称多处理器下基于调度域的超线程实现[J].成都信息工程学院学报, 2010, 25(2): 146-150

[38] Robert Love. Linux Kernel Development, Third Edition[M]. China Machine Press, 2011

[39] R. Rajwar and J. Goodman. Transactional Lock-free Execution of Lock-based Programs[J]. ACM Press, 2002: 5–17

[40] 汤子瀛,哲凤屏,汤小丹. 计算机操作系统[M]. 西安：西安电子科技大学出版社, 2002: 39-40

[41] 管 理 处 理 器 的 亲 和 力 [EB/OL].http://www.ibm.com/developerworks/cn/linux/l-affinity.html#resources

[42] Jiexi Zha, Junping Wang, Renmin Han, Maoqiang Song. Research on load balance of Service Capability Interaction Management [J]. Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on, 2010: 212-217

[43] Antonopoulos, Christos D, Nikolopoulos, Dimitrios S, Papatheodorou, Theodore S. Scheduling algorithms with bus Ban- dwidth considerations for SMPs[J]. Parallel Processing, 2003. Proceedings. 2003 International Conference on, 2003, 547-554

[44] Youhui Zhang, Ziqiang Qian, Weimin Zheng. A Software-Controlled Cache Coherence Optimization for Snoopy-based SMP System[J]. Computational Intelligence and Natural Computing, 2009. CINC '09. International Conference on, 2009: 155-157

[45] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler[J]. Silicon Graphics, Inc., 2005

[46] M Harchol-Balter, A B Downey. Exploiting process lifetime distributions for dynamic load balancing [J]. ACM Transactions on Computer Systems, 2003, 15(3): 253-285

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

[48] 蒋江,张民选,廖湘科.基于多种资源的负载平衡算法的研究[J].电子学报, 2002, 30(8): 1148-1152

[49] Zhaoliang Guo, Qinfen Hao. Optimization of KVM Network Based on CPU Affinity on Multi-cores[J]. Information Technology, Computer Engineering and Management Sciences (ICM), 2011 International Conference on, 2011:347-351

[50] 范光雄，李毅.多处理器系统中的线程调度研究:[硕士学位论文]. 成都：电子科技大学, 2009

[51] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking[J]. In Proceedings of the Ottawa Linux Symposium, Ottawa, Canada, 2002: 479-495

[52] U. Drepper. Futexes are tricky[EB/OL]. http://people.redhat.com/drepper/futex.pdf

[53] 余敦福，李鸿健，唐红，豆育升. 基于反馈机制的动态负载平衡算法研究[J].计算机应用研究, 2012, 29(2): 527-529

硕士学位论文

MASTER DISSERTATION