电子科技大学

硕士学位论文

基于多核系统的线程调度

姓名：覃中

申请学位级别：硕士

专业：计算机系统结构

指导教师：李毅

20090501

摘 要

微处理器自诞生以来，性能的提升主要是通过提高主频来实现的，而主频的提升要求大幅度增加晶体管的数量。巨大的晶体管数量则意味着巨大的能耗，随之带来的散热问题也日益凸显。当处理器性能受到半导体工艺限制的时候，研究人员将目光投到了处理器的体系结构，提出了多核处理器的概念。多核处理器已经成为微处理器发展的趋势，使用多核技术可以使微处理器的性能得到极大的提高，但同时也对调度策略提出了更高的要求。

本文首先分析了课题研究的背景和意义，简要介绍了进程、线程和多线程以及在单处理器调度的基本理论知识。介绍了多核处理器硬件相关的主要实现技术，包括同时多线程(Simultaneous Multithreading，SMT)，片上多处理器(Chip MultiProcessor，CMP)，片上多线程(Chip Multiple Threading, CMT)；并分析了目前多核线程调度的研究现状。

其次，本文详细研究了 Linux2.6 内核的 O(1)调度算法及其具体实现。O(1)调度器中增加了数据结构 runqueue，就绪队列被分成活动队列和过期队列，结合 bitmap[]不必遍历整个就绪队列，查找 next 进程的时间复杂度降为 O(1)，进程运行时间片的重新分配更及时；动态优先级的计算过程更简单。详细分析了在 Linux 中的 SMP 的具体实现，着重讨论了 Linux 负载均衡系统。然后指出了 Linux2.6 调度算法的不足：1. 因为在处理器间迁移不同进程的代价是不尽相同的，所以在迁移进程的时候，应该适当考虑进程的特点。2. 调度器给处理器分配进程的时候应该考虑进程的相关性。3.当系统的负载不平衡且很轻微的时候，不一定需要平衡负载。

最后，提出一种负载均衡的通用模型，使用四元组<E，T，L，S>来表示。然后根据该模型的各个因子对 Linux 的负载均衡系统进行剖析，着重分析了 Linux 的负载评价因子 L 和调度策略因子 S。针对 Linux 调度器不考虑进程迁移代价的不足提出了基于资源利用率的负载均衡算法，通过计算进程的 CPU 利用率和内存利用率来选择迁移进程，然后通过统计系统中的过载 CPU 和轻载 CPU，根据 Donor 或者 Reciever 算法来匹配源 CPU 和目的 CPU。

关键词：多核处理器，对称多处理，Linux 进程调度，O(1)调度算法，负载均衡

ABSTRACT

Since microprocessor was born, the enhancement of its performance has been achieved by increasing its frequency, of which the upgrade requires a substantial increase of the number of the transistors, requiring more power consumptions and causing significant heat dissipation. When cpu's performance was restricted by the semiconductor craft, people turned to multi-process architecture and proposed a new concept of multi-core process.Nowadays, multicore processor has become the trend of microprocessor and the utilization of multicore technology increases the performance of microprocessors significantly. Furthermore, it puts forward higher requirements on the scheduling strategy.

First, this thesis describes the background and the significance of the research, introduces briefly the basic theoretical knowledge on process, thread and multi-thread and scheduling policies on UP. After that, it outlines the implement of corresponding hardwares on mulit-core and the current study of multi-thread scheduling.

And then,the O(1)scheduling algorithm and it's implementation in Linux 2.6 kernel are researched detailed.Data structure runqueue is added and it's separated into two parts:active and expired . Integrate with bitmap[],the time complexity of O(1) scheduler is reached to O(1).The reallocation of the process' timeslice becomes more timely and the calculation of process' priority becomes more simple than before. Then , it analyses the specific implementation of the SMP and focuses on the load balancing system in Linux. After that, three deficiencies of scheduler of linux2.6 are pointed out as follows: 1. no special considerations for the features of processes ,for the prices are different when migration occurs among CPUs. 2. the lack of relativity of processes dispatched by the scheduler.3. unnecessary load-balancing when the unbalance of loads is tiny in the system.

Finally, this thesis proposes a general template for load balancing, which is signified with a four-dimension vector. The load balancing system of Linux is analysed according to the factors in the module, where L factor for loading evaluation and S factor for sheduling policy are concentrated. This thesis puts forward a Load balancing

algorithm based on the utilization of resources for the shortages mentioned above. The algorithm selects process to be migrated by calculating the CPU utilization and memory utilization, statistic over load CPU and light load CPU, and matches the source CPU and the purpose CPU according to Donor algorithm or Receiver algorithm.

Keywords: multi-core processor, multi-processor architecture, Linux process schedule, O(1) schedule algorithm, load balancing

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：曾中　　日期：2009年 6月2日

关于论文使用授权的说明

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

（保密的学位论文在解密后应遵守此规定）

签名：曾中　　导师签名：李敬

日期：2009年 6月 2日

第一章 绪论

1.1 课题背景

从 20 世纪 70 年代开始，处理器的发展突飞猛进，许多知名厂家纷纷推出一代又一代处理器。随着处理器继承晶体管数量越来越多，主频越来越高，而处理器的性能也越来越强大[1]。

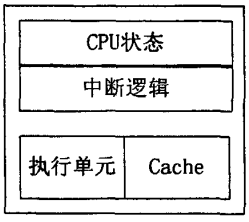
1965 年，Intel CEO 戈登摩尔提出了著名的摩尔定律：IC 上可容纳的晶体管数目，约每隔 18 个月便会增加一倍，性能也将提升一倍。半导体工艺的不断发展，的确带来了处理器性能的提高，但是到了 21 世纪，处理器频率已接近 4GB，集成的晶体管数量有数亿个，半导体工艺达到了物理极限，很难再提高处理器的频率。另外，单纯的通过提高主频来提升性能，引起功耗的不断提高。

处理器性能的提高主要依赖于两方面：一是半导体工艺的飞速发展；二是处理器体系结构的不断创新。当处理器性能受到半导体工艺限制的时候，研究人员将目光投到了处理器体系结构，提出了多核处理器的概念。多核，即在一个单芯片上面集成两个或两个以上处理器内核，每个内核都有控制单元、逻辑单元、中断控制器、运算单元、一级缓存、二级缓存（可以是共享或独有），这些部件和单核处理器内核相比都是一样的。多核体系通过增加计算机中物理处理器的数量，能够有效地利用线程级并行性，支持了真正意义上的并行执行。

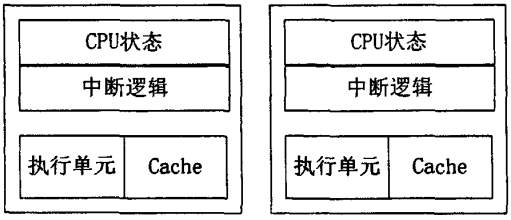
为了充分利用线程级并行性所带来的性能收益，研究人员首先提出了同时多线程（Simultaneous Multi-Threading，SMT）技术。线程，可以被定义成 CPU 资源占用的一个基本单位，包括指向指令流中当前指令的程序计数器、当前线程的 CPU 状态信息以及其他一些资源，比如堆栈。通过复制这些状态信息的方法就能够创建多个逻辑处理器，然后执行资源就被这些不同的逻辑处理器所共享。当多核处理器出现之后，可以将相互独立的执行核与 SMT 技术相结合，从而将逻辑处理器上的数量增加到执行核数量的两倍。图 1-1 给出了各种不同的处理器结构[2]。

多核处理器的发展，对操作系统提出了新的要求和挑战。首先，如何合理地分配、组织和调度任务才能最大程度地发挥多核处理器结构的性能?其次，如何保持操作系统的外部接口的相对稳定?对用户而言，当然希望从单核到多核能够平滑的过渡。一方面系统的操作界面最好和以前的操作系统完全相同，另一方面以前

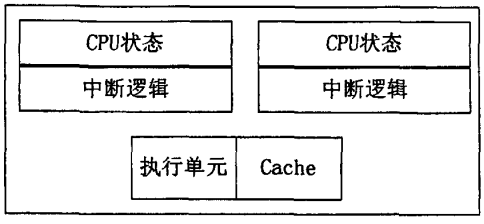
的应用程序最好能够不做任何修改就直接在多核处理器系统上运行。也就是说多核这个特性对于用户来讲是透明的，用户不知道也不关心底层的实现问题，这需要操作系统在用户界面和编程接口方面都保持不变[3]。



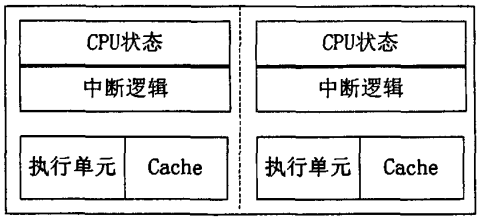
（a）单核结构



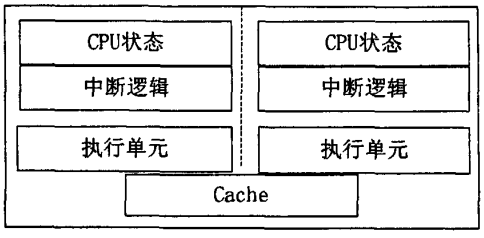
（b）多处理器结构



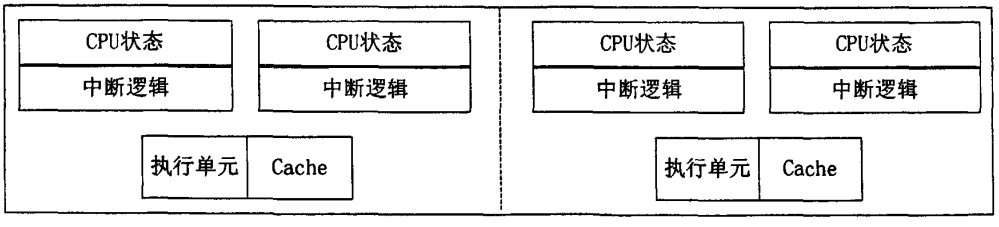
（c）超线程技术



（d）多核体系结构



（e）共享cache的多核体系结构



（f）采用超线程技术的多核体系结构

图 1-1 单核结构、多处理器结构以及多核结构之间的简单对比

如何更好地分配、组织和调度任务以便将多核处理器结构的性能发挥到极致是多核操作系统的核心问题，这也是人们对多核处理器最大的期望。要解决这个问题，需要软硬件共同协作，从任务调度、中断分配、资源共享等几个方面入手。硬件方面则要求多核处理器系统提供全新的同步与互斥、中断分配以及 CPU 内核

之间的中断等机制[1]。

目前为止，世界上还没有专门针对多核体系结构的多核操作系统出现，尤其是缺乏成熟的调度机制。因此，面向多核体系结构的操作系统调度是目前多核软件的一个研究热点。

1.2 多核系统调度问题的提出

目前，面向多核体系结构的操作系统调度研究的热点主要有以下几个方面：任务的分配与调度，程序的并行研究，多进程的时间相关性，进程间通信，缓存的错误共享，一致性访问研究以及多核处理器核内部资源竞争等。这几个因素相互独立又相互依赖，因此考虑一个系统的性能时必须将其中的几点同时加以考虑而不能只考虑其中一个因素。

核的任务的分配是多核时代提出的新概念。单核时代，没有核的任务分配问题，因为只有一个核的计算资源可以使用。而在多核时代，有多个核的计算资源可以被使用，这就要考虑核的任务分配问题。如果系统中有几个进程需要分配，要么一起分配到一个核，要么将它们均匀的分配到各个核，或者是按照一定的算法进行分配。任务分配结束后，接下来需要考虑任务调度。对于不同的核，每个核都可以有自己独立的调度算法来调度执行不同的任务，也可以使用一致的调度算法。此外，还可以考虑一个进程上一个时间片运行在一个核上，下一个时间片是否还继续在这个核上运行，还是进行线程迁移让其他核来承载该线程剩下的执行活动？系统的核资源分配不平衡时是否要进行负载均衡？怎样直接调度实时任务和普通任务……

对于多核处理器系统的调度，目前还没有明确的标准与规范。由于多核系统有多个处理器核可用，必须负责分配，有可能为每个处理器核提供单独的队列。在这种情况下，一个具有空队列的处理器核就会空闲，而另一个处理器核就会很忙，所以如何处理好负载均衡问题是这种调度策略的关键问题所在。为了解决这种情况，可以考虑共同就绪队列，所有处理器共用一个就绪队列。但这无疑是增加了进程上下文切换、锁的转换的执行时间，降低了系统的性能。另外一种考虑就是创建主从结构，选择一个处理器来为其他处理器调度。有的系统将主从结构进行扩展，采用单一处理器来处理所有调度的调度策略、I/O 处理和其他系统活动。只有一个处理器处理访问系统数据，从而减轻了数据共享需要，但它的执行效率并不高[1]。

1.3 论文的主要工作

本课题主要讨论了在多核处理器平台上对 Linux 操作系统调度中负载均衡系统的扩展和改造，主要研究内容如下：

1. 多核处理器体系结构的调查研究
2. 多核系统下线程调度的调查研究
3. 对 Linux2.6 内核的 O(1)调度算法分析
4. 对 Linux2.6 负载均衡系统的分析
5. 具体规划并设计基于资源利用率的负载均衡系统

1.4 论文组织结构

本文接下来的章节将具体介绍课题的内容。第 2 章中，主要介绍与多核体系结构相关的一些基础知识；第 3 章将介绍 Linux2.6 版本内核的调度机制及其存在的问题；第 4 章提出一个通用的负载均衡模型，根据该模型定性分析 Linux 的负载均衡系统，针对 Linux 不考虑进程迁移代价的不足提出了基于资源利用率的负载均衡系统；第 5 章对本课题进行归纳和总结，指明改进和完善方向。

第二章 多核系统的有关技术基础

2.1 基本概念

2.1.1 进程

“进程”这一术语，在60年代初期，首先在麻省理工工学院的MULTICS系统和IBM公司的CTSS/360系统中引入[4]。我们把“进程”定义为可并发执行的程序在一个数据集合上的运行过程。或者说，进程是进程实体的运行过程。

进程有五个基本特征：动态性、并发性、.独立性、.异步性和结构特征（由程序段、数据段以及进程控制块三部分组成）。

进程在运行中不断地改变其运行状态，进程的基本状态有：

新建（New）：刚刚创建的进程，操作系统还没有把它加入到可执行进程组中，通常是还没有加载到主存中的新进程。

就绪（Ready）：进程做好了准备，只要有机会就开始执行。

运行（Running）：该进程正在被执行。

阻塞（Blocked）：进程在某些事件发生前不能执行。如I/O操作完成。

退出（Exit）：操作系统从可执行进程组中释放出的进程，或者是因为它自身停止了，或者是因为某种原因被取消。

2.1.2 线程

为了在共享存储环境下有效地开发应用程序的细粒度并行度，将进程分成两部分，分派的部分称作“线程（thread）”或“轻量级进程（lightweight process）”，而资源所有权的部分仍称作进程或任务。线程是进程中的一个实体，是被系统独立调度和分派的基本单位。线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器、一组寄存器和栈），但它可以与同属一个进程的其他线程共享进程所拥有的全部资源。

2.1.3 多线程

多线程是指操作系统支持在一个进程中执行多个线程的能力。在支持多线程

的系统中，进程成为资源分配和保护的实体，而线程是被调度执行的基本单元。进程的资源包括进程的地址空间，打开的文件和 I/O 等资源。属于同一个进程的线程共享进程的代码段和数据段，打开的文件、信号等。除了共享资源，每个线程还包含各自的线程 ID，线程执行状态，CPU 寄存器状态和栈。如图 2-1 所示：

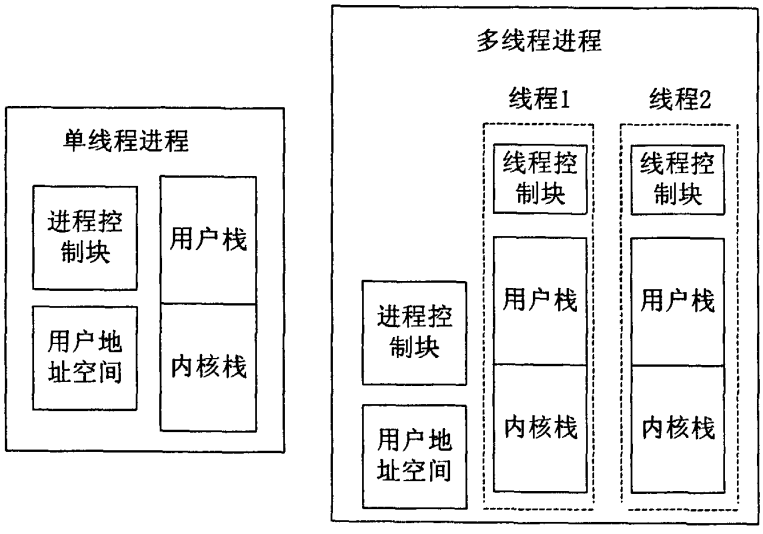


图 2-1 单线程和多线程的进程模型

采用多线程一般主要有两个主要目的：一是为了应付各种并发性问题，二是为了加快任务处理的速度。二者表面上是类似的，但实际上前者是被动的，后者是主动的。一个多线程的进程可以利用线程来管理相互独立的工作，包括执行大量运算、用户界面等[6]。

2.2 单核处理器调度

2.2.1 处理器调度的类型

操作系统必须为多个进程可能有竞争的请求分配计算机资源。对处理器而言，可分配的资源是在处理器上的时间，分配途径是调度（scheduling）。调度功能必须设计成可以满足多个目标，包括公平、任何进程都不会饿死、低开销和有效地使用处理器时间。此外，调度功能可能需要为某些进程的启动或结束考虑不同优先级和实时最后期限。

处理器调度的目标是以满足系统目标（如响应时间、吞吐率、处理器效率）的方式，把进程指定给一个处理器执行。在许多系统中，这个调度活动分成三个

独立的功能：长程（long term）调度、中程（mindium term）调度和短程（short term）调度[5]。

1. 长程调度

长程调度决定哪一个程序可以进入到系统中处理，因此它控制多道程序的程度。一旦允许进入，一个作业或用户程序成为一个进程，并被添加到短程调度程序使用的队列中。在某些系统中，一个新近创建的进程从一个换出条件开始，在这种情况下，它被添加到供中程调度程序使用的队列中。

2. 中程调度

中程调度是交换功能的一部分，换入决策基于管理多道程序程度的要求。对不使用虚存的系统，存储器管理也是一个问题。因此，换入决策将考虑换出进程的存储要求。

3. 短程调度

按照执行的频率，长程调度程序的执行频率相对较低，并且仅仅是粗略地决定是否接受新进程以及接受哪一个。为进行交换决策，中程调度程序执行得略微频繁一些。短程调度程序，也称作分派程序（Dispatcher），执行的最频繁，并且精确地决定下一次执行哪一个进程。

2.2.2 调度算法

1. 先来先服务

最简单的策略是先来先服务（FCFS），也称作先进先出（FIFO）或严格排队方案。当每个进程就绪后，它加入就绪队列。当前正在运行的进程停止执行时，选择在就绪队列中存在时间最长的运行。

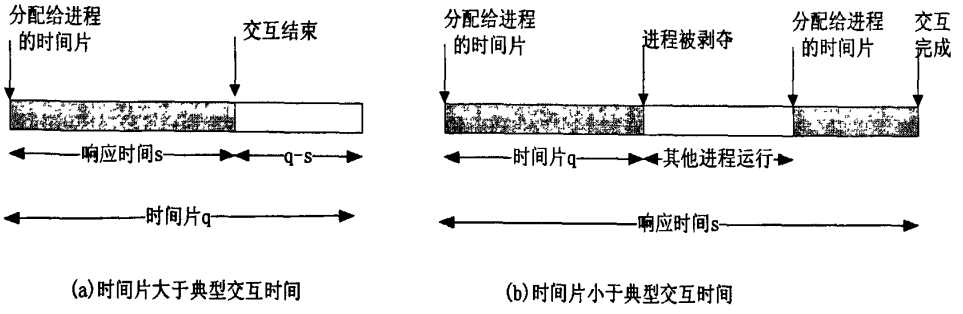
FCFS 执行长进程比执行短进程更好。如果一个短进程紧随着一个长进程之后到达，那么它在系统中的总时间比所需要的处理时间长很多。FCFS 的另一个难点是相对于受 I/O 限制的进程，它更偏爱受处理器限制的进程。

2. 循环

为减少 FCFS 策略下短作业的不利情况，一种简单的方法是基于时钟的剥夺。以一定的间隔周期性地产生一个时钟中断，当中断发生时，当前正在运行的进程被置于就绪队列中，然后基于 FCFS 选择一下就绪作业运行。这种技术也称作时间片（time slicing）。因此每个进程在被剥夺前都给定一个时间片。

对于循环法，基于设计问题是使用时间片的长度。如果时间片非常短，则短作业会相对比较快地移动通过系统。另一方面，处理时钟中断、执行调度和分配函数都需要处理器开销。典型的，时间片最好略大于一次典型的交互所需要的时

间。如果小于这个时间，大多数进程都会需要两个时间量。图 2-2 显示了时间片的大小对响应时间的影响。



(a)时间片大于典型交互时间　　(b)时间片小于典型交互时间

图 2-2 时间片大小的影响

3. 最短进程

减少 FCFS 固有的对长进程的偏爱的另一种方法是最短进程（Shortest Process Next，SPN）策略。这是一个非剥夺的策略，其原则是下一次选择所需处理时间最短的进程。因此，短进程将会越过长作业，跳到队列头。

SPN 的风险在于只要持续不断地提供更短的进程，长进程就有可能被饿死。另一方面，尽管 SPN 减少了对长作业的偏爱，但是由于缺少剥夺机制，它对分时系统或事务处理环境仍然不理想。

4. 最短剩余时间

最短剩余时间（Shortest Remaining Time，SRT）是对 SPN 增加了剥夺机制的版本。在这种调度机制下，调度程序总是选择预期剩余时间最短的进程投入运行。当一个进程加入就绪队列时，它可能比当前运行的进程具有更短的剩余时间，因此，只要新进程就绪，调度程序就剥夺当前进程并使新进程运行。和 SPN 一样，调度程序在执行选择函数时必须包含对处理时间的估计，并且存在长进程被饿死的危险。

5. 最高响应比

根据排队模型，周转时间（turnaround time，TAT）就是驻留时间，或这一项在系统中花费的总时间（等待时间加服务时间）。一个更有用的数字是标准化的周转时间，它是周转时间与服务时间的比率，可作为质量因子，这个值表明一个进程的相对延迟。考虑下面的比率：

$$R=\frac{w+s}{s} \tag{2-1}$$

其中，R 是响应比，w 是等待处理器的时间 s 是期待的服务时间。

调度规则如下：在当前进程完成或被阻塞时，选择 R 值最大的就绪进程。这个方法非常具有吸引力，因为它说明进程的年龄。当偏爱短作业时（因为小分母产生大比率值），长进程由于等不到服务的时间增加，从而使这个比率值增大了，最终在竞争中胜了短进程。

6. 反馈

基本思想如下：调度基于剥夺（按时间片）使用动态优先级机制。当一个进程第一次进入系统中时，它被放置在 RQ0。当它第一次执行后并返回就绪状态时，它被放置在 RQ1。在随后的时间里，每当它被剥夺时，它被降级到下一个优先级队列中。在每个队列中，除了在优先级最低的队列中之外，都使用简单的 FCFS 机制。一旦一个进程处于优先级最低的队列中，它就不可能再降低，但是会重复地返回这个队列，直到运行结束。图 2-3 显示一个进程经过各种队列的过程来说明反馈调度机制。

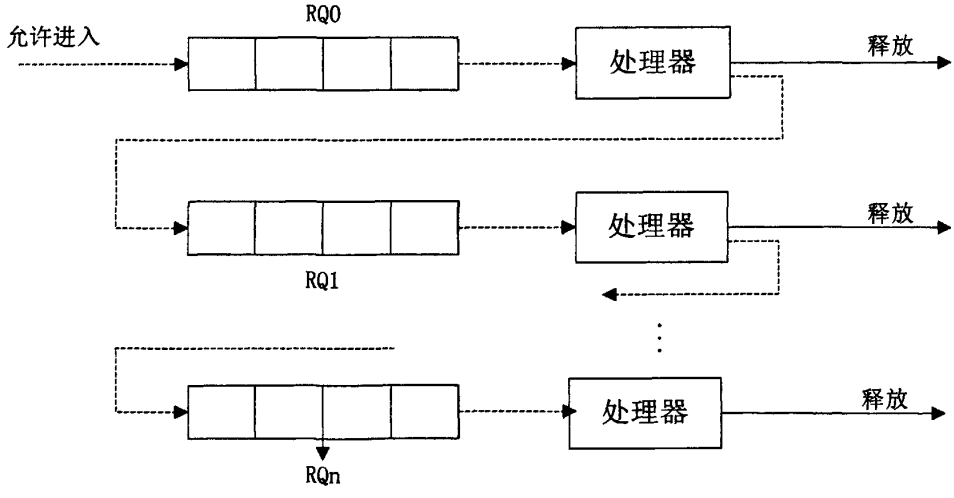


图 2-3 反馈调度

2.3 多核相关技术

随着单个芯片上晶体管数目的增加，处理器设计者需要提出下一代高性能处理器的体系结构。近年来，高端处理器主要利用超标量（Superscalar）技术来提高性能。超标量处理器在每个时钟周期发射多条指令到功能部件上执行，其目的是

利用程序的指令级并行性（Instruction-Level Parallelism，ILP）来提高性能。但是单个程序的有限 ILP 导致了超标量处理的资源利用率不高，进一步增加指令发射宽度只会加重这种情况。对于下一代高性能处理器而言，开发的并行性不应该仅限于单个程序内细粒度的 ILP。实际上，在许多工作负载中，存在多种形式的粗粒度的线程级并行性（Thread-Level Parallelism，TLP）。例如，对于多道编程负载而言，存在多个独立运行的应用；而并行程序则可能包含多个线程或者进程。

研究人员提出多种利用 TLP 来提高处理器资源利用率的处理器体系结构，包括多线程处理器（Multithreaded Processor）、单片多处理器（Chip MultiProcessor，CMP）和同时多线程（Simultaneous Multithreading，SMT）结构。2005 年 SUN 公司提出了片上多线程技术（Chip Multiple Threading，CMT），即将 CMP 技术和 SMT 技术结合起来。

2.3.1 SMT 结构

SMT 结构的基本思想是：在一个时钟周期内发射多个线程的指令到功能部件上执行[8]，以提高功能部件的利用率。同时多线程的概念是加利福尼亚大学的 Tullsen 在 1995 年提出的，他提出的 SMT 结构只对传统超标量处理器结构做了很少改动，但是获得了很好的性能[9]，因此，主流的商业微处理器主要受它影响，如 DEC Alpha21464[10]、Intel Xeon[11]。

SMT 结合了超标量和多线程处理器的特点，可以同时减少水平和垂直浪费。SMT 在两个方面提高了处理器的总体性能[9]：

1) SMT 允许在一个时钟周期内执行来自不同线程的多条指令。在一个时钟周期内，SMT 能够同时利用程序的 ILP 和 TLP 来消除水平浪费，提高处理器发射槽以及功能部件的利用率。
2) 从理论上来说，SMT 允许任何活动线程的组合来发射指令。当由于在长延迟操作或者资源冲突导致只有一个活动线程时，该线程能够使用所有可获得的发射槽。这使得可以通过使用其他线程的未阻塞指令来消除垂直浪费。

基于乱序执行（Out-Of-Order Execution）的超标量处理器的 SMT 结构解决了包括大寄存器读写和指令调度在内的具体实现问题。图 2-4 为该方法的处理器结构模型[12]；图 2-5 为传统超标量处理器流水线和 SMT 的流水线[9]。

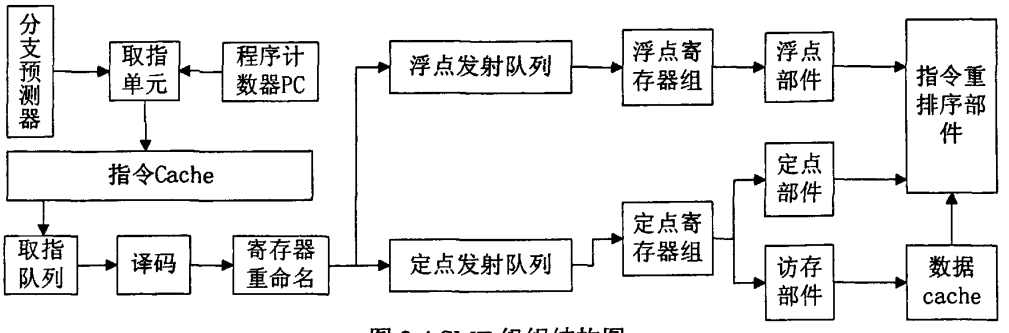
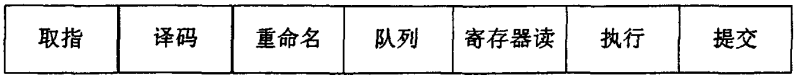


图 2-4 SMT 组织结构图

(a)超标量处理的流水线



(b)SMT结构的流水线

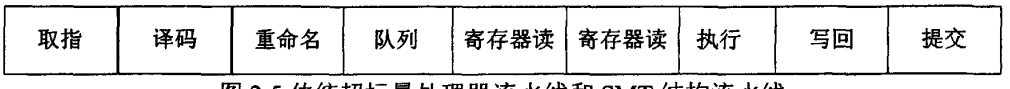


图 2-5 传统超标量处理器流水线和 SMT 结构流水线

Intel 公司所实现的 SMT 技术就是超线程(Hyper-Threading, HT)[13]技术。超线程技术实际上只有一个实际的物理处理器但是从软件角度来看，存在多个逻辑处理器。超线程技术支持操作系统和应用程序将多个线程调度到多个逻辑处理器上，就像多处理器一样。从微体系结构的角度来看，逻辑处理器的指令都是固定的，并且在共享的执行资源上同时执行。

2.3.2 CMP 结构

在同时多线程技术之后就出现了多核处理器。在 1996 年斯坦福大学的研究人员提出了单片多处理器（Chip Multi-Processor，CMP）结构，并进行了研究[14]。单片多核处理器的主要思想是通过简化超标量结构设计，将多个相对简单的超标量处理器核集成到一个芯片上，从而避免线延的影响，并充分开发 TLP，提高吞吐量。CMP 存在的主要问题是由于单片多处理器系统的资源是采用划分方式的,当没有足够的线程时,资源就浪费了[15]。

按计算内核的对等与否，CMP 可分为同构多核和异构多核。计算内核相同，地位对等的称为“同构多核”，现在 Intel 和 AMD 主推的双核处理器就是同构的双核处理器。计算内核不同，地位不等的称为“异构多核”，异构多核多采用“主处理器核+协处理器核”的设计，IBM、索尼和东芝等联手设计推出的 Cell 处理器正

是这种异构架构。处理核本身的结构，关系到整个芯片的面积、功耗和性能。怎样继承和发展传统处理器的成果，直接影响多核的性能和实现周期[1]。

CMP 处理器的各 CPU 核心执行的程序之间有时需要进行数据共享与同步，因此其硬件结构必须支持核间通信。高效的通信机制是 CMP 处理器高性能的重要保障，目前比较主流的片上高效通信机制有两种，一种是基于总线共享的 cache 结构，一种是基于片上的互连结构。总线共享 cache 结构是指每个 CPU 内核拥有共享的二级或三级 cache，用于保存比较常用的数据，并通过连接核心的总线进行通信。这种系统的优点是结构简单；通信速度高，缺点是基于总线的结构可扩展性较差。基于片上互连的结构是指每个 CPU 核心具有独立的处理单元和 cache，各个 CPU 核心通过交叉开关或片上网络等方式连接在一起。各个 CPU 核心间通过消息通信。这种结构的优点是可扩展性好，数据带宽有保证；缺点是硬件结构复杂，且软件改动较大。

典型的同构多核处理器是 Intel 最新的多核处理器架构—Core 微架构[16]，Core 微架构拥有双核心、64bit 指令集、4 发射的超标量体系结构和乱序执行机制等技术，使用 65nm 制造工艺生产，支持 36bit 的物理寻址 和 48bit 的虚拟内存寻址，支持包括 SSE4 在内的 Intel 所有扩展指令集。Core 微架构的每个内核拥有 32KB 的一级指令缓存、32KB 的双端口一级数据缓存，且 2 个核心的一级数据缓存之间可以直接传输数据，2 个内核共同拥有 4MB 或 2MB 的共享式二级缓存。每个核心内建 4 组指令解码单元，支持微指令融合与宏指令融合技术；每个核心内建 5 个执行单元；采用新的内存相关性预测技术。如图 2-6 所示。

典型的异构多核处理器是由索尼、东芝和 IBM 共同研发的高性能多核心微处理器——Cell[16]处理器，它以 IBM 的 PowerPC 微处理器为基础，具备独特的非对称多核心设计和庞大的总线与内存带宽，拥有高性能的浮点数运算性能。Cell 架构是一个单芯片多核心处理单元，它内置 9 个处理单元，它们之间共享存储器资源，从功能块来看，它具有两种类型的处理单元：Power Processor Element (PPE)、the Synergistic Processor Element (SPE)，PPE 64-bit 架构核心同时兼任 32-bit 程序，而 SPE（又称协处理单元）则是一个独立处理单元，每个 SPE 可以全权访问内存储器，它们之间通过高速通道有效连接起来，共同交互完成各种复杂工作[3]。如图 2-7 所示。

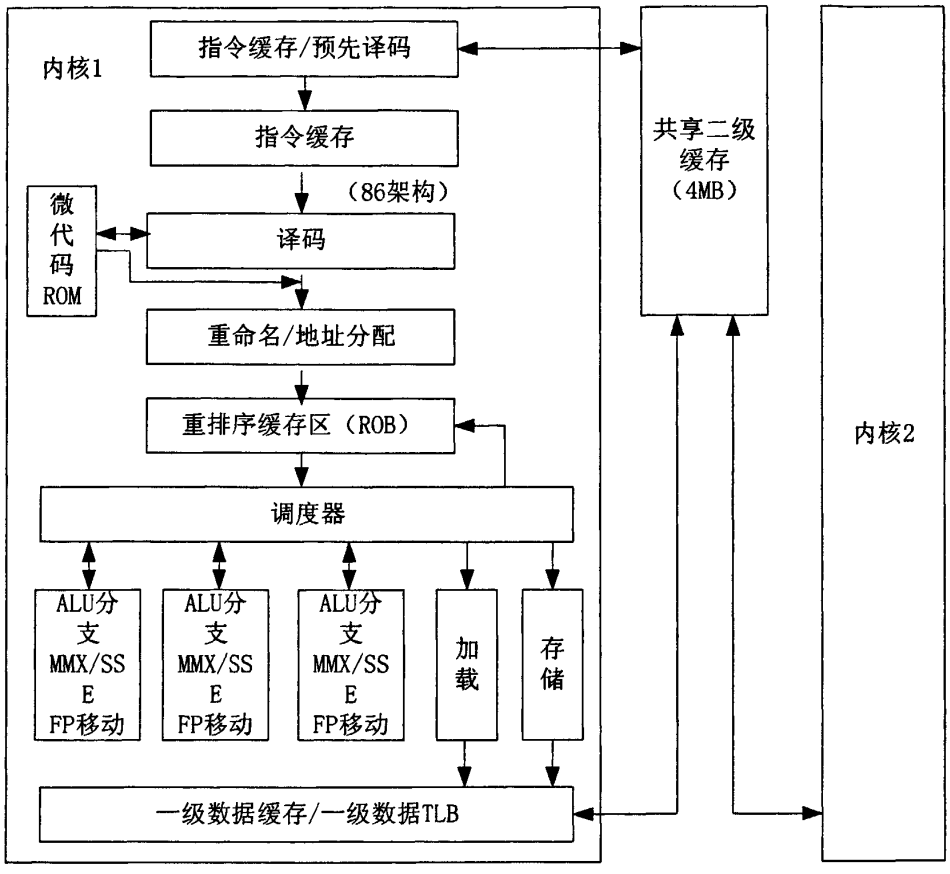


图 2-6 Core 微结构示意图

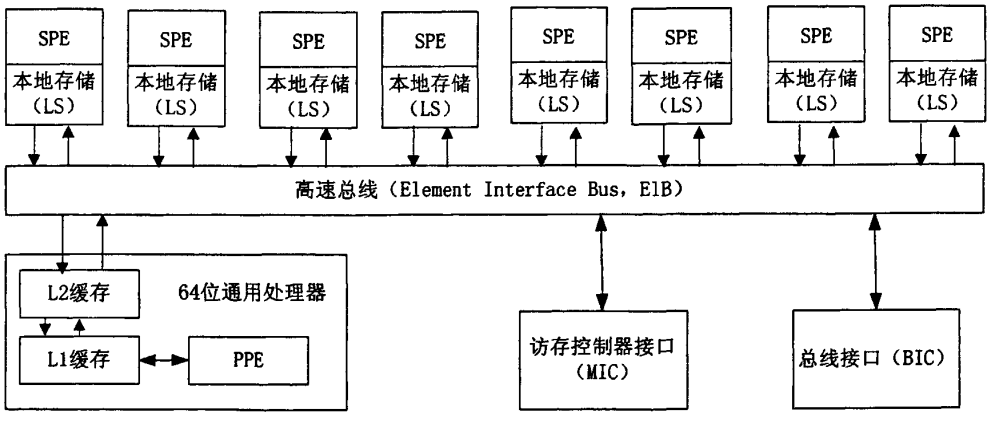


图 2-7 Cell 结构示意图

2.3.3 CMT 结构

对于 TLP 较低 CMP 不能充分被利用这种情况，CMT 是一种很好的选择。CMT 是 SMT 和 CMP 的结合体，既可以利用 SMT 充分利用资源的优点，又可以结合 CMP 的简单性[10]：

1) 利用 SMT 的特点，每个时钟周期可以发射多个线程的指令到执行部件，这样可以减少垂直浪费。SMT 的这一特点也可以进一步减少水平浪费。
2) 利用 CMP 的特点，不同的线程在不同核上执行，提高资源的利用率，减少水平浪费。

2005 年 SUN 公司提出了片上多线程技术（Chip Multiple Threading，CMT），即将 CMP 技术和 SMT 技术结合起来。SUN 公司同时宣称将在其新一代处理器 Niagara 中使用这种新的多核多线程技术。Niagara 处理器主要包含了 8 个处理器核，每个处理器核可以同时运行 4 个线程[19]。Niagara 是 CMP 和细粒度多线程的结合，每个核是 6 站台的单发射处理器，采用 CMT 技术最大限度地减少垂直浪费。Niagara[20]结构图如图 2-8 所示：

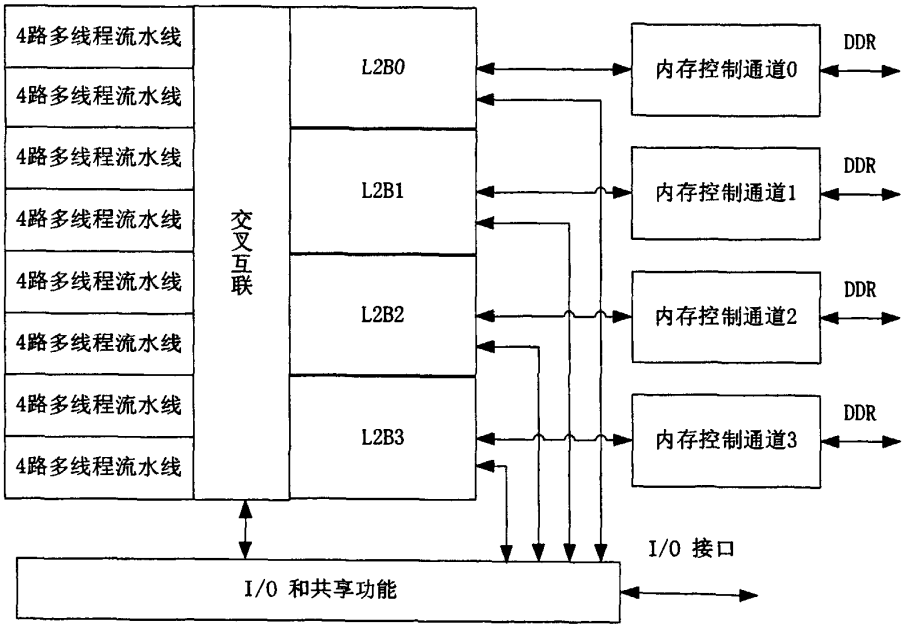


图 2-8 Niagara 框图

多核、多线程处理器带来了更高的系统性能、更低的系统开销、更低的功耗、同时能够保正已存在的软件架构的前期投入。多核、多线程处理器技术具有普通单核、单线程处理器所未有的性能优势,目前已成为提高处理器性能的主要途径,同

时,多核、多线程处理器也为很多的应用领域提供了新的解决方案。

2.3.4 SMP 结构

对称对处理机(Symmetrical Multi-Processing ，SMP)是指在一个计算机上汇集了一组处理器(多 CPU)，各 CPU 之间共享内存子系统以及总线结构。在这种架构中，计算机不再由单个 CPU 组成，而由多个处理器同时运行操作系统的单一副本，并共享内存和计算机上的其他资源。系统将任务队列对称地分布于多个 CPU 之上，所有 CPU 都可以平等地访问内存、外部中断和 I/O。在对称多处理系统中，系统资源被系统中所有 CPU 共享，工作负载能够均匀地分配到所有可用处理器之上，从而极大地提高整个系统的数据处理能力[22]。虽然使用多个 CPU，但是从用户角度来看，它们的表现就像一台单机一样。SMP 体系结构[23]如图 2-9 所示：

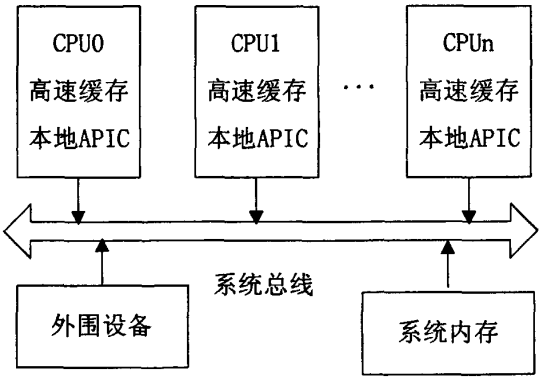


图 2-9 SMP 体系结构图

2.3.4.1 SMP 的操作系统类型

对称多处理机操作系统，目前有三种类型主从式(master-slave)OS、独立监督式(separatesupervisor)OS 和浮动监督式(floatingsupervisor)OS。

1. 主从式(master--slave)

主从式操作系统由一台处理机记录、控制其它从处理机的状态，并分配任务给从处理机。操作系统在主处理机上运行，从处理机的请求通过陷入传送给处理机，然后主处理机回答并执行相应的服务操作。主从式操作系统的监控程序及其提供的过程不必迁移，因为只有主处理机利用它们。主从式 OS 的缺点是：当不可恢复错误发生时，系统很容易崩溃，此时要重启主处理机。由于主处理机扮演的角色非常重要，当它来不及处理进程请求时，其他从属处理机的利用率就会随之降低。

2. 独立监督式(separatesupervisor)

独立监督式与主从式不同，在这种类型中，每一个处理机均有各自的管理程序(核心)。

3. 浮动监督式(floatingsupervisor)

每次只有一台处理机作为执行全面管理功能的“主处理机”，但根据需要，“主处理机”是可浮动的，即从一台切换到另一台处理机。这是最复杂、最有效、最灵活的一种多处理机操作系统，常用于对称多处理机系统(即系统中所有处理机的权限是相同的，有公用主存和 I/O 子系统)。浮动监督式操作系统适用于紧耦合多处理机体系。

2.3.4.2 SMP 设计的原则

对称多处理机操作系统的设计与单处理机多道程序设计的不同主要基于两点：

1. 系统的体系结构

各处理机是异构的还是同构的，如果各处理机是异构(nonhomogeneity)的而不是同构(homogeneity)的，则各台处理机上的可执行代码格式不一样，浮动管理就困难。有一个解决的办法就是通过仿真软件来“缩小”异构机之间的硬件差异。在非对称主存系统中，不是所有的处理机都能访问所有的存储器，这给操作系统的主存管理带来复杂性。

2. 同步

一个并行程序由两个或多个相互作用的进程组成，进程间的同步和通信显得更加重要，因为并行算法设计得不是很有效，就会降低并行系统的性能。同步原语通常由软件实现，也可由硬件实现，但硬件实现性能要好些，它可以减少系统在同步上的开销。

1) 同步的实现

如果多处理机器的系统是共享存储的，那么各处理机上的进程可通过共享存储器进行同步。但是，对于松耦合的多处理机，各处理机有很大的独立性，它们的进程一部分采用分布式同步机制，另一部分采用集中式。在狭义的多处理机系统中，进程间更多的是同步且其实现相对简单。广义的多处理机系统利用计算机网络与分布式系统实现进程间的通信。

2) 利用中心进程实现同步

中心进程是多处理机系统管理程序的一部分，它保存了所有用户的冲突图

(Conflict graph)和存取权限(Access permissions)等信息。每一个要求访问共享资源的进程先向中心进程发送请求消息，中心进程收到该请求后首先去查看冲突图。如果该请求不会引起死锁，就将它插入请求队列，否则将请求退回。当轮到该请求对应的进程使用共享资源时，中心进程便向该对应进程发送一个应答信息并让其进入临界区访问共享资源。在退出临界区时，该进程还得向中心进程发送一个释放资源的消息，中心进程收到消息后又向请求队列的下一个请求进程发送应答消息，允许它进入临界区。如此继续下去，直到请求队列为空。可以说，中心进程的作用是为了使共享资源有序地被访问，因此它又称为协调进程。由于每个要访问共享资源的进程都需要申请、应答、释放 3 个消息传递，同步的效率并不高[22]。

2.4 多核线程调度研究现状

随着多核处理器的发展，对软件开发有非常大的影响，而且瓶颈在软件上。软件开发在多核环境下的核心是多线程开发。这个多线程不仅代表了软件实现上需采用多线程，要求在硬件上也采用多线程技术。只有与多核硬件相适应的软件才能真正地发挥多核的性能。多核对软件的要求包括对多核操作系统的要求和对应用软件的要求。

本节先从 CMP 结构的调度研究开始介绍，然后介绍 SMT 结构的线程调度策略研究现状，最后再介绍 CMT 结构的线程调度策略研究现状。作为一种新的体系结构，CMT 的线程调度策略研究是建立在 SMT 结构相关研究基础上的[24]。

2.4.1 CMP 结构调度研究

目前为止，世界上还没有专门针对 CMP 体系结构的多核操作系统出现，尤其是缺乏成熟的 CMP 调度算法。多核操作系统的关注点在于进程的分配和调度。进程的分配将进程合理地分配到物理核上，因为不同的核在历史运行情况下和共享性都是不同的。有的物理核能够共享二级 Cache，而有的却是独立的。如果将有数据共享的进程分配给有共享二级 cache 的核上，将大大提升性能；反之，就可能影响性能。进程调度会涉及到比较广泛的问题，比如负载均衡、实时性等。

下面是目前几个具有代表性的多核调度算法。

1）对任务的分配进行优化。使同一应用程序的任务尽量在一个核上执行，以便达到有共享数据的任务能够尽量在一个核上运行的目的，而共享数据量少或者没有数据共享的任务尽量在不同的核上运行。这样，可以显著地降低 cache 的缺失

率，进而在很大程度上提升系统的整体性能。

2）对任务的负载均衡优化。当任务在调度时，出现了负载不均衡的情况，考虑将较忙处理器中与其他任务最不相关的任务迁移，以达到数据的冲突量减小。

3）对任务的共享数据优化。由于 CMP 体系结构共享二级缓存，可以改变任务在内存中的数据分布，尽量使任务在执行时增加二级缓存的命中率。

2.4.2 SMT 结构调度研究

在 SMT 结构中，当系统中线程数目超过所支持的硬件线程数时，操作系统的调度器就需要调度线程集合的一个子集合到处理器上运行。由于在 SMT 机器上同时执行的一组线程在每个时钟周期都要竞争硬件资源，这使得执行不同的线程集合可能会获得不同的性能。操作系统的调度器的作用是从等待运行的任务集合中，选择一组适合同时执行的线程并调度执行[3]。

Snavely A[25]首先研究了 SMT 结构上的调度问题，提出了“共生（Symbiotic）”概念，用来衡量多个线程在 SMT 上同时执行的有效性。

Snavely A 等[26]针对 SMT 结构提出了一种调度算法：S.O.S(Sample,Opitmize,Symbiosis)。S.O.S 主要包括采样阶段和共生阶段。S.O.S 策略在采样阶段，利用一组硬件性能计数器从线程集合中收集不同线程子集合的执行信息，然后在共生阶段利用这些信息来预测不同作业子集的性能，并选择最佳性能线程子集来进行调度。实验表明，在 SOS 方案中，一小部分样本就可以快速确定进行调度的最佳性能子集；其性能比随机调度方案提高 17%。

由于 SOS 方案没有考虑优先级，Sanvely，Tullsen 和 Volker 等[27]提出了一种适用于 SMT 结构的调度机制，可以保证进程优先级和共生调度低优先级线程和高优先级线程都能提高系统吞吐量。仿真结果表明，考虑了优先级的调度策略可以将响应时间降低到 33%。

Parekh，Eggers 和 Levy 等人[28]也研究了基于 SMT 结构的调度算法，提出了线程敏感的调度算法。为了使处理器的吞吐量达到最大，线程敏感调度算法采用线程行为反馈来选择一起执行的最佳线程集合。线程敏感调度算法，可以权衡性能和公平性。与时间片轮转算法相比，基于 IPC 线程敏感调度算法可以用最小的硬件成本将加速比提高 7%~15%。

2.4.3 CMT 结构调度研究

随着多核时代的到来，在机器中使用片上共享 cache 技术变得普遍起来。尽管这样的共享有利于隐藏线程间通信延迟和 cache 的灵活使用，但是却引起了 cache 的争用。当前研究表明，协同调度——分配合适的线程到同一个芯片上——有利于减轻 cache 争用带来的影响[29]。目前对协同调度的研究主要分为两类：一类是依赖于运行时间采样，比如共生调度。CMT 共生策略是 SMT 共生策略的进一步扩展。CMT 共生策略对运行时间的性能进行采样，然后选择性能最好的进行调度。该策略对运行时间进行采样的方法对线程数目少的情况效果很好。另一类包含了 profiling-directed 技术。profiling-directed 技术先分析线程的运行，然后对它们进行协同调度。虽然这些技术表现出了一定的效率，但是如果实际的执行输入不同于实验中的输入，我们并不清楚它们将如何工作。

当搜索空间扩展的时候，基于采样的智能调度策略变得不那么有效。Electron 策略[9]与这类策略不同，它依赖更多清晰的证据，比如一个特定的核是否过度调度，适当调度或者不能调度。该策略计算每个核的EDF(Energy Delay Product)，在运行中定期将EDF最低的核上的一个线程转移到EDF最高的核上，经过若干次这样的操作，就可能会出现空闲的核。这时，彻底释放该处理器核，从而达到降低功耗的效果。

在 CMT 系统中，根据线程特征对所有资源进行核间划分可以减少核间资源争用。王晶[24]等提出首先对核间 L2cache 进行划分，缓解核间资源争用情况，然后对核内线程进行协同调度减少核内资源竞争的策略。该策略作为一种软硬件相互适应的解决 CMT 系统资源共享问题的方案，可能会得到令人满意的性能。

2.5 典型的支持多核处理器的操作系统

传统的通用的操作系统都是支持多任务执行的，对单处理器中唯一的一个计算核心通过分时处理，把时间片轮流分配给各个任务使用，从而实现单个 CPU 执行多个任务的能。对于多核这种新的体系架构，操作系统并不需要多少修改就已经能够很好地支持了，从软件角度来看，多核处理器和多路处理器是一样的，所有针对单核多处理器的软件优化方式都可以用在多核处理器系统上。比如 Windows NT 之后的 Windows 系列操作系统，其中可以支持 SMP 的都可以支持多核。而对于 Linux 操作系统内核，其 SMP 版本的内核能够很好地支持多核 CPU，Linux 2.0

内核是第一个支持对称多处理器硬件的内核。

在 2.6 版本的 Linux 中，提出了一种新的 O(1)的进程调度器，此调度器可以更好地支持 SMP 系统。它的优势在于平衡了多个 CPU 的负载的同时又能有效地兼顾 cache 的有效性。Linux 2.6 内核通过给每一个 CPU 设立单独的任务队列，有效地实现系统中的各个 CPU 的负载均衡。内核划分了 140 个优先级，高 100 个优先级用于实时任务，低 40 个优先级用于普通任务。分配给每个就绪任务一个时间片，时间片未用完的任务留在活动队列，当时间片用完后任务就转移到过期队列并重新分配时间片；等活动队列的所有任务的时间片均用完之后，就交换活动队列和过期队列。这样，调度器为每个任务提供了公平使用 CPU 的机会，并通过“CPU 亲和力”实现了任务和 CPU 的绑定。

微软的 Windows 操作系统对于多核的支持也非常容易解决，微软的操作系统支持多任务操作，支持 IA 架构 32 位与 64 位处理器。Windows 目前能够在多核处理器上运行，但并没有针对这类处理器进行优化。

2.6 本章小结

本章首先介绍了进程、线程以及多线程的定义，接着描述了单核处理器的调度策略，然后重点介绍了目前与多核相关的几种技术，包括 SMT，CMP，CMT 和 SMP 等，并分析了目前多核线程调度的研究现状。最后列举了目前典型的支持多核处理器的操作系统。

第三章Linux 调度机制

Linux 操作系统是自由软件和开发源码发展中最著名的例子，本课题是以Linux2.6 内核版本为基础展开的。在这一章中，我们先着重介绍 Linux2.6 版本的调度机制，特别是 O(1)调度算法。然后详细分析了在 Linux 中的 SMP 的具体实现，讨论了 Linux 负载均衡系统。最后指出了 Linux2.6 调度算法的不足等等。由于 Linux 内核并没有区分进程和线程这两者在概念上的差异，在内核代码中进程和线程都使用更通用的名字—任务，根据同样的思路，本文中所出现的“任务”和“线程”具有相同的意义。

3.1 Linux2.6 的调度机制

Linux 调度程序定义于 kernel/sched.c 中。现在的调度程序与以前版本的调度程序区别很大，实现了下列目标[30]：

1) 充分实现 O(1)调度。不管有多少进程，新调度器采用的每个算法都能在恒定的时间内完成。
2) 加强交互性能。即使在系统处于重载的情况下，也能保证系统的响应，并立即调度交互式进程。
3) 保证公平性。在合理设定的时间范围内，没有进程会处于饥饿状态，也没有进程能够得到大量时间片。
4) 实现 SMP 的可扩展性。每个处理器拥有自己的锁和自己的可执行队列。
5) 强化 SMP 的亲和力。尽量将相关一组任务分配给同一个 CPU 进行连续的执行，只有在需要平衡任务队列的大小时才在 CPU 之间移动进程。

3.1.1 调度程序所使用的数据结构

3.1.1.1 新的数据结构 runqueue

数据结构 runqueue 是 Linux2.6 调度程序最重要的数据结构。系统中每个 CPU 都有自己的运行队列，所有的 runqueue 结构都存放在 runqueues 每 CPU 变量中。表 3-1 列出了 runqueue 数据结构所包含的字段。

表 3-1 runqueue 结构

类型	名称	说明
spinlock_t	lock	保护进程链表的自旋锁
unsigned long	nr_running	运行队列链表中可运行进程的数量
unsigned long	cpu_load	基于运行队列中进程的平均数量的 CPU 负载因子
unsigned long	nr_switches	CPU 执行进程切换的次数
unsigned long	nr_uninteruptible	先前在运行队列链表中而现在睡眠在 TASK_UNINTERRUPTIBLE 状态的进程数量
unsigned long	expired_timestamp	过期队列中最老的进程被插入队列的时间
unsigned long long	timestamp_last_tick	最近一次定时器中断时间戳
task_t *	curr	指向当前正在运行进程
task_t *	idle	当前 CPU 上 swapper 进程的进程描述符的指针
struct mm_struct *	prev_mm	在进程切换期间用来存放被替换进程的内存描述符地址
prio_array_t *	active	指向活动进程链表的指针
prio_array_t *	expired	指向过期进程链表的指针
prio_array_t[2]	arrays	活动和过期进程的两个集合
int	best_expierd_prio	过期进程中静态优先级最高的进程(权值最小)
atomic_t	nr_iowait	先前在运行队列的链表中而现在正等待磁盘 I/O 操作结束的进程的数量
struct sched_domain *	sd	指向当前 CPU 的基本调度域
int	active_balance	如果要把一些进程从本地运行队列迁移到另外的运行队列就设置这个标志

task_t *	migration_thread	迁移内核线程的进程描述符指针
struct list_head	migration_queue	从运行队列中被删除的进程的链表

在 2.4 版的内核里，查找最佳候选就绪进程的过程是在调度器 schedule() 中进行的，每一次调度都要进行一次，这种查找过程与当前就绪进程的个数相关，因此，查找所耗费的时间是 O(n) 级的，n 是当前就绪进程个数。正因为如此，调度动作的执行时间就和当前系统负载相关，无法给定一个上限，这与实时性的要求相违背。在新的 O(1) 调度中，这一查找过程分解为 n 步，每一步所耗费的时间都是 O(1) 量级的。

在 runqueue 中有两个优先级数组，一个活跃的和一个过期的，它们是 prio_array 类型的结构体，优先级数组是一种能够提供 O(1)级算法复杂度的数据结构，定义如下：

```
struct prio_array {
    unsigned int nr_active;
    DECLARE_BITMAP(bitmap, MAX_PRIO+1); // include 1 bit for delimiter
    struct list_head queue[MAX_PRIO];
};
```

系统拥有的优先级个数由 MAX_PRIO 定义，默认值是 140。优先级数组使可运行处理器的每一种优先级都有一个相应的队列，而这些队列包含对应优先级上的可执行进程链表。优先级数组还拥有一个优先级位图，当需要查找当前系统内拥有最高优先级的可执行进程时，它可以帮助提高效率。优先级位图数组的类型为 unsigned long 长整型，长 32 位，如果每一位代表一个优先级的话，140 个优先级需要 5 个长整型数才能表示。bitmap 含有五个数组项，总共 160 位。开始的时候，所有的位都被置为 0，当某个拥有一定优先级的进程开始准备执行时，位图中相应的位就被置为 1。这样，查找系统中最高优先级就变成了查找位图中被设置的第一个位。因为优先级个数是个定值，查找时间恒定，不受系统的可执行进程数目的影响。优先级数组的示意图如图 3-1 所示。

O(1)级的调度程序为每个处理器维护两个优先级数组：活动数组 active 和过期数组 expired。arrays 二元数组就是两类就绪队列的容器， active 和 expired 分别指向其中一个。活动数组内的可执行队列上的进程都还有时间片剩余；而过期数组

的可执行队列上的进程都耗尽了时间片。当一个进程的时间片耗尽时，它会被迁至过期数组。当活动数组为空时，则表示当前所有进程的时间片都消耗完了，此时，active 和 expired 进行一次对调，重新开始下一轮的时间片递减过程。

下面就是 O(1)级调度的核心：

```
array = rq->active;
if (unlikely(!array->nr_active)) {
    /*
     * Switch the active and expired arrays.
     */
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
}
```

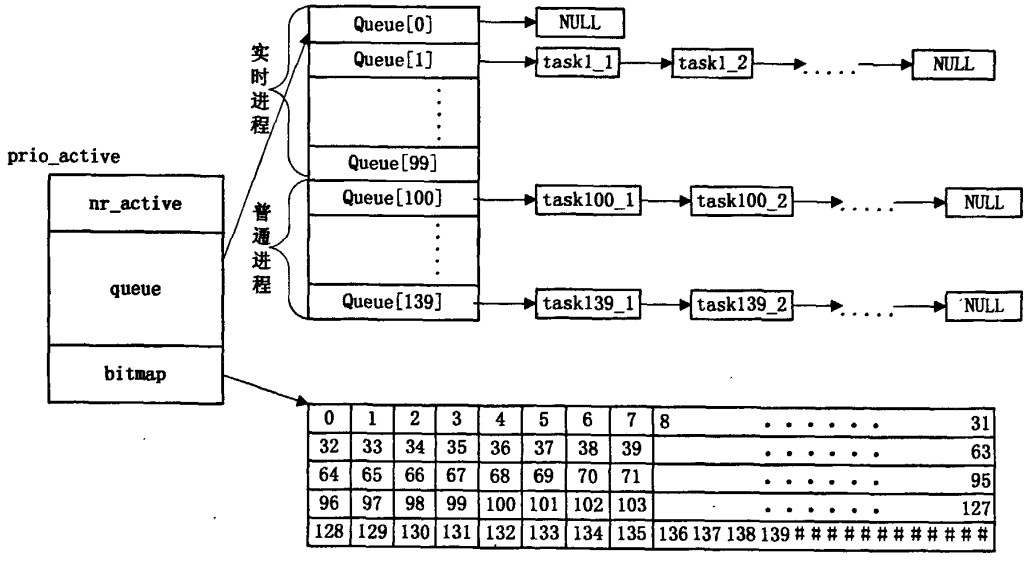


图 3-1 prio_array 结构示意图

3.1.1.2 改进后的 task_struct

Linux 的每个进程都用一个进程信息结构 task_struct 来说明。每个进程的

task_struct 结构都记录了关于进程的信息：调度相关的数据成员，进程信号处理的有关数据成员，进程队列的指针，进程标志数据成员，信号量数据成员，时间定时器数据成员，文件系统相关数据成员和进程虚拟空间管理数据成员等等。表 3-2 列出了 task_struct 结构中与进程调度相关的字段。

1. policy

每个 Linux 进程都是按照下面的调度类型被调度[31]：

SCHED_FIFO：先进先出的实时进程。

SCHED_RR：时间片轮转的实时进程。

SCHED_NORMAL：普通分时进程。

SCHED_BATCH： CPU 密集型进程。2.6.16 版本中添加的新的调度类型。

2. avg_sleep

进程的平均等待时间（以 nanosecond 为单位），在 0 到 NS_MAX_SLEEP_AVG 之间取值，初值为 0，相当于进程等待时间与运行时间的差值。sleep_avg 所代表的含义比较丰富，既可用于评价该进程的"交互程度"，又可用于表示该进程需要运行的紧迫性。这个值是动态优先级计算的关键因子，sleep_avg 越大，计算出来的进程优先级也越高（数值越小）。

进程的 sleep_avg 值是决定进程动态优先级的关键，也是进程交互程度评价的关键，它的设计是 2.6 调度系统中最为复杂的一个环节，可以说，2.6 调度系统的性能改进，很大一部分应该归功于 sleep_avg 的设计。

3. static_prio

每个普通进程都有自己的静态优先级，调度程序使用静态优先级来评估系统中某个进程与其他进程之间调度的程度。内核用从 100（最高优先级）到 139（最低优先级）来表示普通进程的静态优先级。新进程总是继承父进程的静态优先级。nice 值沿用 Linux 的传统，在 -20 到 19 之间变动，数值越大，进程的优先级越小。nice 是用户可维护的，但仅影响非实时进程的优先级。2.6 内核中不再存储 nice 值，而代之以 static_prio。进程初始时间片的大小仅决定于进程的静态优先级，这一点不论是实时进程还是非实时进程都一样，不过实时进程的 static_prio 不参与优先级计算。

nice 与 static_prio 之间的关系如下：

static_prio = MAX_RT_PRIO + nice + 20;

其中，MAX_RT_PRIO 的值定义为 100。

表 3-2 task_struct 中与进程调度相关的字段

类型	名称	说明
unsigned long	thread_info->flag	存 TIF_NEED_RESCHED 标志，如果必须调用调度程序，则设置该标志
unsigned int	thread_info->cpu	可运行进程所在运行队列的 CPU 逻辑号
unsigned long	state	进程的当前状态
int	prio	进程的动态优先级
int	static_prio	进程的静态优先级
struct list_head	run_list	指向进程所属的运行队列链表中的下一个和前一个元素
prio_array_t	array	指向包含进程的运行队列的集合
unsigned long	sleep_avg	进程的平均睡眠时间
unsigned long long	timestamp	进程最近插入运行队列的时间，或涉及本进程的最近一次进程切换的时间
unsigned long long	last_ran	最近一次替换本进程的进程切换时间
int	activated	进程被唤醒时所使用的条件代码
unsigned long	policy	进程的调度类型
cpumark_t	cpus_allowed	能执行进程的CPU的位掩码
unsigned int	time_slice	在进程的时间片中还剩余的时钟节拍数
unsigned int	first_time_slice	如果进程肯定用不完其时间片，就把该标志设为 1
unsigned long	rt_priority	进程的实时优先级

4. prio

普通进程除了静态优先级还有动态优先级。其值范围是 100（最高优先级）到 139（最低优先级）。动态优先级是调度程序在选择新进程来运行的时候使用的数。它与静态优先级的关系可用下面的公式来表示：

```
动态优先级 = max(100, min(静态优先级-bouns+5, 139))
```

(3-1)

bouns 的范围从 0 到 10，小于 5 表示降低动态优先级以示惩罚，大于 5 表示提高动态优先级以示奖赏。bouns 值与进程的平均睡眠时间有关。

动态优先级的计算主要由 effect_prio() 函数完成，从该函数中还可以看到非实时进程的优先级仅决定于静态优先级（static_prio）和进程的 sleep_avg 值两个因素，而实时进程的优先级实际上是在 setscheduler() 中设置的，且一经设定就不再改变。

5. rt_priority

每个实时进程都与一个实时优先级相关，实时优先级是一个从 1（最高优先级）到 99（最低优先级）的值。与普通进程相反，实时进程总是被当成活动进程。

6. time_slice

静态优先级决定了进程的基本时间片，它们之间的关系可以用下面的公式来确定：

```
基本时间片   = { (140-静态优先级) *20    若静态优先级<120
(单位为ms)     (140-静态优先级)*5      若静态优先级>=120
```

(3-2)

进程的 time_slice 值代表进程的运行时间片剩余大小，在进程创建时与父进程平分时间片，在运行过程中递减，一旦归 0，则按 static_prio 值根据公式(3-2)重新赋予上述基准值，并请求调度。时间片的递减和重置在时钟中断中进行（sched_tick()）。

3.1.2 调度流程

调度器的基本流程仍然可以概括为下面几个步骤：

1. 前期工作

2.6 内核实现了抢占运行，没有锁保护的任何代码段都有可能被中断。因此，调度器一开始就禁用内核抢占。然后保证 prev 不占用大内核锁。

2. 清理当前运行中的进程（prev）

关掉本地中断，并获得要保护的运行队列的自旋锁，然后计算 prev 所使用的 CPU 时间片长度并检查 prev 的状态。如果它没有在内核态被抢占就应该从运行队列删除。如果它是非阻塞挂起信号，且状态为 TASK_INTERRUPIBLE，就把该进程的状态设为 TASK_RUNNING，并插入运行队列。

3. 选择下一个投入运行的进程（next）

检查运行队列中剩余的可运行进程数 nr_running。如果为 0，就调用 idle_balance()函数从其他 CPU 上迁移进程到本地 CPU。迁移失败就运行 idle 进程。如果不为 0，就检查 active 队列的活动进程数 nr_active。如果为 0，就交换 active 和 expired 指针内容。现在可以在活动的优先级数组中查找一个优先级最高的可运行进程了。

由上述可知，2.6 的调度器对候选进程的定位有三种可能：

1) active 就绪队列中优先级最高且等待时间最久的进程；
2) 当前 runqueue 中没有就绪进程了，则启动负载平衡从别的 cpu 上转移进程，再进行挑选。
3) 如果仍然没有就绪进程，则将本 cpu 的 idle 进程设为候选。

4. 设置新进程的运行环境

内核访问 next 进程的 thread_info 数据结构，prefetch 宏提示 CPU 控制单元将 next 的进程描述符第一部分字段内容装入硬件高速缓存。

prev 进程的 sleep_avg 减去其运行时间； timestamp 更新为当前时间，记录被切换下去的时间，用于计算该进程等待时间。

5. 执行进程上下文切换

如果 prev 和 next 是不同的进程，调用 context_switch ()函数建立 next 的地址空间，调用 switch_to()执行 prev 和 next 之间的进程切换。

6. 后期整理

如果 prev 是一个内核线程，mmdrop()函数就减少内存描述符的使用计数器，如果该计数器等于 0，还要释放与页表相关的所有描述符虚拟存储区。

释放运行队列自旋锁，打开本地中断。在需要的时候重获大内核锁，重新启用内核抢占。

3.2 Linux 对 SMP 的支持

对称多处理结构（Symmetric Multi-Processor Architecture，SMP）中，所有的CPU 在运行时都是“对称”的，或者说是“平等”的，没有主次之分[32]。所有的CPU 通过同一条总线共享同一个内存以及所有的外设。为了减少访问内存的冲突，SMP 结构中的各个CPU 通常都有自己的高速缓存，它们之间通过特殊的硬件通信。相对应 UP（单处理器），Linux 操作系统内核对 SMP 系统有额外的代码支持，内核中的 SMP 代码部分被单独编译[34]。

在单 CPU 系统中，任一时刻都只有一个进程在运行，其他进程都不运行。可是在 SMP 架构的系统中，可以有多个进程同时运行，所以 Linux 在进程描述符 task_struct 结构中增加了下面的字段：

```
#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
    int oncpu;
#endif
#endif
```

oncpu 字段用来表示该进程是否正在运行，值为 1 表示进程正在某个 CPU 上运行，值为 0 表示进程不在运行。一个进程只有在 oncpu 字段为 0 时才可以接受调度以投入运行或者被迁移到其他 CPU 上。在 Linux 支持的 SMP 结构中使用 task_running()函数来判断一个进程是否正在运行：

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

下面讨论内核在多个 CPU 之间是如何安排进程调度问题的。

3.2.1 SMP 系统中的进程调度

为了唤醒一个进程，wake_up_process()函数将调用 try_to_wake_up()函数，后

者通过把进程状态设置为 TASK_RUNNING，并把该进程插入本地 CPU 的运行队列来唤醒或停止的进程。try_to_wake_up()函数的有的部分对 SMP 和 UP 来说都适用，而有的部分仅适用于 SMP。try_to_wake_up()函数调用 active_task()函数，将进程移到运行队列并改变优先级。而 active_task()函数调用 recalc_task_prio()函数跟新进程的平均睡眠时间和动态优先级。如果被唤醒的进程优先级比当前正在运行的进程的优先级高，就设置 need_resched 标志。通常哪段代码促使条件达成，它就负责随后调用 wake_up_process()函数。图 3-2，图 3-3 和图 3-4 分别是这些函数的流程图。

在 Linux 用于 SMP 架构的进程调度中，对每一个被剥夺的非 idle 进程，尽力去寻找空闲 CPU 让其恢复运行。需要注意的是，由于在有的多核处理器架构中，L1 Cache 是集成在核内的，即每个核都有自己的 Cache。在这种情况下，如果进程从一个处理器核迁移到另一个处理器核，那么这个过程会带来一定的开销。因此，在 SMP 结构的进程调度中要考虑这些因素[3]。

3.2.2 多处理器系统中运行队列的平衡

为了从多处理器系统获得最佳性能，负载平衡算法应该考虑系统中 CPU 的拓扑结构。从内核 2.6.7 版本开始，Linux 提出一种基于“调度域”概念的复杂运行队列平衡算法[31]。内核周期性地检查运行队列的工作量是否平衡，并在需要的时候把一些进程从一个运行队列迁移到另一个运行队列。

调度域实际上是一个 CPU 集合，它们的工作量应当由内核保持平衡。调度域采取分层的组织形式：最上层的调度域包括多个子调度域，每个调度域包括一个 CPU 子集。每个调度域被依次划分为一个或多个组，每个组代表调度域的一个 CPU 子集，工作量的平衡总是在调度域的组之间来完成的。只有在某个调度域的某个组的总工作量远远低于同一个调度域另一个组的工作量时，才把进程从一个 CPU 迁移到另一个 CPU。在 Linux 中使用结构 sched_domain 来表示调度域，使用结构 sched_group 来表示调度域的组。

每一个 CPU 都有一个“基本”的调度域（struct sched_domain），当然，这是在定义了 CONFIG_SMP 的情况下才有效。这些调度域是通过函数 cpu_sched_domain(i)和宏 this_sched_domain()用来访问的。调度域的层次结构均是通过->parent 指针所指向的基础域而构建的。->parent 指针要保证以 NULL 结束，而且应当为每个 CPU 分配域结构以便于 CPU 的增加。

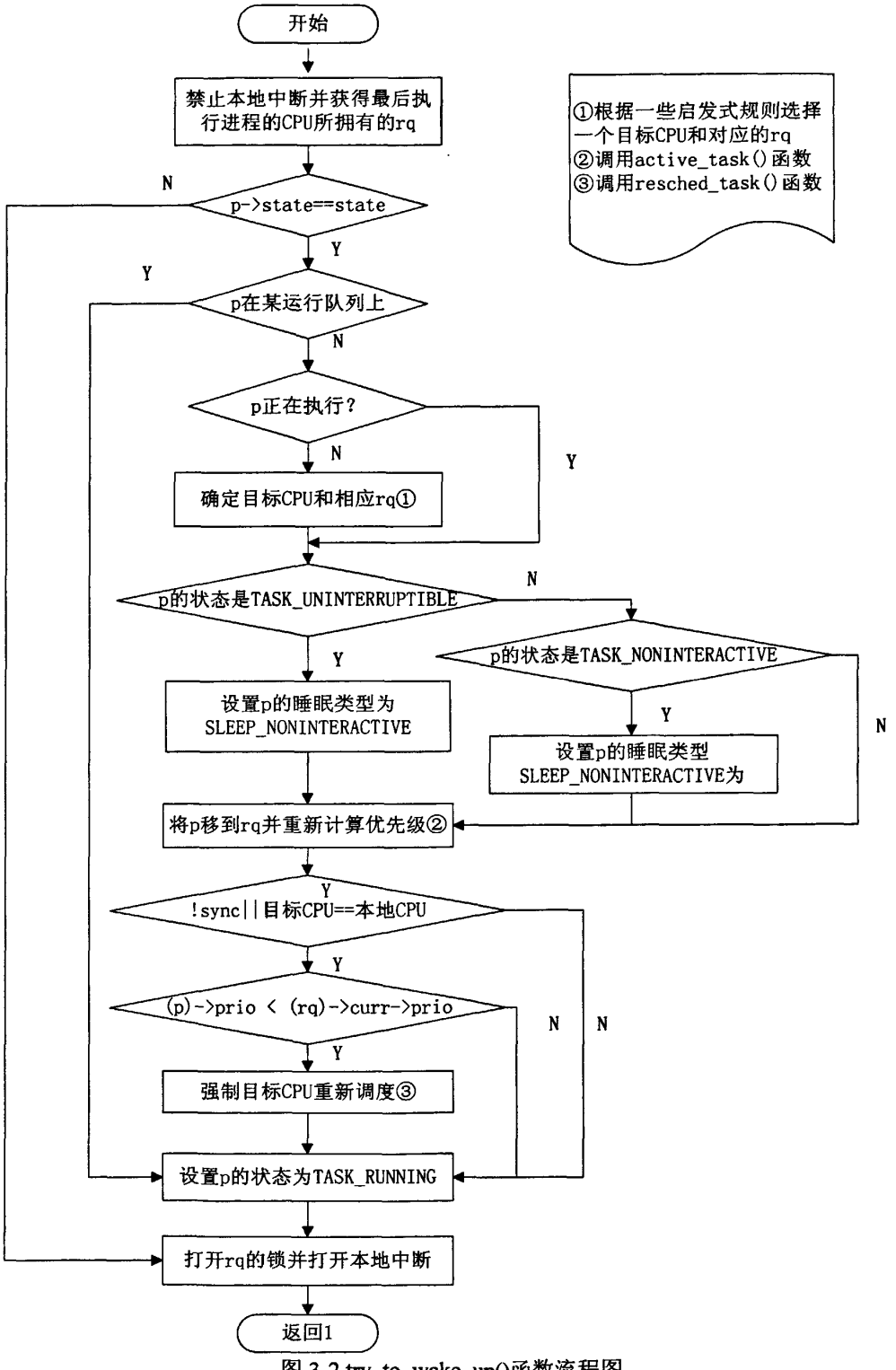


图 3-2 try_to_wake_up()函数流程图

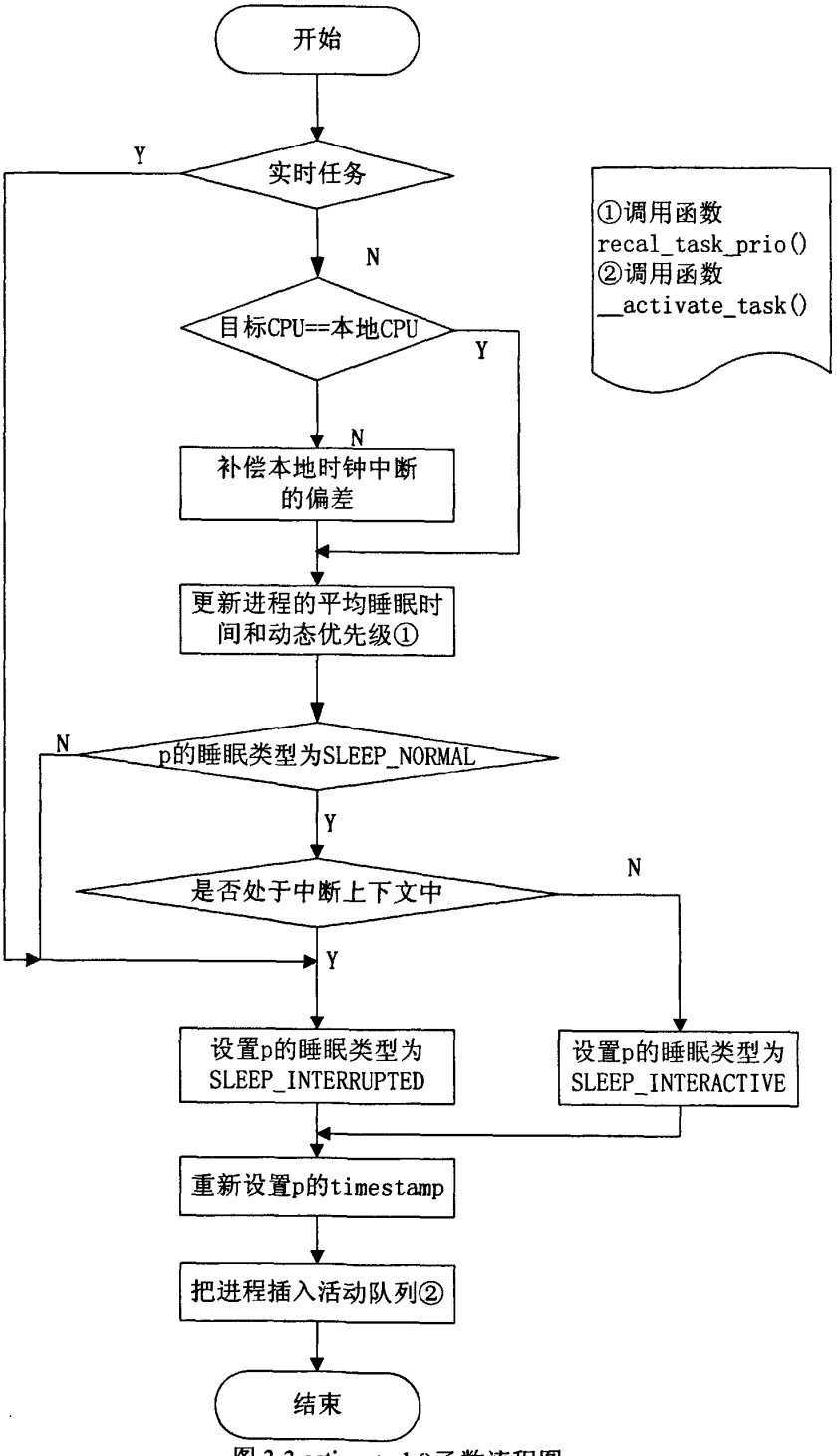


图 3-3 active_task()函数流程图

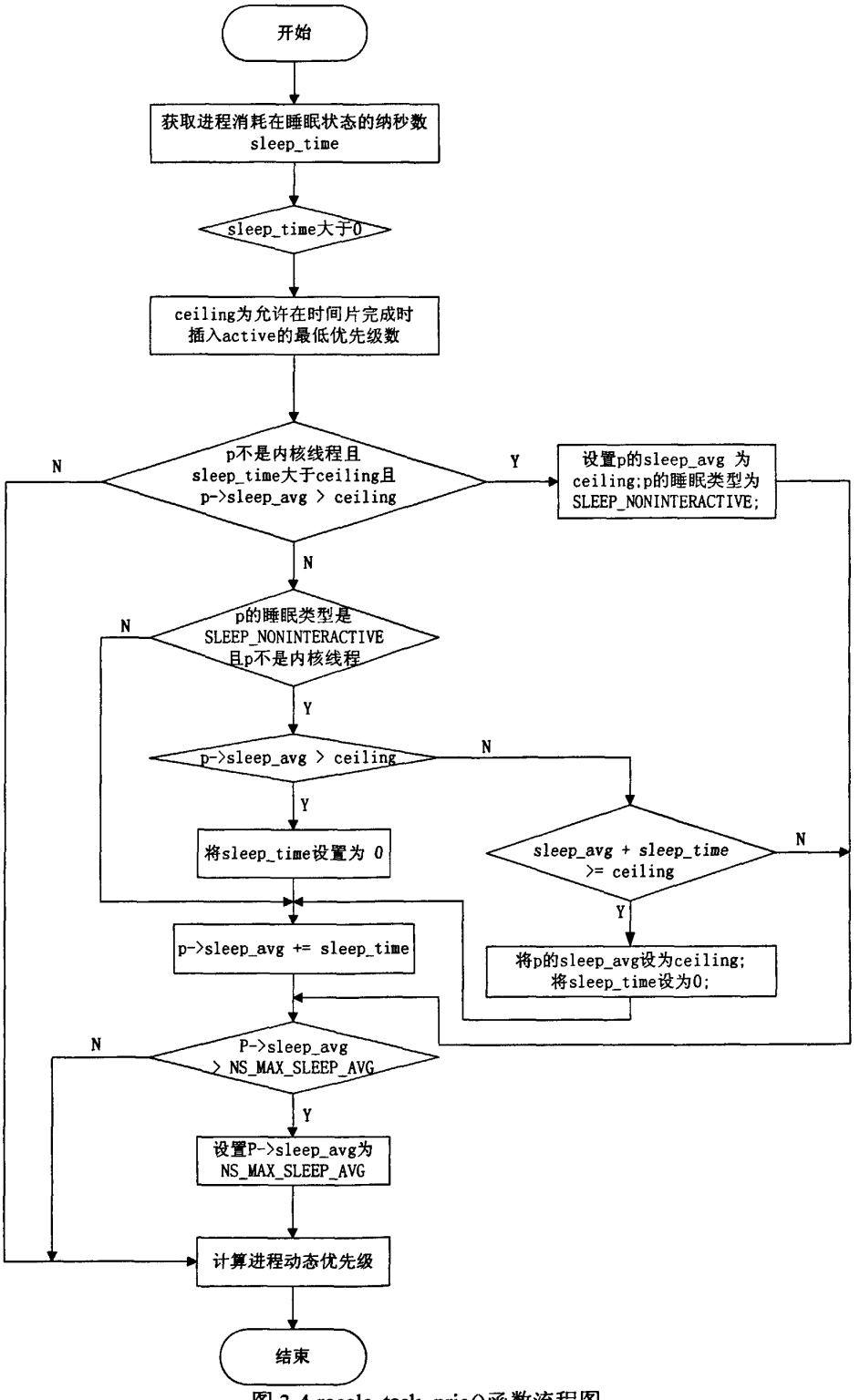


图 3-4 recalc_task_prio()函数流程图

在每一个调度域的范围内都覆盖几个 CPU（存于->span 中）。调度域的范围(span)意思是："在各 CPU 之间达到负载的平衡"。调度域的 span 必须是子调度域 span 的超集，换句话就是，一个调度域的范围必须不小于它的子调度域范围，而且 CPU_i 的基础域(Base Domain)范围至少为 i。位于调度域层次结构最顶层的域的范围将覆盖系统中的所有 CPU，这样做的好处是，一些 CPU 在 cpu_allowed 位掩码没有明确设置的情况下，系统也可以进行任务调度。

每一个调度域都必须有一个或多个 CPU 组（struct sched_group），这些组用->groups 指针链接成一个单向环形链表。而这些 CPU 组的 cpumask 在调度域范围内都必须是相同的。任意两个组的 cpumask 交集必须为空集（即不相同）。->groups 指针所指向的组必须包含这个域所拥有的 CPU。而当某些组被建立后，它们可能因为包含一些只读数据而被其他 CPU 所共享。

每一个调度域的平衡单位是组，即调度域的平衡操作在组与组之间进行。这个时候，每一个组将被当成一个整体。一个组的负载定义为该组内所有 CPU 的负载总和。当且仅当一个组的负载超出平衡的时候才会发生任务迁移。"基础"域的"范围"将构成域层次结构的第一级。对 SMT 而言，基本调度域涵盖所有的物理 CPU，而每个组包含一个虚拟 CPU[35]。

在 kernel/sched.c 中，run_rebalance_domains()函数(以前使用的是 rebalance_tick()函数)在每一个 CPU 上周期性的运行。这个函数在 CPU 的基础调度域内检查是否到了重新平衡的时间间隔，如果是，则执行该调度域的 load_balance()函数。load_balance()函数检查本地 CPU 所在调度域是否处于严重不平衡状态，如果不平衡就调用 move_task()函数实现迁移。而 run_rebalance_domains()函数将接着检查父调度域（如果 parent 不为空），进而是父调度域的父调度域，如此下去。load_balance()函数的流程图如图 3-5 所示。

3.2.3 Linux2.6 的 CPU 亲和力特性（affinity）

CPU 亲和力(affinity)是进程要在某个给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器的倾向性[36]。在 Linux2.6 版本中，引入了一种机制，使开发人员可以编程实现 CPU 亲和力。

在进程描述符 task_struct 结构中，有一个字段与 CPU 亲和力紧密相关—cpu_allowed 位掩码。这个位掩码由 n 个位组成，与系统中 n 个逻辑处理器一一对应。具有 2 个物理 CPU 的系统可以有 2 位；若这些 CPU 都采用超线程技术，那

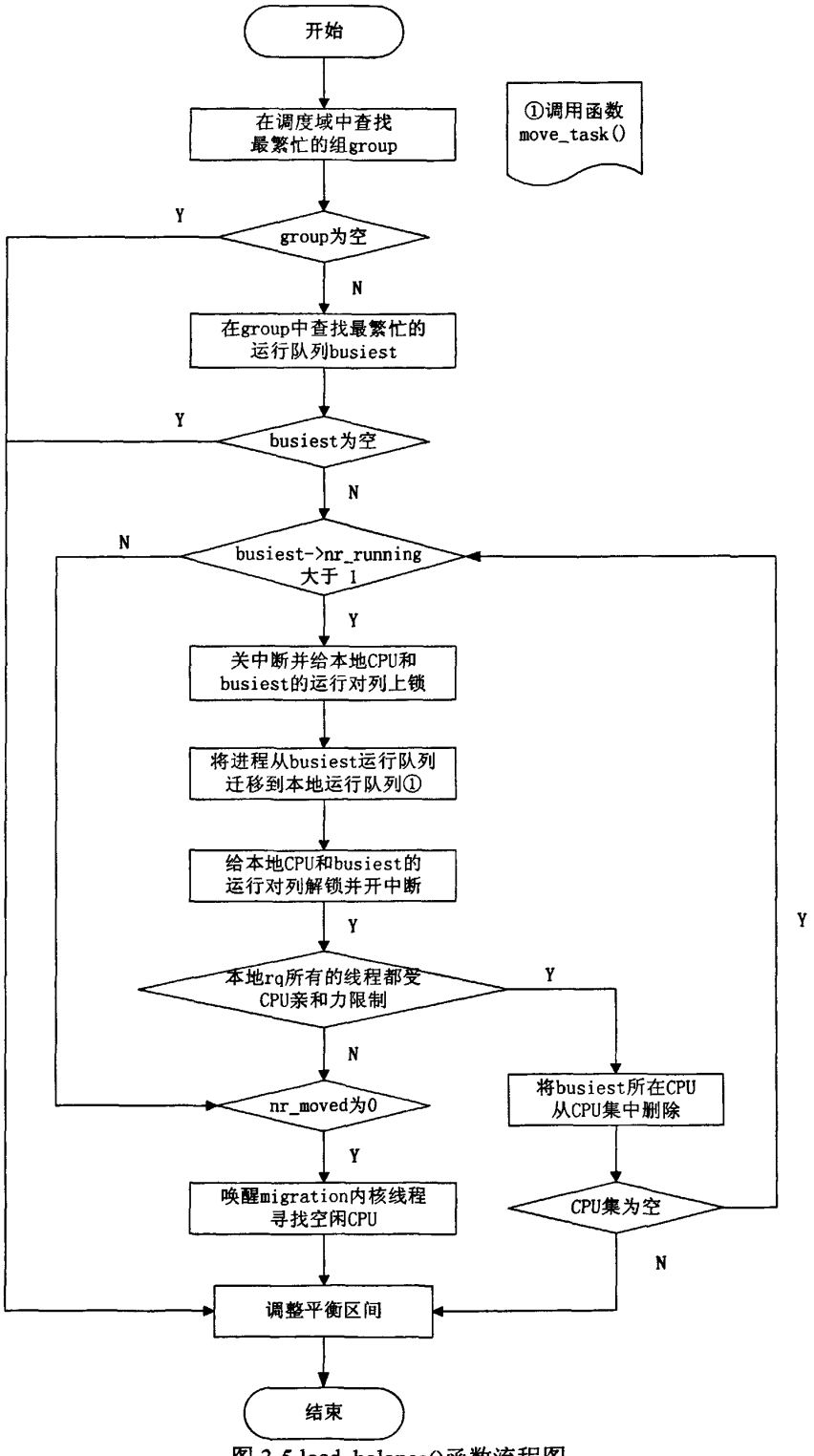


图 3-5 load_balance()函数流程图

么这个系统就有一个 4 位的位掩码。如果为给定的进程设置了给定的位，那么这个进程就可以在相关的 CPU 上运行。因此，如果一个进程可以在任何 CPU 上运行，并且能够根据需要在处理器之间进行迁移，那么位掩码就全是 1。实际上，这就是 Linux 中进程的缺省状态。

Linux 内核 API 提供了一些方法，让用户可以修改位掩码或查看当前的位掩码：

sched_set_affinity() （用来修改位掩码）

sched_get_affinity() （用来查看当前的位掩码）

注意，cpu_affinity 会被传递给子线程，因此应该适当地调用 sched_set_affinity。

如果一个给定的进程迁移到其他地方去了，那么它就失去了利用 CPU 缓存的优势。因此，如果有多个线程都需要相同的数据，那么将这些线程绑定到一个特定的 CPU 上是非常有意义的，这样就确保它们可以访问相同的缓存数据（或者至少可以提高缓存的命中率）。否则，这些线程可能会在不同的 CPU 上执行，这样会频繁地使其他缓存项失效。

3.3 Linux 调度机制的不足

Linux2.6 的调度算法在以下几个方面有待改进：

1. 因为在处理器间迁移不同进程的代价是不相同的，所以在迁移进程的时候，应该适当考虑进程的特点。比如，迁移进程的时应考虑进程的大小(这里是指占有内存资源的大小)以及进程的类型（CPU 受限型，还是 IO 受限型）。Linux2.6 内核版本的调度算法在迁移进程的时候，并没有考虑到进程占用内存的大小，迁移大的进程到其他的处理器可能会较严重的影响系统的性能。
2. 调度器给处理器分配进程的时候应该考虑进程的相关性[错误！未找到引用源。]。比如，两个进程频繁的通信（通过管道或者共享内存），将它们分配在同一个处理器上工作会更好，因为如果这些进程的共享数据是“高速缓存命中的”，就不用把共享数据从一个处理器的 caehe 里拷贝到另一个处理器的 cache 里，这样就减少了许多不必要的麻烦。
3. 当系统的负载不平衡程度很轻微的时候，不一定需要平衡负载。这样就不会造成进程在 CPU 之间“跳跃”。

4. 从调度策略上看，没有采用在实时任务中最常用的 EDF 等调度策略。当然，这与它是一种通用的强调公平性的操作系统有关。

3.4 本章小结

本章首先概括了 Linux2.6 的 O(1)调度器的特性和优点，然后具体介绍了 O(1)调度器中两个最重要的数据结构：struct runqueue 和 struct tast_struct，讨论了调度策略、进程时间片和优先级的计算，详细分析了调度器的工作流程和 Linux 对 SMP 的支持，包括多处理器负载均衡和 CPU 的亲和力，介绍了相关的主要接口函数和功能。最后，指出 Linux2.6 的 O(1)调度器的不足。本章的分析为下面的讨论打下了基础。

第四章 基于资源利用率的负载均衡系统

4.1 负载均衡

负载平衡调度的一个重要目标就是缩短任务的平均响应时间，提高系统性能。负载平衡的另一个重要目标是均匀地、充分地利用整个系统的资源。这两个目标是一致的,任务的响应时间依赖于其所运行的系统上的负载，资源的使用越平衡,任务的响应时间就越短。负载平衡调度算法需要考虑并解决下列问题：何时启动负载平衡调度？选择哪些任务进行调度负载平衡？转移多少任务？调度的源节点和目标节点是哪些[37]？

负载不平衡是导致进程空闲等待的一个重要因素。在设计调度算法的时候要充分考虑负载平衡问题，必要时使用动态负载平衡技术，即根据各进程计算完成的情况动态的分配或调整各进程的计算任务。动态调整负载时要考虑负载调整的开销以及由于负载不平衡引起的空闲等待对性能的影响，寻找最优负载调整方案。

前人通过对负载平衡系统的大量研究得出下面的结论[39]：

1) 实现了负载均衡的系统比不实现负载均衡的系统性能要好很多。
2) 状态依赖的负载均衡策略比概率论策略性能好很多，但是开销也更大。
3) 概率论策略有时对系统的动态改变不敏感，可能会导致不理想的性能。
4) 考虑源端状态的负载均衡决策提高性能的可能性不如考虑目的端状态的负载均衡决策大[40]。
5) 过多的状态信息不一定能提高负载均衡效率，反而可能降低系统性能[41]。
6) 在状态依赖决策中的状态信息必须是现成的。决策基于过期的或者不准确的状态信息可能会降低系统性能。

4.2 理论分析

负载平衡问题是一个经典的组合优化难题之一，其难度与 Hamilton 问题相当，是一个 NP 完全问题。负载平衡问题可分为静态负载平衡和动态负载平衡[42]。人们通过对静态任务划分的研究，间接地实现了静态负载平衡。解决静态负载平衡的

方法主要有：启发式算法，遗传算法，模拟退火算法，基于图论的方法等。对于动态负载平衡，解决方法主要有以下几种：根据负载变化而基于梯度模型的调度算法，基于随机选择任务移动节点的概率调度算法，自适应的近邻契约算法等[43]。

影响负载平衡的调度问题因素很多，Kim[44]提出的调度模型将负载平衡调度过程分为以下四个阶段：负载估算、负载平衡收益性决策、任务选择和任务转移。文献[43]通过深入研究负载均衡算法，给出了一个分布式系统的负载平衡调度问题的一般模型的形式化描述，用一个四元组来表示负载调度的模型，分别为：分布式系统的网络环境，用户提交的任务集，系统的负载评价，系统所采用的调度策略。文献[45]的调度模型考虑了以下因素：调度时间、调度的源结点和目标结点、调度任务的选择。

4.2.1 负载均衡调度模型

在深入研究各种负载平衡调度算法和调度模型的基础上，综合考虑影响负载平衡调度问题的各个因素，可以将负载平衡调度模型用四元组<E，T，L，S>来表示。其中：

因子 E(Environment)：表示系统的环境；

因子 T(Task)：表示用户提交的任务属性；

因子 L(Load)：表示系统的负载评价；

因子 S(Strategy)：表示系统所采用的调度策略。

4.2.1.1 Environment（SE，SD，SG）

定义 1 系统环境因子 E 是一个三元组（SE，SD，SG）。其中：

系统环境(System Environment)：为了从多处理器系统获得最佳性能，负载平衡算法应该考虑系统中 CPU 的拓扑结构；

负载平衡调度域 SD(Scheduling Domain)：表示负载平衡调度节点的逻辑拓扑结构；

调度组 SG(Scheduling Group)：参与某项调度活动的 CPU 集合。

1. 系统环境 SE

SE 需要描述各节点的性能特性和节点之间物理上的拓扑结构。不同的拓扑结构对负载平衡调度算法有很大影响。

按照构建多核系统的处理器是否相同，可以分为同构多处理器系统和异构多处理器系统。同构多处理器系统大多由通用的处理器组成，多个处理器执行相同

或者类似的任务，而异构多处理器系统除含有通用处理器作为控制、通用计算之外，多集成 DSP、ASIC、VLIW 处理器等针对特定的应用提高计算的性能。

多核处理中的 SMP 是含有两个或两个以上的自治处理单元的情形，每个处理单元有同样芯片，同结构的核，运行同样的软件。SMP 可共用内存及缓存粘接部分。SMP 使用同构的核，尽管同构的内核再不必一定是 SMP。若内部是同构的核，但各核运行的不是相同的操作系统，那就变成非对称多处理机(AMP)。AMP 系统也可以用异构的核和共用存储接口或使用分布存储器。任何情况下，同构或异构核都可构成多核处理器，多核处理器都可以按 SMP（对称）或 AMP(非对称)模式运行。

2. 负载平衡调度域 SD

在负载平衡调度过程中，根据节点之间的协作协议、任务性质与功能对应关系以及负载平衡调度策略的差异，多核系统重新构成逻辑意义上的拓扑结构，即负载平衡调度域 SD(Scheduling Domain)。所有负载平衡调度策略均在负载平衡调度环境 SD 上进行讨论。

3. 调度组 SG

负载平衡调度活动（即执行一次负载平衡调度算法）称为调度事件 Sched_event。调度组是负载平衡调度活动的基本单位，负责处理具体的调度事件。

调度组中的节点分为两类：一个是启动负载平衡事件的，称为调度服务器；一个是参与到负载平衡事件中来的，称为调度成员。调度服务器和调度成员不是固定的。例如，有 P_1，P_2 ，P_3 ，P_4 四个节点，当系统不均衡时，若 P_1 执行了负载平衡例程，则 P_1 就是调度服务器，P_2 ，P_3 和 P_4 就是调度成员；同样的，若 P_2 执行了负载平衡例程，则 P_1 就是调度服务器，P_1 ，P_3 和 P_4 就是调度成员。

4.2.1.2 Task(TT，TC)

定义 2 用户任务因子 T 是一个二元组(TT，TC)。其中：

任务类型 TT(Task Type)：表示由用户提交的任务的类型；

任务约束 TC(Task Constraints)：表示任务的结构特征与性能要求。

1. 任务类型 TT

用户提交的任务往往是多种多样的，一般是根据不同系统的侧重点进行区分。

2. 任务约束 TC

表示任务的结构特征与性能要求。

4.2.1.3 Load(φ，f，Γ，χ，ξ)

定义 3 负载评价因子 L 是一个五元组(φ，f，Γ，χ，ξ)，其中

负载参数 (Load Parameter) φ：表示影响系统或节点负载的因素；

负载函数(Load Function) f：表示系统或节点的负载大小的计算函数；

负载阈值 (Load Threshold) Γ：表示系统或节点的负载阈值；

负载状态(Load State) χ：表示系统或节点的负载状态；

负载状态修正因子ξ(Load State modifying factor)：表示调节系统或节点的负载状态的因子。

1. 负载参数φ

负载平衡调度的一个重要目标就是提供系统性能，缩短平均响应时间。由于负载越重，运行时间越长，因此，理想的负载指标与响应时间有密切关系。负载是对一个在系统上运行的所有任务占用资源的衡量，负载参数是对负载进行量化的评价标准，不同的负载参数定义会得出当前时刻系统的不同负载程度。因此，一个可以正确反映当前系统负载情况的负载指标对动态负载平衡系统来说至关重要。

影响负载的因素非常多，衡量这些因素的负载参数包括：运行队列任务数、系统调用的速率、CPU 进程切换率、CPU 利用率、空闲内存大小、平均响应时间、未完成工作量、可用资源、任务到达数等等。因此，负载参数φ是这些因素的集合，可以表示为：

$$\varphi=\{(\varphi_1,\varphi_2,\varphi_3,\ldots\varphi_n\ldots)\,|\,\varphi_1,\varphi_2,\varphi_3,\ldots\varphi_n\ldots\in R\} \tag{4-1}$$

其中，$\varphi_1,\varphi_2,\varphi_3,\ldots\varphi_n\ldots$ 分别与运行队列任务数、系统调用的速率、CPU 进程切换率、CPU 利用率、空闲内存大小、平均响应时间、未完成工作量、可用资源、任务到达数等等这些因素相对应。

理想的、体现系统负载情况的负载指标应当满足以下条件[46]：

①测量开销低，这意味着可以频繁测量以确保信息最新；

②能体现所有竞争资源上的负载；

③各个负载指标在测量及控制上彼此独立。

2. 负载函数 f

设 R 为实数集，则负载函数如下：

$$f = f(\varphi), \varphi \in R \tag{4-2}$$

若使用运行队列中的任务数 NR 作为唯一负载参数的话，那么该系统的负载函数如下：

$$f = f(\mathrm{N}R), \mathrm{N}R \geq 0 \tag{4-3}$$

3. 负载阈值 Γ

负载阈值 Γ 是一个序偶 $<\beta_1,\beta_2>$，其中 $\beta_1,\beta_2 \in R$，$0<\beta_1 \leq \beta_2$。通过判断当前节点的负载函数 f 随负载参数 φ 变化所得的值与 β_1，β_2 相比较来判断该节点当前的负载状态 χ。

4. 负载状态 χ

负载状态一般有空载(idle)、轻载(light load)、适载(suitable load)和过载(over load)四种，故负载状态集为{空载，轻载，适载，过载}。

由上述可知，负载状态 χ 与负载阈值 Γ 关系紧密。根据负载阈值 Γ 的两个元素 β_1 和 β_2 在来负载函数 f 值域中划分的区间来判断当前节点的负载状态：

$$\chi = \begin{cases} 空载, f(\varphi) = 0 \\ 轻载, f(\varphi) \in (0,\ \beta_1] \\ 适载, f(\varphi) \in (\beta_1,\ \beta_2] \\ 过载, f(\varphi) \in (\beta_2,\ \infty) \end{cases} \tag{4-4}$$

5. 负载状态修正因子 ξ

负载状态修正因子 ξ 用于修正负载阈值 Γ，它可以根据环境负载情况动态调整

负载阈值。当$\xi \neq 0$时，它表示一种自适应负载平衡调度策略。

4.2.1.4 Strategy(ST，SC)

定义4 调度策略S是一个二元组(ST，SC)，其中：

调度类型ST(Scheduling Type)：表示负载平衡调度策略的类型。

调度约束SC(Scheduling Constraint)：表示调度策略的约束条件。

1. 调度类型ST

负载平衡调度的类型通常分为两类：静态调度和动态调度[47]。

①静态调度是根据系统的先验知识做出决策，将任务均匀地分配给各个节点，每个节点的任务数都是定值，运行时负载不能重新分配，不随时间变化。这种方法的优点是可以得到最优负载均衡结果，缺点是在现实环境中很难做到。

②动态调度通过收集并分析系统的实时负载信息，动态地将不平衡节点上的任务迁移到其他合适的节点，在执行过程中调整负载分布的不均衡性。它具有超过静态算法的执行潜力，能够适应系统负载变化情况，比静态算法更灵活、有效，但是由于必须收集、存储并分析状态信息，因此动态算法会产生比静态算法更多的系统开销，不过这种开销和付出常是有所回报的[48]。

动态负载均衡算法的组成包括四个部分[49]：

①信息策略负责收集整个系统的状态信息。

②转移策略决定节点是否处于适合参加任务转移的状态，即判断某节点是任务发送者还是接收者。

③选择策略决定哪一个任务应该转移。选择一个任务进行转移的基本判别依据是：转移任务的开销比起它的响应时间的减少是划算的。

④定位策略决定把所选择的任务转移到哪个节点上。

2. 调度约束SC

表示调度策略的约束条件。

4.2.2 Linux的负载平衡调度模型

4.2.2.1 Environment（SE，SD，SG）

1. 系统环境SE

Linux一直坚持采用对称多处理模型，这意味着，与其他CPU相比，内核不应该偏向任何一个CPU。多处理机器与很多不同的风格，而且调度程序的实现随

硬件特征的不同而有所不同。Linux 目前比较关注的是下面 3 中不同类型的多处理机器：

标准的多处理器体系结构。这些机器所共有的 RAM 芯片集被所有 CPU 共享。

超线程。超线程芯片是一个同时执行几个线程的微处理器，它包括内部寄存器的拷贝，并快速在它们之间切换。一个超线程的物理 CPU 可以被 Linux 看作几个不同的逻辑 CPU。

NUMA。把 CPU 和 RAM 以本地“节点”为单位分组(通常一个节点包括一个 CPU 和几个 RAM 芯片)。在 NUMA 体系结构中，当 CPU 访问与它同在一个节点中的“本地”RAM 芯片时，速度很快；访问其他节点的 RAM 芯片就非常慢。

2. 负载平衡调度域 SD

在 Linux 中，调度域采取分层的组织形式。最上层的调度域包括多个子调度域，每个调度域包括一个 CPU 子集。

3. 调度组 SG

每个调度域被依次划分为一个或多个组，每个组代表调度域的一个 CPU 子集。调度组是负载平衡调度活动的基本单位，负责处理具体的调度事件，因此工作量的平衡总是在调度域的组之间来完成的。

4.2.2.2 Task(TT，TC)

任务类型 TT：Linux 根据 policy 从整体上区分实时进程和普通进程。

任务约束 TC：实时进程和普通进程的优先级是不同的，实时进程总是先于普通进程运行。

4.2.2.3 Load(φ，f，Γ，χ，ξ)

1. 负载参数φ

Linux 采用 CPU 运行队列长度即进程数目(nr_running)和运行队列的 raw_weight_load 这两个指标作为负载参数。设 rq 是指向某 CPU 上的运行队列的指针，p 指向该运行队列的第一个任务的执行，则

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

每个 p->load_weight 是通过下面的方法来计算的：

若 p 是内核迁移线程则

p->load_weight = 0;

若 p 是实时任务则

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

若 p 是普通任务则

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

可见，任务的 load_weight 与任务的优先级密切相关。

2. 负载函数 f

Linux 每个调度域的平衡单位是组，当且仅当一个组的负载超出平衡才会发生迁移，故负载函数的定义并不依赖单个运行队列长度，而是依赖于整个组所有运行队列的 nr_running 之和与 raw_weight_load 之和，设 n 是组拥有的 CPU 数目，则负载函数 f 为：

$$f(\text{nr\_running},\text{raw\_weight\_load}) = \{(\text{sum\_nr\_running}, \text{sum\_weighted\_load}) \mid \text{sum\_nr\_running}=\sum_{i=1}^{n}\text{nr\_running},\ \text{sum\_weighted\_load}=\sum_{i=1}^{n}\text{raw\_weight\_load}\} \tag{4-5}$$

3. 负载阈值 Γ

在每个 tick 处理函数 scheduler_tick()会调用 update_load()函数来更新当前运行队列的负载，便于在 CPU 平衡调度时选取最忙的 CPU。Linux 使用函数 find_busiest_group()选取调度域中最忙的 CPU。

1) sum_weighted_load 的上限：

avg_load = (SCHED_LOAD_SCALE * total_load) / total_pwr;

其中，total_load 和 total_pwr 由下面示例代码得出：

```
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 的上限：

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

在 sched_group 结构中，cpu_power 字段表明了调度组的容量，在同一调度域中不同组之间分配负载时使用。典型地，在一个调度域中所有组的 cpu_power 都是一样的，只有在非对称的拓扑结构中才会不一样。cpu_power 等于 SCHED_LOAD_SCALE 乘以某个数的积，这个倍数代表了该组在本调度域中还存在其他空载或轻载的组的情况下可以处理的任务的最大数。

4. 负载状态 χ

Linux 的负载状态分为空载，轻载，适载，过载。

5. 负载状态修正因子 ξ

Linux 负载状态修正因子 ξ 不是一个固定值，而是根据实际负载情况动态调整的。

4.2.2.4 Strategy(ST, SC)

1. 调度类型 ST

从内核 Linux2.6.7 版本，Linux 提出一种基于“调度域”概念的复杂的队列平衡算法，这是一种动态负载平衡算法。只有在某调度域的某个组的总工作量远远低于同一个调度域的另一个组的工作量时，才把进程从一个 CPU 迁移到另一个 CPU。Linux 启动负载平衡事件的时机有：

1) 当一个 CPU 将要空闲的时候，schedule()函数就调用 idle_balance()函数将其他 CPU 上的任务“拉”过来。
2) 无论当前 cpu 是否繁忙或空闲，时钟中断（通过 run_rebalance_domains()

函数）每隔一段时间都会检查每个调度域是否平衡，如果不平衡就启动一次 load_balance() 函数平衡负载。

load_balance() 函数有两种调用方式，分别用于当前 CPU 不空闲和空闲两种状态。一般选择当前调度组(sched group)中第一个 idleCPU 或者第一个 busiest CPU 来执行当前调度域或者上层调度域的负载均衡。

Linux 的选择策略：

1) 在调度域中，寻找最繁忙的组；
2) 如果存在最繁忙的组，就在该组中寻找最繁忙的运行队列；
3) 如果存在最繁忙的运行队列，首先扫描过期队列优先级最高的任务；如果过期队列为空，则扫描活动队列；
4) 对所有候选进程进行筛选：
   I. 进程当前没有在 CPU 上执行。
   II. 进程设置的 cpu_allowed 字段允许迁移到某些 CPU。
   III. 被迁移的进程不是“高速缓存命中”的。

Linux 的定位策略：

启动负载均衡操作的 CPU 确定自己所在调度域处于负载均衡状态的时候，将其他过载 CPU 的多余负载迁移到自己的运行队列上。如果内核支持超线程技术，则本地物理芯片中的逻辑 CPU 必须空闲。

2. 调度约束 SC

在 Linux 中，由于引入了 CPU 亲和力(affinity)，用户可以设定自己的任务到某个 CPU 上执行，因此在定位的时候要综合考虑任务的进程描述符的 cpu_allowed 字段，即目标 CPU 要在的 cpu_allowed 字段中置位。

4.2.3 负载平衡中的几个关键问题

4.2.3.1 负载指标的选择

文献[50]提出了一种基于公平指标的任务调度负载均衡算法，可以有效地提高系统的性能和效率。Kunz 比较了运行队列任务数、系统调用的速率、CPU 进程切换率、CPU 利用率、空闲内存大小和 1 分钟内负载平均值等 6 个单项指标以及它们的“或”及“与”组合，发现其中效果最好的是单项指标中的运行队列任务数，任何其它单项指标或其组合都不比它好[46]。国际上现有的负载平衡系统多使用运行队列任务数作为负载指标。文献[51]的研究表明，一般效果较好的是，将单项指

标中的资源队列长度作为负载指标。文献[46]提出，虽然多数系统使用资源队列长度(即 CPU)队列长度作为负载指标，但是这种负载指标存在不区分进程的性质和大小的缺点，一个进程对资源的占用量及其运行时间可能比几个小进程对资源的占有量综合还要多，但直接使用资源利用率，即 CPU 利用率和 I/O 利用率却更直观、准确，而且使用资源利用率作为负载指标比使用资源队列长度作为负载指标对资源利用率及响应时间的改进更有效。该文献的实验结果还表明，同时考虑了资源利用率和 CPU 队列长度时比单独考虑这两个因素具有更好的性能。现有的负载平衡系统之所以使用 CPU 队列长度作为负载指标，因为它比较容易获得，而且它与响应时间有密切联系。

从提高系统吞吐量和降低响应时间这两方面来考虑，Linux 选择过期队列中优先级最高的任务进行迁移不一定是最好的选择。本文使用 CPU 和内存两种资源作为负载指标，该模型主要适用于 I/O 需求和进程间通信需求不强的应用。

4.2.3.2 迁移进程的选择

Leland 和 Ott[52]VAX750 和 780 上分析了 9.5×10^6 个 UNIX 进程，提出进程的生命周期拥有 UBNE(used-better-than-new-in-expectation)特性。也就是说，进程当前已占用 CPU 的时间(age)越大，它剩余的 CPU 时间也被认为是越大的。

Harchol-Balter 和 Downey [53]通过分析模拟，从大量观测结果中得出下面的结论：

1) 优先迁移“老的”进程，因为这些进程很可能拥有足够长的生命周期，足以补偿迁移带来的开销。
2) 对进程生命周期分布的功能建模为选择合适的迁移进程提供了分析方法，进程当前已占用 CPU 时间(age)，迁移开销，在源端的负载和在目的端的负载都要考虑。

根据结论 1)，本文选择拥有较长的未来运行时间的进程来迁移。原因有二：首先，从进程本身来看，迁移时间对响应时间影响很大。只有进程的未来运行时间能够补偿迁移带来的开销的情况下才选择该进程进行迁移。其次，从源端来看，它迁移一个进程就要增加工作量，只有选择那些在本地运行所花费的开销大于迁移开销的进程。因此，本文选择迁移进程的一个基本要求是：进程的未来运行时间大于迁移开销。

文献[53]提出了进程迁移代价计算公式：

$$c = f + \mu / b \tag{4-6}$$

其中，f 为固定迁移代价,μ 为迁移进程的内存大小,b 为内存传输带宽。

设进程 p 的未来运行时间为 T(p)，则根据公式(4-6)有：

当 T(p)>c，即 T(p)> f+μ / b 时，进程 p 可以被迁移到另一个 CPU 上。

进程 p 的未来运行时间 T(p)如何求？结论 2)提到了进程生命周期分布，那么进程生命周期分布是怎样的？这些问题留到“4.2.3.3 进程生命周期”小节再具体讨论。

理想的迁移进程应当具有最大的资源负载向量模、最小的迁移代价和最长的当前占用 CPU 时间(age)[56]。并且，被迁移的任务当前不在 CPU 上运行，因为迁移一个正在 CPU 上运行的任务带来开销可能远远大于迁移带来的收益。

设进程 p 的资源负载向量模为 R(p)，由公式(4-6)求得迁移代价为 f+μ (p)/ b，当前占用 CPU 时间为 age(p)。

当下面条件成立时，不处于运行状态的进程 p 可以被迁移到其他 CPU 上：

$$\begin{aligned} &\| R(p) \| \times \frac{age(p)}{f+\mu(p)/b} = \\ &\max(\| R(p_1) \| \times \frac{age(p_1)}{f+\mu(p_1)/b}, \ldots, \| R(p_j) \| \times \frac{age(p_j)}{f+\mu(p_j)/b}) \end{aligned} \tag{4-7}$$

其中，j 为进程 p 所在 CPU 上的进程数量。

由上面推导过程，本文得到了选择迁移进程的两个的条件，算法 1(见 4.3.1 算法 1)的选择策略就是基于这两个条件来实现的。

下面要讨论的是算法 2 (见 4.3.2 算法 2)的选择策略使用的迁移进程的判断条件。

基于 Harchol-Balter 和 Downey[53]的工作，可以推导出适合迁移的进程的最小运行时间为：

$$Minimum\_migration\_age = \frac{f+\mu/b}{n-m} \tag{4-8}$$

其中，$f+\mu/b$ 同公式(4-6)，n 是源端 CPU 上进程数目，m 是目的端 CPU 上进程数目。

设进程 p 当前已占用 CPU 时间为 age(p)，由公式(4-8)得：

当 $age(p) > Minimum\_migration\_age$，即 $age(p) > \frac{f+\mu/b}{n-m}$ 时，可以迁移进程 p 到另一个 CPU 上。

由结论 2)可知，一个进程从一个 CPU 迁移到另外一个 CPU 上，势必会对这两个CPU负载造成影响，因此迁移进程可以选择除了要考虑进程当前运行时间(age)还要考虑源 CPU 和目的 CPU 负载。一般考虑迁移对源 CPU 和目的 CPU 负载造成影响最小的进程，即

$$\begin{aligned}&\| R_s - R(s,p) \| + \| R_d + R(s,p,d) \| \\ &= \min(\| R_s - R(s,p_0) \| + \| R_d + R(s,p_0,d) \|,..., \\ &\| R_s - R(s,p_j) \| + \| R_d + R(s,p_j,d) \|)\end{aligned} \tag{4-9}$$

至此，我们得到了两种不同的选择策略，每个选择策略都由两个判断条件组成。

4.2.3.3 进程生命周期

负载平衡的效率依赖于工作负载的特性，包括进程生命周期的分布和到达的进程类型。如何预测进程的生命周期？下面给出两种方案。

1. 进程生命周期的分布

通过对大量进程的生命时间数据的统计和分析，Harchol-Balter[53]采用统计学方法，得出了当进程当前已占用CPU时间(age)T大于1s时进程的生命周期(lifetime)概率分布形式：

$$P\{ \text{Lifetime} > T\} = T^k \tag{4-10}$$

其中，k 的值随不同的系统在- 1.3 到- 0.8 之间变化，通常靠近- 1。基于这个分布，可以使用进程的当前生命时间来预测进程的未来运行时间。

当进程当前已占用 CPU 时间(age)T 小于 1 秒时，该进程再运行(live)T 秒的概率大于 1/2。因此，对生命周期(lifetime)小于 1 秒的进程来说，剩余生命时间的中位数比当前已占用 CPU 时间(age)大。

因此进程未来的运行时间 T 大于或等于用当前进程占用 CPU 时间(age)的可能性很大，即 $T \geq age$ 的可能性很大。我们使用当前进程占用 CPU 时间(age)来预测进程未来的运行时间 T。在“4.3.1.2 迁移进程的选择”小节中，提到“当 T(p)>c，即 T(p)> f + μ / b 时，进程 p 可以被迁移到另一个 CPU 上”，进程 p 的未来运行时间 T(p)可以通过当前进程占用 CPU 时间(age)来预测，那么该迁移条件可以写成：

当 age(p)>c，即 age(p)> f + μ / b 时，进程 p 可以被迁移到另一个 CPU 上。

2. 跟踪预测技术

文献[54]出跟踪预测的方法对任务的执行时间进行估计。

设跟踪预测期(Trace Time)的时间长度为 T_t，磁盘访问速率为 v，任务所需内存空间大小 M，任务所要访问文件的总长 L，任务读写内存数据的总量 D_m，任务要打开的文件名 FN，任务从 FN 中读写数据的总量 D_f，于是有：

访存速率 v_m：

$$v_m = \frac{D_m}{T_t} \tag{4-11}$$

访问文件的速率 v_f：

$$v_f = \frac{D_f}{T_t} \tag{4-12}$$

执行时间 T

$$T = \frac{M}{v_m} + \frac{L}{v_f} \tag{4-13}$$

由上式：

$$T = T_t \times (\frac{M}{D_m} + \frac{L}{D_f}) \tag{4-14}$$

将上述两种方案进行比较，方案 2 的跟踪预测时间难以把握，首先，跟踪预测的时间要保证能够收集足以判断一个任务行为的信息；其次，跟踪预测的时间要尽可能短，这样才能减小系统的开销。因此本文选择方案 1 提出的方法来计算进程的生命周期。

4.2.3.4 核间通信

在“2.3.2CMP 结构”小节中提到，CMP 处理器的各 CPU 核心执行的程序之间有时需要进行数据共享与同步，因此其硬件结构必须支持核间通信。目前比较主流的片上高效通信机制有两种，一种是基于总线共享的 cache 结构，一种是基于片上的互连结构。总线共享 cache 结构是指每个 CPU 内核拥有共享的二级或三级 cache，用于保存比较常用的数据，并通过连接核心的总线进行通信。基于片上互连的结构是指每个 CPU 核心具有独立的处理单元和 cache，各个 CPU 核心通过交

又开关或片上网络等方式连接在一起。各个 CPU 核心间通过消息通信。

除了硬件，软件上也要提供高效通信机制保证。Linux 内核使用了多种同步技术来避免由于多个 CPU 对共享数据的不安全访问导致的数据崩溃[31]。

1. 每 CPU 变量。把内核变量声明为每 CPU 变量是最简单也是最重要的同步技术。每 CPU 变量主要是数据结构的数组，系统的每个 CPU 对应数组的一个元素。一个 CPU 可以随意地读写自己的元素而不用担心出现竞争条件，但是它不应该访问其他 CPU 对应的数组元素。内核抢占可能使每 CPU 变量产生竞争条件，因此内核控制路径在禁用抢占的情况下访问每 CPU 变量。
2. 原子操作。避免由于“读—修改—写”指令引起的竞争条件的最容易的方法就是确保这样的操作在芯片级是原子的。这样的操作必须以单个指令执行，不能中断，且避免其他 CPU 访问同一存储器单元。Linux 内核提供了专门的数据类型 atomic_t 和一些专门的函数和宏，这些函数和宏作用于 atomic_t 类型的变量，并当作原子的汇编语言指令来使用。
3. 自旋锁。自旋锁是用来在多处理器环境中工作的一种特殊的锁。如果内核控制路径发现自旋锁开了，就获取锁继续自己的执行；如果发现锁由运行在其他 CPU 上的内核控制路径锁着，就忙等，即反复执行一条紧凑的循环指令直到锁被释放。即使忙等的内核控制路径无事可作，它也在 CPU 上保持运行。
4. 信号量。Linux 提供了两种信号量：内核信号量，由内核控制路径使用；System V IPC 信号量，由用户态进程使用。在此，我们只讨论内核信号量。当内核控制路径试图获取内核信号量所保护的资源被其他内核控制路径占用时，与前者相应的进程被挂起。只有在资源被释放时，该进程才再次变为可运行状态。
5. 读—拷贝—更新（RCU）。为了保护在多数情况下被多个 CPU 读的数据结构而设计了另一种同步技术，读—拷贝—更新（RCU）。RCU 允许多个读者和写者并发执行。它通过指针而不是锁来访问共享数据结构，其关键思想包括限制 RCP 的范围，如下所述：

1) RCU 只保护被动态分配并通过指针引用的数据结构。
2) 在被 RCU 保护的临界区中，任何内核控制路径都不能睡眠。

4.2.3.5 进程迁移代价

在多核系统中，“进程迁移”是一种在进程生命周期内将它从一个核迁移到另一个核上，并使进程从其“断点”继续运行下去的方法。多核系统中的“进程迁移”与传统的进程迁移是不同的。多核系统的进程迁移基于内核的实现可以充分利用操作系统提供的功能，获取进程和操作系统的状态，因此实现效率较高，提供了很好的透明性。但是，在有的多核处理器架构中，L1 Cache 是集成在核内的，即每个核都有自己的 Cache。在这种情况下，如果进程从一个处理器核迁移到另一个处理器核，那么这个过程会带来一定的开销。

在我们的负载均衡系统中还是继续沿用 Linux2.6 内核已有的函数 can_migrate_task ()来做最后的判断，即候选迁移进程最后都要满足下面的要求：

(1)进程当前没有在 CPU 上执行。

(2)进程设置的 cpu_allowed 字段允许迁移到某些 CPU。

(3)被迁移的进程不是“高速缓存命中”的。

注意，为了叙述的简洁，在下面的算法中不再显示说明，最后确定的被迁移进程默认都满足了上面的要求。

传统的进程迁移的主要工作在于提取进程的状态，以便它在另一台计算机上根据这些状态再生该进程，继续存取它所有资源并运行。一般，进程的状态包括[54]：

1) 进程执行状态：表示当前运行进程的处理器状态，包括内核在上下文切换时保存和恢复信息，如通用寄存器，浮点寄存器，栈指针等。
2) 进程地址空间和内存状态信息：这是进程状态最主要的一部分，包括进程的虚存信息，进程数据信息和堆栈信息等。
3) 进程控制信息：包括进程标识符，进程优先级，父进程标识等。
4) 打开文件信息：进程打开的文件信息包括文件描述符和文件缓冲块。
5) 进程的消息状态：包括进程缓冲的消息和连接的控制信息。

在多核系统中，进程的迁移要简单很多。内核暂停一个进程的执行时，就把几个相关处理器寄存器的内容保存在进程描述符中，这些寄存器包括：程序计数器（PC），栈指针寄存器（SP），通用寄存器，浮点寄存器，包含 CPU 状态信息的处理器控制寄存器，用来跟踪进程对 RAM 访问的内存管理寄存器。当系统要迁移该进程，在指定 CPU 核心上恢复该进程的执行时，内核用进程描述符合适的字段来装载 CPU 寄存器就可以了。

由于在 Linux 中，被迁移的进程都不是正在执行的并且不是“高速缓存命中”

的进程，因此迁移操作只需要下面几句简单的代码就可以完成：

```
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);
```

首先，将被迁移进程 p 从源 CPU 的优先级数组 src_array（可能是活动数组也可能是过期数组）中摘下，并将该优先级数组的 nr_active 以及该优先级数组所在运行队列 src_rq 的 nr_running 减一，然后将进程 p 的 thread_info->cpu 设置为将要被迁移到的目的 CPU，并将目的 CPU 运行队列 this_rq 的 nr_running 增 1，最后将进程 p 插入 this_rq 的优先级数组 this_array（可能是活动数组也可能是过期数组）中，this_array->nr_active 增 1。

上述迁移操作涉及两个 CPU 的运行队列，将一个进程描述符从一个运行队列删除然后再插入另一个运行队列所花费的时间是恒定，因此可以假设其固定开销为 f。

由于大多数多核系统都是共享内存，因此迁移进程的时候不必像传统的进程迁移那样将该进程的用户上下文和系统上下文（系统上下文是否迁移视具体系统而定）都传送到目的节点。另外，由于被迁移进程不是“高速缓存命中的”，因此当它迁移到另一个核上要运行时，虽然不需要从一个 cache 拷贝数据到另一个 cache，但还是需要从内存读取与该进程相关的数据，这个操作的开销可以用进程稳定工作集大小 μ 除以内存带宽 b 来获得。

综上，我们可以得到迁移一个不正在运行、不是“高速缓存命中”的进程总的开销是 $c = f + \mu / b$。这与文献[53]提出的进程迁移代价计算公式是一样的。

4.3 算法设计

本文使用任务的 CPU 利用率和内存利用率作为负载因子，所以每个 CPU 的负载 R 是一个向量（%CPU，%MEM），资源负载向量的各个分量表示不同资源的使用情况，向量的角度能够显示资源使用的平衡状态。

设某个 CPU 上运行队列的长度为 n，n 的值与 CPU 的处理能力成正比。L 是采用指数平滑算法得到的 CPU 队列的长度。M 为可用内存总量，m(i)是进程 i 的稳定工作集大小。

$$\%CPU = \frac{n}{L} \times 100\% \tag{4-15}$$

$$\%MEM = \frac{\sum_{i=1}^{n} m(i)}{M} \times 100\% \tag{4-16}$$

为了衡量迁移进程对源 CPU 的影响，定义迁移进程 i 相对于源 CPU 的资源负载向量 R(s,i)为：

$$R(s,i) = (\frac{1}{L_s}, \frac{m(i)}{M}) \tag{4-17}$$

为了衡量迁移进程对目的 CPU 的影响，定义迁移进程 i 相对应目的 CPU 的资源负载向量 R(s,i, d)为：

$$R(s,i,d) = (\frac{1}{L_d}, \frac{m(i)}{M}) \tag{4-18}$$

通过资源负载向量的向量加减运算，可以方便地获得迁移进程对源 CPU 和目的 CPU 资源使用情况的影响。该负载均衡算法的目标之一是找到 CPU 资源利用率和内存利用率之间的某种关系使系统负载平衡，即找到系统平衡时每个 CPU 负载向量角度的大小。

4.3.1 算法 1

4.3.1.1 转移策略

本文使用普遍认可的阈值策略作为转移策略：当某节点的负载小于阈值β_1时，就将该节点确定为接受者；当某节点的负载大于阈值β_2时，就将该节点确定为发送者。

4.3.1.2 选择策略

算法的具体步骤如下：

第一步 确定适合迁移的进程的集合 $P_{mingration}$。根据“4.2.3.2 迁移进程的选择”，小节的分析，可以用进程的当前生命时间来预测进程的未来运行时间。进程的未来运行时间大于进程迁移代价，因此，

$$P_{migration}=\{p_i \mid p_i \in P_{source} \wedge age(p_i)>f+mem(p_i)/b\} \tag{4-19}$$

若 $P_{migration}=\varnothing$，跳至第五步。

第二步 设源 CPU 过期队列进程集合 $P_{expired\_array}$，优先考虑在 $P_{expired\_array}$ 和 $P_{mingration}$ 的进程作为候选的迁移进程，即

$$P_{expierd}=\{p_i \mid p_i \in P_{migration} \wedge p_i \in P_{expired\_array}\} \tag{4-20}$$

第三步 如果 $P_{expired}=\varnothing$，则 $P=P_{migration}$；如果 $P_{expired}\neq\varnothing$，则 $P=P_{expired}$。

第四步 从集合 P 中选择迁移进程 p_m，p_m 应具有最大的资源负载向量模、最小的迁移代价(迁移代价由公式(4-6)求得)和最长的当前生命时间：

$$p_m=\{p_i \mid p_i \in P \wedge \| R(s,i)\| \times \frac{age(p_i)}{f+mem(p_i)/b}=$$

$$\max(\| R(s,1)\| \times \frac{age(p_1)}{f+mem(p_1)/b},...,\| R(s,j)\| \times \frac{age(p_j)}{f+mem(p_j)/b}), \tag{4-21}$$

其中$j=\mid P \mid\}$

第五步 算法结束。

4.3.1.3 定位策略

文献[56]提出了一种最小 k 子集随机算法 SKR，主要步骤为：当位置策略进行目标节点选择时，首先从所有节点中挑选 k 个负载最少的节点,然后再从这 k(1≤k≤N)个节点中随机选择一个节点作为目标节点。但是这样做的话一次只能减轻一个节点的负载。文献[57]的启发式负载重分配算法中，提出了 LDS,LRS,LHS 三种算法，其中 LDS 算法的目标是使过载 CPU 发送出的消息数目最少，LRS 算法是使轻载 CPU 接收到的消息数数目最少。基于文献[57]和[58]的工作，本文得出负载均衡系统的定位策略。

根据负载状态χ，本文只讨论两类 CPU：一类为过载 CPU，其负载状态χ=过载；一类为轻载 CPU，其负载状态χ=空载或轻载。由于过载 CPU 提供多余的负载，故将其称为提供者(Donor)，记为 D。轻载 CPU 称之为接受者(Receiver)，记为 R，R 中的 CPU 负载较轻，有能力接受新的负载。注意，为了简单起见，本文下面直接用“负载”来代表 CPU 的负载向量的模。

D 提供的多余负载使用$\overrightarrow{W}$表示：

$$\overrightarrow{W}=\{\Delta w_1,\Delta w_2...\Delta w_n\}$$

其中，Δw_i是 D 中第 i 个 CPU 的多余负载，即$\Delta w_i=W_i-\Gamma_w$，Γ_w是负载阈值，若 CPU 的负载W_i超过这个值，该 CPU 的负载过重。由于 Linux 引入了调度域的概念，因此$\overrightarrow{W}_{domain}$表示某个过载的调度域中所有 CPU 的多余负载，而$\overrightarrow{W}_{group}$表示某个调度域中的某一过载的组所有 CPU 的多余负载。

R 能够接受的负载用$\overrightarrow{C}$表示：

$$\overrightarrow{C}=\{\Delta c_1,\Delta c_2...\Delta c_m\}$$

其中，Δc_i是 R 中第 i 个 CPU 在能力范围内还能接受的负载，即$\Delta c_i=\Gamma_w'-W_i$，Γ_w'是负载阈值，若 CPU 的负载W_i小于这个值，该 CPU 的负载过轻。$\overrightarrow{C}_{domain}$表示某个轻载调度域中所有 CPU 能接受的负载，而$\overrightarrow{C}_{group}$表示一个调度域中某一个轻载的组能接受的负载。

假设$\sum_{i=1}^{m}\Delta w_i=\sum_{j=1}^{n}\Delta c_j$，其中$i\in D,j\in R$。

将$\overrightarrow{W}=\{\Delta w_1,\Delta w_2...\Delta w_m\}$和$\overrightarrow{C}=\{\Delta c_1,\Delta c_2...\Delta c_n\}$中的元素按降序排列，得到：

$W=\{w_1,w_2...w_m\}$，其中$w_1>w_2>...>w_m$

$C=\{c_1,c_2...c_n\}$，其中$c_1>c_2>...>c_n$

假设存在一个负载分配矩阵 A，使得 C=W×A，则 A 为 m×n 阶矩阵：

$$A_{m\times n}=\begin{pmatrix}x_{11} & \cdots & x_{1n}\\ \vdots & \ddots & \vdots\\ x_{m1} & \cdots & x_{mn}\end{pmatrix}$$

$$\sum_{i\in D}w_i(j)x_{ij}=c_j,j\in R$$

$$\sum_{j\in R} w_i(j) = w_i, \forall i \in D$$

$$x_{ij} \in \{0,1\}, i \in D, j \in R$$

当 x_{ij} 为 1 时，发送者 i 与接受者 j 之间发生通信。$\sum_{i\in D} x_{ij}$ 表示第 j 个 CPU 收到的信息；$\sum_{j\in R} x_{ij}$ 表示第 i 个 CPU 发出的信息。$w_i(j)$ 表示第 i 个 CPU 向第 j 个 CPU 发送的负载。

CPU 之间通信的开销是很大的[57]，因此，定位策略的目标是减少 CPU 之间的通信开销，也就是使每个 CPU 发送和接受的消息总数最小。由于负载分配矩阵 A 可以有很多解，这个问题可以抽象为下面的组合优化问题：

$$F=\min\{\max\sum_{i\in D} x_{ij}, \forall j \in R, \max\sum_{j\in R} x_{ij}, \forall i \in D\} \quad (4\text{-}22)$$

若要 F 最小，可以从两方面考虑。使 $\max\sum_{i\in D} x_{ij}, \forall j \in R$, 最小，或者使 $\max\sum_{j\in R} x_{ij}, \forall i \in D$ 最小。Donor 算法的目标是使 $\max\sum_{j\in R} x_{ij}, \forall i \in D$ 最小，即使 W 中的每个 CPU 发出的消息数最小。

Donor 算法描述为：

1) 令 $W = W_{group}, C = C_{group}$。

2) 将 W 与 C 按降序排列。若 w_1=0，跳至 10)。

3) 若 $c_1 = 0$，搜索范围从本调度组 group 扩大到本调度域 sd，若 sd 的多余负载为 W_{sd}，能接受的负载为 C_{sd}。令 $W = W_{sd}, C = C_{sd}$，将 W 与 C 按降序排列。若 $c_1 \neq 0$，跳至 7)。

4) 若 $c_1 = 0$，继续向上层调度域搜索，直到 $c_1 \neq 0$ 或者搜索到了最上

层调度域为止。

5) 如果搜索到了最上层调度域且$c_1=0$，所有 CPU 都处于过载状态，则跳至 10)。

6) 若当前搜索到的调度域的多余负载为 W_{domain}，能接受的负载为 C_{domain}。令$W=W_{domain}, C=C_{domain}$，将 W 与 C 按降序排列。

7) 若 W 中$w_1 \le c_1$，则在 C 中搜索，看是否存在一个$c_j \in C$，使得$c_j = w_1$。若存在，则将w_1 与c_j进行负载均衡。

8) 若不存在c_j使得$c_j = w_1$，则直接将w_1与c_1进行负载均衡，此时，$w_1 = 0$，$c_1 = c_1 - w_1$。将c_1按顺序重新插入 C 中。跳至 7)。

9) 若$w_1 > \max\{c_1, c_2 ... c_n\}$，即$w_1 > c_1$，直接将$w_1$与$c_1$进行负载均衡，此时$c_1 = 0$，$w_1 = w_1 - c_1$。将$w_1$按顺序重新插入 W 中。跳至 7)。

10) 算法结束。

假设 W={33,9,7,6},C={26,20,7,2},使用 Donor 算法的过程如下：

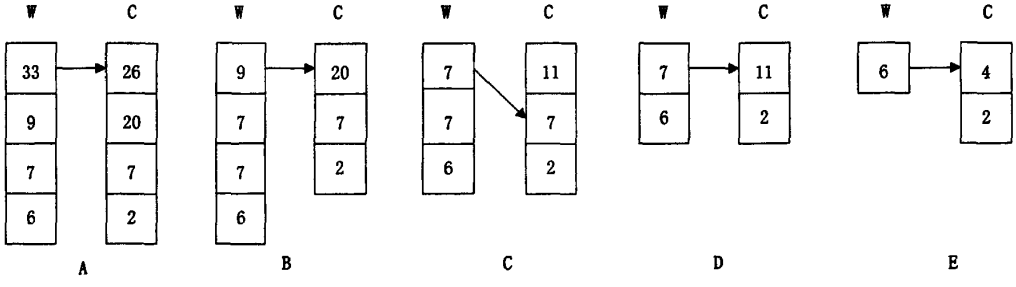


图 4-1 Donor 算法

此时，相应的各 CPU 之间的通信关系可以通过矩阵 B 表示为：

$$B_{4\times4} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

矩阵的行表示负载接受者，即在 C={26，20，7，2}中各负载对应的 CPU；列

表示负载提供者，即在 W={33，9，7，6}中各负载对应的 CPU。负载为 33 的 CPU（列的第一个元素）与负载为 26 的 CPU（行的第一个元素）进行了一次通信，因此矩阵的第一行第一列的值为 1。负载为 9 的 CPU（列的第二个元素）与负载为 20 的 CPU（行的第二个元素）进行了一次通信，因此矩阵的第二行第二列的值为 1……可得到这些 CPU 之间通信的总数为 6 次。在这些 CPU 之间，经过 6 次通信就可以使系统达到平衡。

4.3.2 算法 2

4.3.2.1 转移策略

采用与算法 1 相同的转移策略，即阈值策略。

4.3.2.2 选择策略

可以使用与算法 1 相同的选择策略。也可以使用下面的算法：

第一步 确定适合迁移的进程的集合 $P_{mingration}$。根据“4.2.3.2 迁移进程的选择”，小节的分析，适合迁移的进程集合为：

$$P_{migration}=\{p_i \mid p_i \in P_{source} \wedge age(p_i) > \frac{f+mem(p_i)/b}{n-m}\} \tag{4-23}$$

若 $P_{migration}=\varnothing$，跳至第五步。

第二步 设源 CPU 过期队列进程集合 $P_{expired\_array}$，优先考虑在 $P_{expired\_array}$ 和 $P_{mingration}$ 的进程作为候选的迁移进程，即

$$P_{expierd}=\{p_i \mid p_i \in P_{migration} \wedge p_i \in P_{expired\_array}\}$$

第三步 如果 $P_{expired}=\varnothing$，则 $P=P_{migration}$；如果 $P_{expired}\neq\varnothing$，则 $P=P_{expired}$。

第四步 从集合 P 中选择迁移进程 p_m，迁移进程 p_m 对源节点和目标节点的资源负载向量的影响是最小的：

$$\begin{aligned} p_m &= \{p_i \mid p_i \in P \wedge \| R_s - R(s,i) \| + \| R_d + R(s,i,d) \| \\ &= \min(\| R_s - R(s,0) \| + \| R_d + R(s,0,d) \|, \ldots, \\ &\quad \| R_s - R(s,j) \| + \| R_d + R(s,j,d) \|), \text{其中，} j = \mid P \mid \} \end{aligned} \tag{4-24}$$

第五步 算法结束。

4.3.2.3 定位策略

使 F 最小的另一个方法就是使$\max\sum_{i\in D} x_{ij}, \forall j \in R$,最小。Receiver 算法的目标就是为了实现该目的，即 C 中的每个 CPU 接受的消息数最小。Receiver 算法与 Donor 算法很相似，它们的不同之处只是前者以多余负载接受者为主体来进行负载均衡。

Receiver 算法描述：

1) 令$W = W_{group}, C = C_{group}$。
2) 将 W 与 C 按降序排列。若c_1=0，跳至 10)。
3) 若$w_1 = 0$，搜索范围从本调度组 group 扩大到本调度域 sd，若 sd 的多余负载为 W_{sd}，能接受的负载为 C_{sd}。令$W = W_{sd}, C = C_{sd}$，将 W 与 C 按降序排列。若$w_1 \neq 0$，跳至 7)。
4) 继续向上层调度域搜索，直到$w_1 \neq 0$或者搜索到了最上层调度域为止。
5) 如果搜索到了最上层调度域且$w_1 = 0$，所有 CPU 都处于适载状态，则跳至 10)
6) 若当前搜索到的调度域的多余负载为 W_{domain}，能接受的负载为 C_{domain}。令$W = W_{domain}, C = C_{domain}$，将 W 与 C 按降序排列。
7) 若 C 中$c_1 \leq w_1$则在 W 中搜索，看是否存在一个$w_j \in W$，使得$w_j = c_1$，若存在将w_j与c_1进行负载均衡。
8) 如果不存在w_j使得$w_j = c_1$，则直接将w_1与c_1进行负载均衡，此时，$c_1 = 0$，$w_1 = w_1 - c_1$。将w_1按顺序重新插入 W 中。跳至 7)。
9) 若$c_1 > \max\{w_1, w_2 ... w_n\}$，即$c_1 > w_1$，直接将$c_1$与$w_1$进行负载均衡，此时$w_1 = 0$，$c_1 = c_1 - w_1$。将$c_1$按顺序重新插入 C 中。跳至 7)。

10) 算法结束。

假设 W={34,9,7,7},C={26,22,8,1}，使用 Receiver 算法过程如下：

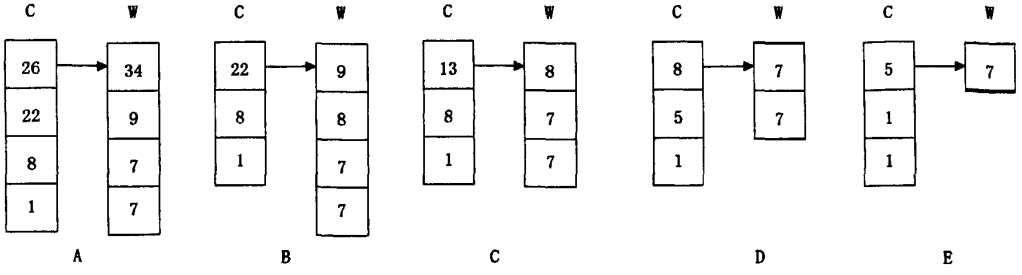


图 4-2 Receiver 算法

此时，相应的各 CPU 之间的通信关系可以通过矩阵 B 表示为：

$$B_{4\times4}=\begin{pmatrix}1&0&0&0\\1&1&0&1\\0&0&1&1\\0&0&0&1\end{pmatrix}$$

矩阵的列表示负载接受者，即在 C={26,22,8,1}中各负载对应的 CPU；行表示负载提供者，即在 W={34,9,7,7}中各负载对应的 CPU。负载为 26 的 CPU（列的第一个元素）与负载为 34 的 CPU（行的第一个元素）进行了一次通信，因此矩阵的第一行第一列的值为 1。负载为 22 的 CPU（列的第二个元素）与负载为 9 的 CPU（行的第二个元素）进行了一次通信，因此矩阵的第二行第二列的值为 1……可得到这些 CPU 之间通信的总数为 7 次。在这些 CPU 之间，经过 7 次通信就可以使系统达到平衡。

4.4 数据结构

1. 进程负载信息表示

```
struct   resource_vector{
unsigned long cpu_usage;   //CPU 利用率
unsigned long mem_usage; //内存利用率
};
```

2. 在进程描述符 task_struct 中添加与负载均衡相关的变量

```
struct resource_vector res_vec;   //存放进程负载信息
```

unsigned long age; //存放进程已占用 CPU 时间

unsigned long overhead; //存放进程迁移开销(估计值)

3. CPU 负载信息表示

struct cpu_load{

spinlock_t lock; //保护该结构的自旋锁

int cpu; //当前 cpu 号

int dest_cpu; //目的 cpu 号

unsigned long delta_work_load; //与负载上限或下限相比的负载差值

struct runqueue *rq; //指向本地运行队列的指针

struct list_head cpu_list; //在负载均衡全局链表中的占位符

};

每个 CPU 都有一个对应的 struct cpu_load 结构，用以记录它们的负载信息以方便系统的负载均衡操作。

4. 负载均衡全局链表

struct list_head * donor; //指向负载提供者链表

struct list_head *receiver; //指向负载接受者链表

这两个链表的元素是 struct cpu_load 结构，根据该结构中 delta_work_load 成员的大小，按从大到小的顺序排列的。一个 CPU 的 struct cpu_load 结构不能同时出现在这两个链表上，因为它不可能既是负载的提供者又是负载的接受者，它要么在 donor 链表上，要么在 receiver 链表上，或者它一个链表都不属于，因为它可以是适载的 CPU。

5. 负载上限 load_upper _threshold 与下限 load_lower_threshold

unsigned long load_upper_threshold; //负载上限

unsigned long load_lower_threshold; //负载下限

定义两个阈值：上限 load_upper _threshold 与下限 load_lower_threshold，使得 load_upper _threshold > load_lower_threshold。对于第 i 个 CPU 上的负载 Res：

(1)如 果 Res>load_upper _threshold，表明第 i 个 CPU 过载，将该 CPU 列人过载集合 D(由 donor 指针指向)中，并将 Res – load_upper _threshold 所得的差值存入该 CPU 对应的 struct cpu_load 结构的 delta_work_load 成员中。

(2) 如 果 Res< load_lower_threshold，表明第 i 个 CPU 轻载，将该 CPU 列人轻载集合 R(由 receiver 指针指向)中，并将 load_lower_threshold – Res 所得差值存入该 CPU 对应的 struct cpu_load 结构的 delta_work_load 成员中。

(3) 如果 load_upper _threshold > Res> load_lower_threshold，表明第 i 个 CPU 负载适中，不需要做负载均衡，这个 CPU 对应的 struct cpu_load 结构就不需要被插入 donor 或者 receiver 链表之中。

6. 改变负载上限与下限的系统调用

set_load_upper_threshold();

set_load_lower_threshold();

负载上限与下限并不是固定不变的，针对系统面向的任务类型，用户可以通过 set_load_upper_threshold()和 set_load_lower_threshold()系统调用来改变负载上限 load_upper _threshold 与下限 load_lower_threshold 的值。

4.5 本章小结

本章首先提出一种负载均衡的通用模型，使用四元组<E，T，L，S>来表示。然后根据该模型的各个因子对 Linux 的负载均衡系统进行剖析，着重分析了 Linux 的负载评价因子 L 和调度策略因子 S。最后通过详细的理论分析，提出了两种不同的算法。这两种负载均衡算法的相同之处在于都采用了资源利用率(CPU 利用率和内存利用率)作为负载因子。不同之处在于，算法 1 的选择策略选出的迁移进程具有最大的资源负载向量模、最小的迁移代价和最长的当前生命时间；而算法 2 选择对源节点和目标节点的资源负载向量的影响是最小的进程进行迁移。算法 1 的定位策略从提供者角度考虑，而算法 2 的定位策略从接受者角度考虑。

第五章 总结与展望

随着对处理器性能要求的不断提高，多核处理器技术越来越受到人们的广泛关注。多核处理器的发展对操作系统提出了新的要求，相应地，操作系统的调度机制也面临新的挑战。传统的操作系统 Linux，FreeBSD,Windows 等目前还不能非常好的在具有几十个核的芯片上运行的。多核处理器的出现对我们来说，既是机遇也是挑战。我们应该抓住这个发展机遇，在系统软件上加大研究力度。同时要利用好开源软件的能力，尤其是 Linux 内核方面的研究与开发，开发出具有核心竞争力的系统软件。

目前国际上对于 CMP 的研究还处于探索阶段，多核操作系统也处在积极研究时期。研究，分析，总结和借鉴支持 CMP 的操作系统的关键技术对于我们认识、理解和设计用于 CMP 的操作系统有着非常重要的意义。通过查阅大量的国内外文献，本文对多核处理器的相关技术及其线程调度技术进行了分析和总结；对 Linux2.6 内核的 O(1)调度算法和负载均衡进行了详细分析;在讨论 Linux 操作系统的一般性原理基础上，针对 Linux 调度器不考虑进程迁移代价的不足，以提高系统的吞吐量减少响应时间为目标，设计了一个基于资源利用率的负载均衡系统。该算法通过计算进程的 CPU 利用率和内存利用率来选择迁移进程，使用 Donor 或 Receiver 算法来定位，具有很好的现实意义和实用价值。该系统在实现上还有很多内容值得我们进一步深入探讨和研究，未来还需要继续进行的工作有：

1. 当前的算法仅仅考虑了 CPU 和内存两种资源，这实际上还不足以体现利用多种资源作为负载指标的平衡算法的效果，下一步将进行更多种类资源情况的研究。

2. 在定位算法中，使用根据降序排列的链表来存储 CPU 负载信息，当查找目标 CPU 时，需要遍历链表上所有结点，这样是很低效的。为了减少时间开销提高查找效率，可以考虑折半查找，或者存储的时候使用哈希表来存储。

3. 当前的算法中得到的多余负载集合与能够接受负载集合中的元素信息可能过时。由于从负载信息的收集到真正负载平衡之间具有时间差，因此元素在开始收集信息时刻是过载的，很可能开始执行负载均衡的时候就已经变成轻载或者空载了，因此要继续深入研究这类问题的解决方案。

4. 同一个任务队列的进程和同一家族的进程尽量映射到同一个处理器上，因为这些进程之间需要频繁通信的可能性是最大的。

致 谢

本文的最后，我衷心地感谢我的导师李毅教授。本课题从选题到调研，再到撰写论文，都是在李老师悉心的指下完成的。他渊博的学识、严谨的治学态度、精益求精的工作态度和诲人不倦的为师风范，让我感受到了一名杰出学者所具有的素质，并深受其影响。在三年的硕士研究生学习阶段，李老师给我提供了优越的科研、学习环境；对我在学习中遇到的困难也尽力的给予指导和帮助。

我要感谢教研室的师兄李一明博士，他在项目开发和本论文撰写的过程给予我很大的帮助。还要感谢教研室的同学陈秋益、曾星科、杨波、何进仙、沈雪峰、董旭，史成伟，我们在相互学习、讨论和争论中度过了愉快的三年时光。也要感谢教研室的师弟师妹，他们使得教研室充满生机而有活力。

我要感谢我的父母，感谢他们的养育之恩，感谢他们对我的一如既往的支持和无私的奉献。

最后，衷心的感谢为评阅本论文而付出辛勤劳动的各位专家和学者。

覃中

二 00 九年三月于电子科技大学

参考文献

[1] 多核系列教材编写组. 多核程序设计[M] . 北京:清华大学出版社,2007.9

[2] Akhter S,Roberts J.Multi-Core Programming—Increasing Preformance through Software Multi-threading[M].Intel Corporation,Intel Press Business Unit,2004.3

[3] 章承科. 基于多核处理器实时操作系统的扩展：[硕士学位论文] . 成都：电子科技大学，2006.6：23-24

[4] 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

[5] Stallings W.Operating Systems:Internals and Design Principles(Fourth Edition)[M]. Prentice—Hall,Inc.,2001：30-48

[6] 金惠芳. 超线程及其实现技术分析[J] . 计算机工程. 2004.12,30(23)：93-95

[7] 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,2006.3：18-19

[8] 屈文新,樊晓桠,张盛兵. 多核多线程处理器存储技术研究进展[J]. 计算机科学,2007,34(4)：13-16

[9] 章隆兵,何立强. 同时多线程结构研究综述 [J] . 中国科学院计算技术研究所内部刊物—信息技术快报,2004,8：1-12

[10] R. Kessler, "The Alpha 21264 Microprocessor", IEEE M*icro*, March-April 1999, pages 24-36

[11] Dual-Core Intel Xeon Processor 7100 Series.

[12] 张盛兵,王晶. 同时多线程结构的线程预构[J] . 西北工业大学学报,2007.4,25(2)：159-162

[13] D.Marr,et al.,Hyper-Threading Technology Architecture and Microarchitecture,Intel Technology Journal, Q1, 2002

[14] Kunle O K, Basem A N, Hammond L, et al.The Case for a Single-chip Multiprocessor[C]//Proc. of the 7th International Conferenceon Architectural Support for Programming Languagesand Operating Systems,New York.1996

[15] Hammond L , Hubbert B A , Siu M , et al . The Stanford Hydra CMP. IEEE Micro , 2000 , 20 (2) :71～84

[16] Inside Intel® Core™ Microarchitecture: Setting New Standards for Energy-Efficient Performance .http://www.intel.com/technology/architecture-silicon/core/index.htm

[17] Kahle J A.Introduction to the Cell Multiprocessor[J].IBM Journa Res,2005,49(4/5):589-604

[18] 颜世云,翁志强. CMT 结构研究综述[J] . 计算机科学,2006,33(7)：196-198

[19] 何军,王飙. 多核处理器的结构设计研究[J] . 计算机工程.2007.8,33(16)：208-210

[20] Kongetira P, Aingaran K , Olukotun K.Niagara : A 32-Way Muhithreaded Sparc Processor[J]. IEEE Micro，2005，25(2)：21-29

[21] 刘近光,梁满贵. 多核多线程处理器的发展及其软件系统架构 [J] . 微处理机,2007,1：1-3

[22] 对称多处理机. http://www.hudong.com/wiki/%E5%AF%B9%E7%A7%B0%E5%A4%9A%E5%A4%84%E7%90%86%E6%9C%BA

[23] Schimmel C.现代体系结构上的 Unix 系统[M] . 张 辉译. 北京:人民邮电出版社,2003.

[24] 王晶,樊晓桠,张盛兵,王海. 多核多线程结构线程调度策略研究[J]. 计算机科学,2007,34(9)：256-258

[25] Snavely A,Mitchell N,Carter L,et al. Explorations in Symbiosis on two Muhithreaded Architectures[C]//In：Workshop on Multithreaded Execution And Compilation (MTEAC)，January 1999：568-572

[26] Snavely A,et al.Symbiotic Jobscheduling for a Simultaneous Multithreading Processor[C]//In：Proceedings of ASPLOS IX，November 2000：234-244

[27] Sanvely A.,Tullsen D.M.,Voelker G. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor[C]//In: Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2002：66-76

[28] Parekh S,Fe. gers S，Levy H.Thread-Sensitive Scheduling for SMT Processors [J].University of Washington,2000：1-18

[29] DeVuyst M.,Kumar R,Dean T. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors[C]//In：IPDPS-2006，Rhodes Island，Greece，April 2006. 10-18

[30] Robert Love. Linux Kernel Development(Second Edition)[M].Pearson Education,Inc.,2005：30-48

[31] Bovet P. D. Cesati M.Understanding the Linux Kernel[M].O'REILLY Media,Inc.,2006：258-290

[32] Linux 2.6 调度系统分析. http://www.ibm.com/developerworks/cn/linux/kernel/l-kn26sch/index.html

[33] 李彬,任国林.Linux 内核基于对称多处理机的实现分析[J]. 计算机技术与发展,2006.1,16(1)：129-131

[34] 高珍,吴永明,周卫华. Linux 操作系统内核对 SMP (对称多处理器) 的支持[J]. 计算机应用研究,2002 ,9：62-63

[35] 初探 Linux2.6 内核—进程调度. http://www.91linux.com/html/article/database/mysql/20070815/5762.html

[36] 管理处理器的亲和力. http://www.ibm.com/developerworks/cn/linux/l-affinity.html#resources

[37] 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

[38] 李冬梅,施海虎,毓清. 基于规则的分层负载平衡调度模型[J] . 计算机科学,2003,30(10)：16-20

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

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

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

[42] 刘振英,方滨兴,胡铭曾等. 一个有效的动态负载平衡方法[J] . 软件学报.2001,12(4)：563-569

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

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

[45] 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[J]. 软件学报,1998,9（1)：25-28

[46] 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

[47] Kunz T. The influence of different workload description on a heuristic load balance scheme[J]. IEEE Trans on Software Engineering. 1991,17(7)：725-730

[48] 王力生,毛昀波. 多处理机系统的负载平衡模型设计[J] . 单片机与嵌入式系统,2008,1：10-13

[49] 胡亮,徐高潮,鞠九滨. 一个基于收益与开销的作业选择策略[J] . 软件学报.1998.4，9(4)：280-283

[50] 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. 计算机工程与设计，2008.3,29(6)：1362-1363

[51] Zhou Songnian,Zhen Xiaohu.Utopia: load sharinig facility for large heterogeneous distributed computer system[J].Software-Practice and Experience,1993,23(12)：1305-1336

[52] Leland W, Ott T. Load balancing huristicsand process behavior[C]//In Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Syst, May 1986.

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

[54] 进程迁移. http://mesopodamia.blogbus.com/logs/37552587.html

[55] Ju Jiubin, Xu Gaochao, YangKun. On-Line Prediction Behaviors of Jobs in Dynamic Load Balance[J].Comput.Sci. Technol. 1996.1,11(1)：39-48

[56] 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,2002.8,30(8)：1148-1152

[57] Pardines,F. F. River.Minimizing the Load Redistribution Cost in Cluster Architectures[C]//In Proceedings of the 12th Euromicro Conference on Parallel,Distributed and Network-Based Processing(EUROMICRO-PDP'04),2004

[58] 王玥,蔡皖东,段琪. 一种自适应动态负载均衡算法[J] . 计算机工程与应用,2006, 21：121-123

攻硕期间取得的研究成果

一、参与项目

2007．3-2008.5 参与教研室和东方电气自动控制工程有限公司（DEA）共同开发的火电自控工程项目—EDA 项目的开发。实现了报表组态子系统，报表运行子系统，单点实时趋势显示控件和画面打印模块,参与系统性能测试并编写用户手册。

二、发表的论文

[1] 覃中,李毅. 一种 Excel 报表运行系统的设计与实现. 科技信息,2009

[2] 覃中,李毅. 基于多核系统的线程调度研究. 研究生学报,2009

基于多核系统的线程调度

作者： 覃中
学位授予单位： 电子科技大学