兰州大学

硕士学位论文

# 一种新的多核体系结构中的线程调度算法及其模拟

姓名：朱文俊

申请学位级别：硕士

专业：数学 计算数学

指导教师：周宇斌

20100501

# 摘　要

随着多核处理器在计算机各个领域的应用，和内含几十甚至上百个核的多核处理器的出现，在操作系统调度算法中考虑这些特性变得越来越迫切。由于与传统多处理器体系结构的不同，如共享的 L2 Cache，内存控制器，和每个核可使用的缓存，在多核处理器上简单地使用 SMP 系统上的调度算法调度任务是不够的。

本文中我们提出了一个模型，将任务分配到各 CPU 和核上，并在 CPU 和核两级调度域上平衡负载，以达到较高的吞吐量。模型中考虑的因素包括运行成本 RC，通信成本 CC 和迁移成本 MC。另外，我们还编写了实现本算法的模拟程序。本模型适合系统中有任意多个 CPU，每个 CPU 有任意多个核的体系结构。

**关键词**: 操作系统；调度；多核；负载平衡。

# Abstract

As multi-core processor begin to emerge in every area of computing, and multi-core processors with tens of hundreds of cores begin to proliferate, operating system scheduling that take the peculiarities of such architectures into account will become mandatory. Due to architectural differences to traditional multi-processors, such as shared L2 cache, memory controllers and smaller cache size available per computational unit, it does not suffice to simply schedule tasks on multi-core processor in the same way as on SMP systems.

To make the best use of the computational power available, it is essential to assign the tasks dynamically to the CPUs and cores, and balance the load after a thread or a task has finished. Here we develop a model to allocate the task to the CPUs and cores and balance the load between the CPUs and cores to maximum the throughput. Running cost(RC),communicating cost(CC) and migrate cost(MC) have been taken into account, besides, simulation programs in Python have been developed for the algorithm on Ubuntu platform. this model is suitable for arbitrary number of CPUs and cores in a CPU.

**Key Words**: Operating System; scheduling; multi-core; load balance.

## 原 创 性 声 明

本人郑重声明：本人所呈交的学位论文，是在导师的指导下独立进行研究所取得的成果。学位论文中凡引用他人已经发表或未发表的成果、数据、观点等，均已明确注明出处。除文中已经注明引用的内容外，不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究成果做出重要贡献的个人和集体，均已在文中以明确方式标明。

本声明的法律责任由本人承担。

论文作者签名：朱文俊　　日期：2010年5月29日

## 关于学位论文使用授权的声明

本人在导师指导下所完成的论文及相关的职务作品，知识产权归属兰州大学。本人完全了解兰州大学有关保存、使用学位论文的规定，同意学校保存或向国家有关部门或机构送交论文的纸质版和电子版，允许论文被查阅和借阅；本人授权兰州大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用任何复制手段保存和汇编本学位论文。本人离校后发表、使用学位论文或与该论文直接相关的学术论文或成果时，第一署名单位仍然为兰州大学。

保密论文在解密后应遵守此规定。

论文作者签名：朱文俊　　导师签名：[illegible]　　日期：2010年5月29日

# 第1章　概论

## 1.1　背景与发展

由于单核处理器性能提高程度有限，多核被引入到了处理器中。Intel、AMD等处理器厂商推出了双核、四核的处理器，而且六核、八核的处理器也即将推出。这一趋势预示着未来会出现内含十几个核甚至上百个核心的处理器，每个核上都可以同时运行多个线程。

与单核处理器相比，多核处理器可以更快地执行任务。通过划分任务，多线程应用程序能够充分利用多个内核，并可在特定的时间内执行更多任务。

多核处理器的出现提出了一种新的体系结构，这种结构类似于原来的多处理器体系结构，如 SMP（Symmetrical Multi-Processing，对称多处理）等，但又有不同于原来体系结构的部分：通过更高的并行度都可以处理更快的处理多任务是相同的，不同之处在于，多核处理器中的核心紧密地耦合在一起，共享 L2 缓存、总线等，并且处理器和核心形成了两级结构，不同于对称多处理结构中所有处理器处于同一级，呈现出平面的结构。这些变化都对操作系统中调度算法提出了新的要求。

微软 Windows 核心操作系统部门的内核架构师 Dave Probert 就表示，如今的计算机并不能从他们的多核处理器中获得足够的性能，处理器制造商不断在他们的每一款新一代处理器中增加内核数量，也许是时候重新考虑现有操作系统的基础架构了。

为了充分利用多核处理器新的特点，从而更快地执行任务，本文中讨论了如何在考虑了运行成本、通信成本和迁移成本的模型中最大化系统的吞吐量，使得系统在给定的时间内执行尽可能多的任务。

在 [3] 中，Fedorova 等人研究了多核体系结构对操作系统的影响，提出：L2 Cache 是多核处理器中关键的共享资源，若 L2 Cache 容量不足，则不能起到 Cache 应有的弥补处理器速度和内存速度差距的作用。在另一篇文章 [2] 中他们展示了怎样优化操作系统的调度器以提升多线程系统的性能，即使在对共享资源竞争激烈时性能表现也相当不错。

Li 在 [44] 中研究了性能表现不对称的多核系统，发现对不同类型的任务，调度算法设计如何对系统中如无执行效率的影响很大。McKenney 在 [29] 中讨论了多核处理器开始出现在嵌入式实时系统中，指出 Linux 操作系统正在适应这一变化。

Ranger 和 Akhter 分别在 [6] 和 [38] 中研究了多核处理器及其体系结构对程序设计、编写的影响，指出多核编程对程序员来说是一种重大的革命，程序员需要适应这一趋势。

Rajagopalan 等人在 [28] 中试图提出了一种新的多核处理器控制框架，兼顾对系统的控制和抽象层次。该框架使用用户提供的信息指导线程调度，使程序员可以比较自由地控制线程。

Zangerl 在 [50] 中总结了近些年来对多核体系结构的调度的研究，认为未来出现的内含可能多达100个核的处理器会彻底改变利用共享资源的方式和当前单线程程序编写的方式。操作系统有责任使用智能的、可配置的调度算法解放程序员，使程序员只专注于业务相关的内容，而不用操心底层的体系结构。

Siddha 介绍了 Linux 内核 2.6 中的调度器，它能感知到处理器中多核的存在，从而可以加速各种程序的执行，无论是多线程的还是单线程的。通过感知到多核，不仅可以提高峰值性能，而且还有节能（Power Saving）的功能。

Kumar 在 [22] 中介绍了 Completely Fair Scheduler，这一算法可以对于一个执行单元（处理器或核），可以精确模拟出多个并行的虚拟执行单元，从而精确地并行执行其上的任务，这样就可以很准确的计算出任务执行的进度。

## 1.2 基础知识

- 进程

  进程是程序的执行，其中不单包括程序代码，还有当前的运行状态，如程序计数器(PC)和寄存器的内容。另外，一个进程一般还包括一个保存临时数据的栈，和一个保存全局变量的数据区。

  程序本身并不是进程，程序是静止的，而进程是运动的，即使两个进程执行的是同一个程序，它们依然是两个不同的进程，因为他们有不同的程序计数器，不同的寄存器和不同的栈。

- 线程

  线程是进程中的一个实体，一个进程可以由多个线程组成。线程不占有系统资源，它通过进程使用系统中的资源。进程中的线程共享进程所拥有的资源，只有很少的私有资源，如寄存器、栈。

- 多核处理器

  多核处理器将两个或多个独立的核心结合在一块集成电路上，其中的核可能共享最高级别的缓存（如Intel Core 2），或者有独立的缓存（如AMD当前

的多核处理器），每个核都可以执行指令。每个核有自己的资源（如执行单元，寄存器，某些级别的缓存等），有了多个执行单元就允许多个线程并行执行，进而实现了更大的并行度。一块电路上的核是否共享资源取决于具体实现。典型的多核实现共享LLC(Last Level Cache)和前端总线资源。多核环境中资源共享（如LLC,前端总线资源,电源管理状态等）带来了新的挑战。

- 调度程序

  当操作系统是多道程序系统时，通常就会有多个任务竞争 CPU。当多个进程就绪时，操作系统就必须按照系统的资源分配策略所规定的资源分配算法决定先运行哪一个进程。操作系统中做出这种决定的部分称为调度器（scheduler）,它使用的算法成为调度算法（scheduling algorithm）。

- 启发式算法

  启发式算法是相对最有算法提出来的，它使得在可接受的计算费用内去寻找最好的解，但不一定能保证所得解的可行性和最有性，甚至在多数情况下，无法阐释所得解同最优解的近似程度。

  在某些情况下，特别是在处理一些现实情况中的问题中，寻找最优解的最有算法的运算时间特别长或计算难度随问题规模的增加而呈现指数增长的情况，此时只能通过启发式算法得到问题的一个可行解，不考虑算法所得解和最有解的偏离程度。常见的贪婪算法（greedy algorithm）即是一种常见的启发式算法。

- 基准测试

  基准测试就是使用专门的测试软件或工具，或者通过让实际的应用程序来执行特定的操作，来考察系统或核心部件的性能。优秀的基准测试应当能够正确地模仿用户在实际应用中的使用情况。

  基准测试通过运行一个计算机程序，一组测试程序，或进行其他操作，通常进行大量的标准测试和试验，以评估测试对象的相对表现。

- 体系结构

  计算机体系结构（Computer Architecture）是程序员所看到的计算机的属性，即概念性结构与功能特性,这是1964年 C.M.Amdahl 在介绍 IBM 360 系统时提出的。按照计算机系统的多级层次结构，不同级程序员所看到的计算机具有不同的属性。一般来说，低级机器的属性对于高层机器程序员基本是透明的，通常所说的计算机体系结构主要指机器语言级机器的系统结构。

  CPU（Central Processing Unit）和内存（Memory）是计算机的两个主要组

成部分，内存中保存着数据和指令，CPU 从内存中取指令（Fetch）执行，其中有些指令让 CPU 做运算，有些指令让 CPU 读写内存中的数据。

Cache 是一种高速缓冲存储器，是为了解决 CPU 和内存之间的速度不匹配而采用的一项重要技术。CPU 运行程序是一条指令一条指令地执行的，而且指令地址往往是连续的，即 CPU 在访问内存时，在较短的一段时间内往往集中于某个局部，这时候可能会碰到一些需要反复调用的子程序。电脑在工作时，把这些活跃的子程序存入比内存快得多的 Cache 中。CPU 在访问内存时，首先判断所要访问的内容是否在 Cache 中，如果在，就称为“命中”，此时 CPU 直接从 Cache 中调用该内容；否则，就称为“不命中”，CPU 只好去内存中调用所需的子程序或指令了。CPU 不但可以直接从 Cache 中读出内容，也可以直接往其中写入内容。由于 Cache 的存取速度相当快，通常是内存的几十倍，使得 CPU 的利用率大大提高，进而使整个系统的性能得以提升。

## 1.3 本文安排

在本文中，我们考虑在多核处理器系统中动态分配任务，按照各个成本（运行成本、通信成本和迁移成本）和的大小综合考虑决定任务的分配位置，将系统分为两级调度域，实现各处理器负载平衡、各核负载平衡，从而达到尽可能大的吞吐量，尽可能快的运行任务。

在本文后面的部分中，“定义与假设”一章中给出本文中使用的定义与假设，包括运行成本、通信成本和迁移成本，以及本文中对负载的定义；算法在“提出的算法及其讨论”一章中给出，其中给出了算法的基本框架和主要步骤，然后讨论了算法的一些细节，包括如何决定分配线程到哪个核上，以及如何迁移线程以平衡负载；“示例及程序模拟”一章中展示算法的一个简单应用，在一个双处理器双核的系统中分配进入的几个任务，然后给出了模拟该算法的程序的主要设计思路；最后，“总结”一节概括了本文的内容，并展望了未来会出现的新的体系结构。

# 第2章　定义和假设

## 2.1　定义

运行单元（**Running Unit**），**RU** 这里概指进程和线程。

一个任务，即一个进程，是由一个或多个线程组成的，在没有歧义的情况下，运行单元同时包括进程和线程。

计算单元（**Computation Unit**），**CU** 这里概指处理器和核。

CPU 进行计算就是依靠其上的核来完成的，所以在不出现歧义的情况下也可以说处理器和核都是计算单元。

计算能力（**Computation Power**），**CP** 这里指计算单元运算的相对速度。

CPU 中的晶体管是非常多的（最新的处理器中晶体管数量达到了23亿[59]），经过长时间的使用多少都会除现破裂、老化、损坏，这样就使得当初计算速度一样的多个核速度不再相同，从而使得各处理器的速度也不相同。

我们这里通过相对速度来衡量各计算单元的运算速度：通过基准测试测出各核的速度，将最慢的那个核的速度定为 1，所有其余的核的相对速度定义为通过基准测试得到的速度与最慢的核的速度的比值，因此，所有的核的相对速度都是 $\geq 1$ 的。

完全公平调度（**Complete Fair Scheduling**），**CFS**，Linux 内核 2.6.23 引入的一个任务调度算法。

CFS 的设计思想可以归结为一句话：“CFS 在真实硬件上模拟出了一个理想的、精确的多任务CPU” [21]，其目标是最大化总体处理器利用率。

CFS 将处理器时间尽量公平地分配给计算单元上的任务，保证每个任务得到相同的时间份额。因为 CFS 可以保证调度是公平的，若知道了每个任务完成的时刻，就可以简单的计算出每个任务在该计算单元上运行时何时运行完成，因此，我们在每个核上使用 CFS 。

由于在核上使用 CFS 调度算法，所以每个核的计算能力被其上的任务平分。例如：三个线程 A、B、C 在一个核上单独运行时分别用 $a$，$b$，$c$ 时间后运行完成，$a < b < c$。若这三个线程同时被分配到这个核上，则线程 A 于 $3a$ 时刻运行完成，线程 B 于 $3a + 2(b - a) = a + 2b$ 时刻运行完成，线程 C 于 $3c$ 时刻运行完成。如图 (2.1) 所示：

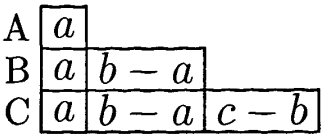


图 2.1: CFS 示意

$RC_T$ 运行成本（Running Cost），线程 T 在计算能力为 1 的核上运行完成所需要的时间。

这里我们将一个线程的运行成本 RC 定义为该线程在计算能力为 1 的核上，即最慢的核上，运行完成所需要的时间。这样，该线程在其他核上的运行时间也可以很容易地计算出来。

例如若一个线程在最慢的核上的运行成本为 $a$，则其在计算能力为 $b$ 的核上的运行成本 RC 为 $\frac{a}{b}$。

$CC_{TT'}$ 通信成本（Communicating Cost），运行单元 T 同其他运行单元 T' 通信的成本。

操作系统中运行的各运行单元可能相互作用，若一个运行单元影响其他运行单元或被其他运行单元所影响，则称该运行单元为合作的。合作的原因包括：

- 信息共享，如同时使用一些资源，如打开同一个文件，使用同一块内存。
- 加速计算，如将一项工作分为几个任务来进行加速。
- 模块分解，如需要以模块化的方式来构建系统。
- 方便使用，如编程时可以同时编辑、编译和调试。

通信的方式包括消息传递、缓冲区共享以及管道等。

通信需要花费一定的时间，会增加运行单元运行完成所需的时间，所以我们这里将通信成本以时间计，同时，我们区分处理器间的通信成本和处理器内部各核的通信成本，显然，CPU 间的通信成本要大于处理器内部各核的通信成本，因为两个处理器对总线的占用比只有一个处理器申请使用所用时间要长。但是，即使两个线程被分配到同一个核上，它们之间的通信仍然需要成本，这里我们将其定为与处理器内部各核的通信成本相同。

$MC_{TAB}$ 迁移成本（Migrating Cost），将运行单元 T 从运行单元 A 迁移到 运行单元B 的成本。

迁移就是在计算单元之间转移运行单元。如果一个计算单元发生故障或者过载，则可以迁移运行单元以平衡负载。

迁移最明显的好处是，把运行单元移动到未充分利用的计算单元上，提高性能和吞吐量，加快运行单元运算完成。另外，迁移也可以提高可靠行，增加容错，如处理器或核可能损坏，或者变得不稳定，若运行单元被迁移到其他计算单元，则运行单元可以继续运行。

由于 Cache 失效、寄存器内容需要重新设置、存储器内容需要传输等原因，进程迁移可能推迟运行单元运行完成的时间，而且，若一个进程中的各线程被分配到不同的处理器上，则各线程的通信成本（CC）会大大增加。所以，除非迁移的好处非常明显，一般还是尽量避免运行单元迁移，且由于在处理器间迁移的成本大于在处理器内部各核间迁移的成本，所以迁移主要在处理器内部各核之间进行。

调度域（**Scheduling Domains**） Linux 2.6 内核引入的概念，用它来维护各处理器、各核之间的负载平衡，最大化多处理器多核系统的效率和性能。调度域（Scheduling Domains）的概念和体系结构密切相关。

调度算法要解决的一个首要问题就是如何发挥这么多CPU 的性能，使得负载均衡。不存某些CPU 一直很忙，进程在排队等待运行，而某些CPU 却是处于空闲状态。

但是在这些CPU 之间进行Load Balance 是有代价的，比如对处于两个不同物理CPU 的进程之间进行负载平衡的话，将会使得Cache 失效。造成效率的下降。而且过多的Load Balance 会大量占用CPU 资源。

为了解决上述的这些问题，内核开发人员Nick Piggin 等人在Linux 2.6 中引入基于Scheduling Domains 的解决方案。

每个 Scheduling Domain 其实就是具有相同属性的一组运算单元的集合，并跟据 CPU、核这样的系统结构划分成不同的级别，结合本文中的体系结构，每个运算单元组成一个独立的调度域。不同级之间通过指针链接在一起，从而形成一种的树状的关系。如图（2.2）所示：

负载平衡就是针对 Scheduling domain 的。从叶节点往上遍历。直到所有的domain 中的负载都是平衡的。当然对不同的domain 会有不同的策略识别是否负载不平衡，以及不同的调度策略。通过这样的方式，从而很好的发挥多处理器多核的效率。

负载（**Load**） 我们这里将负载定义为运行单元在运算单元上运行完成所需要的时间，负载和运行单元上的代码量密切相关。

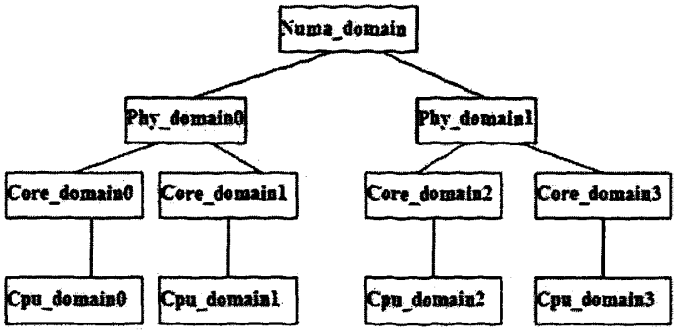


图 2.2: 调度域原理

对于核来说，我们将其负载定义为其上运行单元运行完成需要的时间。

对于处理器来说，我们记录处理器核上的负载最大者和最小者，以供在负载平衡时迁移线程使用。

本文中的目标体系结构 本文中的目标体系结构如图 2.3 所示，其中只关注 CPU、Cache 和内存。

在这种体系结构中包含有多个 CPU，每个 CPU 上的核共享 L2 缓存，所有的 CPU 共享系统的内存。在每个 CPU 上，每个核都有自己的 L1 缓存。每个CPU中核的数目可以是不同的，计算能力也有可能是不同的。

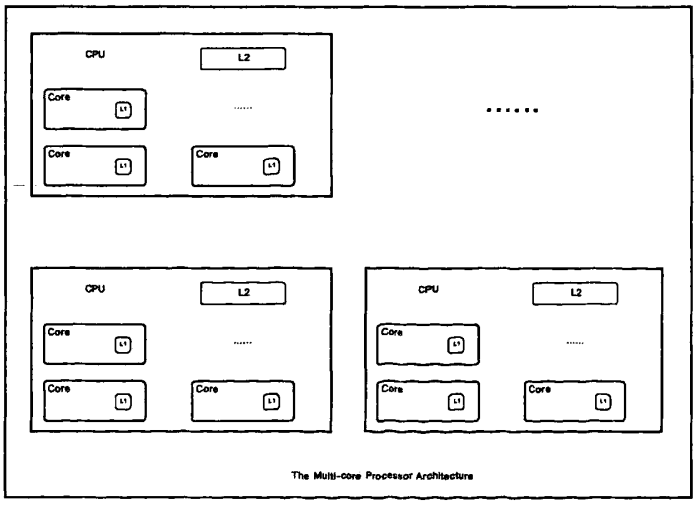


图 2.3: 本文中的目标体系结构

## 2.2 假设

1. 系统中各核的计算能力 CP 是已知的。
   由于各核的计算能力 CP 定义为计算速度的比值，除了从理论上可以估计出来以外，还可以通过基准测试（Benchmark）的方法测量比较出来。

2. 本文中不考虑调度程序本身的成本，且调度程序可以立即完成。

3. 本文中所有的成本统一转换为时间。

4. 一个任务中的各线程指令数目是已知的，即运行成本 RC 是已知的。
   在这里我们假设知道运行成本 RC。但在实际情况中，不可能确切知道任务中各线程的运行成本 RC，既使同一个任务运行多次，其每次的完成时间也可能次次都不相同，不过可以估算出一个大致的时间。

5. 一个任务中的各线程与其它任务中的各线程的通信成本 CC 是已知的。
   由于一个任务间各线程联系紧密，而且内存的速度与 Cache 的速度相差甚大，所以，在一个处理器内部的各核之间的通信成本 CC 比处理器间的通信成本高，所以，尽可能地将任务中的各线程分配到同一个处理器上。

6. 任务中的各线程的在处理器间的迁移成本 MC 是已知的。
   由于 Cache 失效等原因，线程在处理器间的迁移成本 MC 比在同一个处理器的各核间移动的迁移成本 MC 要大，所以，尽可能只在处理器内部的各核之间移动。

7. 各处理器间的通信成本 CC 是相同的，各核间的也是相同的。
   两个线程通信，其在处理器间的的通信成本 CC，不会因为线程处于不同的处理器而不同，同理，两个线程在同一个处理器核间的通信成本 CC 也是相同的。

8. CPU 中核的计算能力可以不相同.
   由于本身的性能原因，以及晶体管使用中的老化损坏，CPU 中各核的计算能力可能是不同的，这里我们将核的运算能力定义为各核运算速度的比值，因此运算能力 $\geq 1$。

9. 在线程运行过程中，为方便计算，我们假设通信成本 CC 按时间平均分配。
   通信成本 CC 是平均分配的，则当线程迁移时，其剩余的通信成本按比例转换为新的通信成本，例如：通信成本原来为 $a$，运行一段时间后剩余 $b$，若

这时线程迁移，通信成本变为 $c$，则当前剩余的通信成本为 $c \times (1 - \frac{b}{a})$。

10. 在本文中任务是进程的同义词。

# 第3章　提出的算法及其讨论

## 3.1　算法

1. 初始化，将各处理器、各内核的负载都置为 0。

2. 有任务进入时，按照负载大小，将任务中的各线程分配到核上。
   当有任务进入系统，根据这个任务中的各个线程在各核上的运行成本 RC 以及线程间的通信成本 CC，找出其和最小的一组核的集合，然后将该任务的各线程分配到该集合中的核上，即能最早完成任务的核上:

$$\min\left\{\text{Load} + RC + CC\right\}$$

   并记录相关的各计算单元的负载。对任务中的每个线程重复以下步骤：

   (2a) 找出原来负载最小的核，记为 $\text{Core}_1$。若多个核心的原负载都相同，则选择计算能力 RC 较大的核；

   (2b) 找出可以使当前线程的负载最小的核，记为 $\text{Core}_2$。若相同则选择原来的负载最小的核；

   (2c) 比较将任务分配到 $\text{Core}_1$ 和 $\text{Core}_2$ 上的负载，分配线程到负载较小的核上。

3. 当有线程结束时，在各个级别的调度域进行负载平衡。

   (3a) 对处理器级别调度域上的负载进行平衡；
   对于每个处理器，找出负载最重的核和负载最轻的核，若一个处理器上负载最轻的核的负载（记该处理器为 $\text{CPU}_a$）比另一个处理器上负载最重的核的负载（记该为 $\text{CPU}_b$）还要重，则负载是不平衡的，需要将任务从 $\text{CPU}_a$ 上移动到 $\text{CPU}_b$ 上。否则，负载是平衡的，进入步骤 (3b)。

   (3b) 对内核级别调度域上的负载进行平衡；
   在一个处理器内，找出负载最重的核 (记为 $\text{Core}_a$) 和负载最轻的核 (记为 $\text{Core}_b$)，若 $\text{Core}_a$ 上的负载比 $\text{Core}_b$ 上的负载重，则考虑将某个线程从 $\text{Core}_a$ 移动到 $\text{Core}_b$，否则负载是平衡的，转到步 (3c)。

   (3c) 当步骤 (3a) 和步骤 (3b) 中有线程移动，返回步骤 (3a) 中再次平衡负载，否则进入下一步。

4. 若有线程结束，则返回步骤（3）平衡负载，若有新任务进入，则返回步骤（2）进行分配，否则，调度程序无事可做，挂起，系统运行各核上的线程。

5. 当系统关闭时，算法结束。

## 3.2 讨论

在算法第 2 步中，由于各核的计算能力和其上原有的负载以及任务中各线程的运行成本 RC 和线程间的通信成本 CC 是已知的，这里使用启发式算法，而不需要计算出每个线程在每个核上的 RC 和 CC，那样的话时间复杂度会大大增加。这样虽然不能保证每次都是最优，但是复杂程度大大降低。

当某线程被分配到一个已经有线程的核上时，要计算这个核的负载时，除了要算上本线程的运行成本 RC 和通信成本 CC 的和，还要加上该核上原有的负载。

当任务进入，首先对任务中的各线程按照运行成本 RC 从大到小进行排序，因为线程间的通信是对称的，所以这里按照运行成本从大到小依次分配任务中的各线程到各核上。

针对每个线程，根据

$$\min\left\{\text{Load} + RC + CC\right\}$$

我们这里需要找出两种核，一种是原有的负载 Load 最小的核，另一种是使得 RC 与 CC 的和最小的核，比较若线程分配其上，这两个核分别的总负载，然后将线程分配到较小的核上。

若一个核上原有的负载是相同的，优先选择那些计算能力 CP 较大的核，若 RC 与 CC 的和相同，则优先选择那些负载较小的核。分配完成之后，计算并记录各核的负载。

每次有线程结束时，处理器、核心可能空闲出来，从而使得负载不平衡，这样我们就需要平衡负载。

在本文中，系统中的调度域分为两级，其中处理器（CPU）为一级，核心（Core）为一级。首先在处理器级别上平衡负载，然后再在各个处理器内部的核心上平衡负载，这样可以保证各级调度域上的负载大致平衡。

在调度域中的处理器这一级中，每次在其上分配线程时都记录其中负载最重的核和负载最轻的核。当要在这一级调度域中进行负载平衡时，即在步

骤 (3a) 中，比较各个处理器的负载最重的核和负载最轻的核，若一个处理器（记为 $CPU_a$）负载最轻的核的负载，比另一个处理器（记为 $CPU_b$）负载最重的核的负载还要中，则这两个处理器间的负载是不平衡的，需要将 $CPU_a$ 上的某个线程移动到 $CPU_b$上。

在调度域中的核心这一级中，即在步骤 (3b) 中，若一个核（记为 $Core_A$ ）的负载比另一个核（记为 $Core_B$ ）的负载重，则考虑将 $Core_A$ 上的某个线程移动到 $Core_B$ 上。

在步骤 (3) 中，无论在调度域中的处理器这一级别，还是在核心这一级别，迁移前都需要计算迁移前后的成本：

$$\begin{aligned} Cost_A &= \mathrm{Load}_A + RC_{TA} + CC_{TA} \\ Cost_B &= \mathrm{Load}_B + RC_{TB} + CC_{TB} + MC_{TAB} \end{aligned}$$

若 $\mathrm{Cost}_A$ 比 $\mathrm{Cost}_B$ 大，则迁移有利，值得迁移，否则不迁移。

在上述两级调度域中，每次只移动一个线程，因此，我们需要步骤 (3c) 来不断迁移负载，直至没有线程被迁移。若没有线程移动，就说明负载已然平衡。

在算法第 4 步中，若有线程结束，则负载可能变得不平衡，返回步骤 (3) 平衡负载；若有新任务进入，则返回步骤 (2) 中决定将任务分配到哪个 CPU 上；否则调度程序挂起，直到有线程结束，或者有新任务进入，这时，系统运行各任务，在每个核上，使用 CFS 算法运行其上的每个线程。

调度算法是操作系统理论中非常重要的一部分，调度程序是操作系统中非常基础的部分，因此，调度程序在系统中不断地运行，在空闲时挂起，一旦需要立即重新投入运行，只有在系统关闭时调度程序才结束运行。

# 第4章　程序模拟

## 4.1　简单示例

假设一系统中有两个 CPU，每个 CPU 都是双核的，如图 (??) 所示

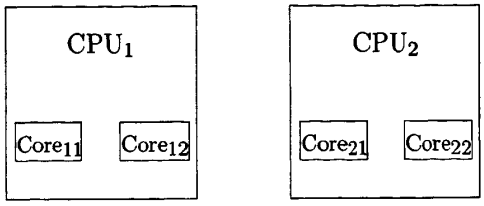


图 4.1: 本例中的体系结构

| 各核 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 计算能力 | 1 | 1.1 | 1.7 | 1.4 |

时间 T=0 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | 0 | 0 | 0 | 0 |
| 线程 | - | - | - | - |

第一个任务 $T_1$ 进入，设其有三个线程，分别记为 $t_{11}$，$t_{12}$，$t_{13}$：

| 线程 | $t_{11}$ | $t_{12}$ | $t_{13}$ |
|---|---|---|---|
| 运行成本 RC | 2 | 3 | 5 |

| 两线程 | $t_{11}$ 与 $t_{12}$ | $t_{12}$ 与 $t_{13}$ | $t_{13}$ 与 $t_{11}$ |
|---|---|---|---|
| CPU 间通信成本 CC | 1 | 2 | 1 |
| 同 CPU 核间 CC | 0.5 | 1 | 0.5 |

首先将线程 $t_{13}$ 分配在核 $Core_{21}$ 上，这样，$Core_{21}$ 上的负载为$\frac{5}{1.7} = 2.94$。

然后分配线程 $t_{12}$ ，现在各核的负载：

| 计算能力 CP | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | 0 | 0 | 2.94 | 0 |

因为核 $Core_{22}$ 的负载小于核 $Core_{21}$，且线程 $t_{12}$ 被分配在核 $Core_{22}$ 上与被分配到核 $Core_{21}$ 与线程 $t_{13}$ 的通信成本 CC 相同，所以将 $t_{12}$ 分配在核 $Core_{22}$ 上，其负载为 $\frac{3}{1.4}=2.14$。

最后，分配线程 $t_{11}$，当前各核的负载如下所示：

| 计算能力 CP | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | 0 | 0 | 2.94 | 2.14 |

在通信成本 CC 相同的情况下，$Core_{21}$ 计算能力 CP 比 $Core_{22}$ 的计算能力 CP 大，$Core_{12}$ 计算能力 CP 比 $Core_{11}$ 的计算能力 CP 大，所以考虑将线程 $t_{11}$ 分配到$Core_{12}$ 和$Core_{21}$ 这两者之一上。

若将线程 $t_{11}$ 分配到核 $Core_{12}$ 上，则其上的负载为

$$RC_{(11)}+CC_{(11)(12)}+CC_{(11)(13)}=\frac{2}{1.1}+1+1=3.82$$

若将线程 $t_{11}$ 分配到核 $Core_{21}$ 上，则其上的负载为

$$RC_{原有}+RC_{(11)}+CC_{(11)(12)}+CC_{(11)(13)}=2.94+\frac{2}{1.7}+0.5+0.5=5.12$$

因此将线程 $t_{11}$ 分配在核 $Core_{12}$ 上。

这样我们可以得到任务 $T_1$ 中各线程的运行时间：

线程 $t_{13}$ 的运行时间：

$$RC_{(13)}+CC_{(13)(12)}+CC_{(13)(11)}=\frac{5}{1.7}+1+0.5=4.44$$

线程 $t_{12}$ 的运行时间：

$$RC_{(12)}+CC_{(12)(13)}+CC_{(12)(11)}=\frac{3}{1.4}+1+0.5=3.64$$

线程 $t_{11}$ 的运行时间：

$$RC_{(11)}+CC_{(11)(12)}+CC_{(11)(13)}=\frac{2}{1.1}+1+1=3.82$$

T=1 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | 0 | $3.82-1.1=2.72$ | $4.44-1.7=2.74$ | $3.64-1.4=2.24$ |
| 线程 | - | $t_{11}$ | $t_{13}$ | $t_{12}$ |

第二个任务 $T_2$ 进入，有两个线程 $t_{21}$、$t_{22}$:

| 线程 | $t_{21}$ | $t_{22}$ |
|---|---|---|
| 运行成本 RC | 2 | 2 |

| 两线程 | $t_{21}$ 与 $t_{22}$ | $t_{21}$ 与$t_{13}$ | $t_{22}$ 与 $t_{13}$ | 其余 |
|---|---|---|---|---|
| CPU 间通信成本 CC | 2 | 2 | 1 | 0 |
| 同 CPU 核间 CC | 1 | 1 | 0.5 | 0 |

两个线程的运行成本 RC 都为 2，由于 $Core_{11}$ 上的负载为 0，所以先将线程 $t_{21}$ 分配到核 $core_{11}$ 上。

然后分配线程 $t_{22}$，当前各核的负载如下：

| 计算能力 CP | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $\frac{2}{1}+2=4$ | 2.72 | 2.74 | 2.24 |

综合考虑，将线程 $t_{22}$ 分配到核 $Core_{22}$。

这样我们可以得到任务 $T_2$ 中各线程的运行时间：

线程 $t_{21}$ 的运行时间：

$$RC_{(21)} + CC_{(21)(22)} + CC_{(21)(13)} = \frac{2}{1}+2+1=5$$

线程 $t_{22}$ 的运行时间：

$$RC_{(22)} + CC_{(22)(21)} + CC_{(22)(31)} = \frac{2}{1}+2+0.5=4.5$$

T=2 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $5-1=4$ | $2.72-1.1=1.61$ | $2.74-1.7=1.04$ | $2.24+4.5-1.4=5.34$ |
| 线程 | $t_{21}$ | $t_{11}$ | $t_{13}$ | $t_{12}$，$t_{22}$ |

此时没有新任务进入系统，且不需要平衡负载，调度程序挂起。

T=3 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $4-1=3$ | $1.61-1.1=0.51$ | $1.04-1<0$，完成 | $5.34-1.4=3.94$ |
| 线程 | $t_{21}$ | $t_{11}$ | - | $t_{12}$，$t_{22}$ |

这时需要将 $Core_{22}$ 上的两个线程 $t_{12}$，$t_{22}$ 中某个线程移动到 $Core_{21}$ 上，两者的迁移成本 MC 为：

| 迁移成本 | $t_{12}$ | $t_{22}$ |
|---|---|---|
| 同 CPU 核间 MC | 1 | 2 |

因此将线程 $t_{12}$ 从核 $Core_{22}$ 上移动到核 $Core_{21}$ 上。由于核 $Core_{22}$ 上有两个线程，两者共享核的运算能力，所以线程 $t_{12}$ 剩余 0.84，线程 $t_{22}$ 剩余 3.1。调度后：

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | 3 | 0.51 | 1.84 | 3.1 |
| 线程 | $t_{21}$ | $t_{11}$ | $t_{12}$ | $t_{22}$ |

T=4 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $3-1=2$ | $0.51-1.1<0$，完成 | $1.84-1.7=0.14$ | $3.1-1.4=1.7$ |
| 线程 | $t_{21}$ | - | $t_{12}$ | $t_{22}$ |

继续运行。

T=5 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $2-1=1$ | 0 | $0.14-1.7<0$，完成 | $1.7-1.4=0.3$ |
| 线程 | $t_{21}$ | - | - | $t_{22}$ |

继续运行。

T=6 时，

| 运行情况 | $Core_{11}$ | $Core_{12}$ | $Core_{21}$ | $Core_{22}$ |
|---|---|---|---|---|
| 负载 | $1-1=0$，完成 | 0 | 0 | $0.3-1.4<0$，完成 |
| 线程 | - | - | - | - |

系统中的所有任务都运行完成，调度程序挂起，等待新任务进入。

## 4.2 程序模拟

在提出的算法后，我们编写了程序来模拟该算法。模拟程序使用Python 语言实现（Python 2.6.5），在Ubuntu 操作系统上实现（Ubuntu 10.04 2.6.32-22-generic）。

整个模拟程序分成四个部分，分别为运行单元（RU）模块，计算单元（CU）模块，系统（System）模块，以及主程序。

在运行单元（RU）模块中，有两个类：Task 和 Thread，分别模拟任务和线程，每个任务中含有一个或多个线程。

在计算单元（CU）模块中，有两个类：CPU 和 Core，分别模拟处理器和内核，每个处理器中含有一个或多个内核。

系统（System）模块是整个模拟程序的核心部分，其中包含两个类：System 和 Scheduler。System 模拟整个系统，其中包含设置处理器的函数，设置调度器的函数，以及向系统添加任务的函数。Scheduler 模拟系统的调度器，实现本文中提出的算法，主要包含两个函数：allocate() 和 balance(),分别对应算法中的第二步和第三步。

主程序main.py 模拟整个系统的运行。初始化时，设置各个任务及其线程的各种属性，以及它们进入系统的时刻，并且设置各个处理器及其上的核的各种属性，然后创建一个系统，在系统中加入这些处理器，并将时间设置为 0，这样系统就初始化完毕。系统运行时，首先将这一时刻要进入系统的任务加入系统中，然后运行调度器类中的 allocate() 函数将这些任务中的线程分配到系统中的各个处理器、内核上，然后系统中的这些运算单元开始运行各个线程，若遇到某个线程运行结束，则调用调度器类中的 balance() 函数平衡负载。然后时间加一，将下一时刻要进入的任务加入系统。这样循环往复，直到系统被关闭，即模拟程序被人为终止。

# 第5章　总结和展望

处理器体系结构为了在未来跟得上摩尔定律，将会经历深刻的变革，含有几十个甚至几百个核的 CPU 也开始大量出现。操作系统的调度算法需要调整以适应正在改变的体系结构，本文提出的算法正是针对这一趋势。

在本文提出的模型中，通过考虑任务的运行成本 RC，通信成本 CC 和迁移成本 MC，将任务中的各个线程分配到不同的处理器和核上，再对各级调度域进行负载平衡，从而达到了尽可能大的吞吐量。

文中的算法虽然使用程序进行了模拟，但是现在尚停留在理论阶段，还只是一个模型，没有在实际的操作系统中实现，以后也许可以在实际的系统中实现，在实践中检验算法实际的运行效果。

随着多核技术的进一步发展，多核处理器广泛应用以及处理器中核心的增多，另外一些因素可能也需要考虑，如对总线、L2 Cache 和内存的竞争加剧等等，研究这些因素对整个系统性能的影响也是未来一个很重要的方向。

另外，含有不同功能的内核的处理器也将会成为未来的趋势之一，Intel 最近就推出了融合了显示芯片 (GPU) 的 处理器（CPU）。在这种类型的处理器中，每个处理器中含有具有不同功能的核，例如有的核擅长做整数、逻辑运算，有的核擅长做浮点运算，有的核可直接解码视频（硬解），有的核可快速解码音频，总之，同一个任务在不同类型的核上运行有不同的运算速度，同一个核运行不同类型的任务有不同的效率，操作系统也需要考虑到这些因素，从而需要对调度算法进一步做出相应的修改。

# 参考文献


[1] Abraham Silberschatz,Peter Baer Galvin,Greg Gagne. *操作系统概念（第7版影印版）*. 高等教育出版社, 2007.

[2] Alexandra Fedorova,Margo Seltzer,Christopher Small,Daniel Nussbaum. Throughput-oriented scheduling on chip multithreading systems. *Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University*, 2004.

[3] Alexandra Fedorova,Margo Seltzer,Christopher Small,Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[4] Andrew S. Tanenbaum,Albert S. Woodhull. *操作系统设计与实现（第三版）（上下册）*. 电子工业出版社, 2007.

[5] Andy Wellings,Martin Schoeberl. Thread-local scope caching for real-time java. In *IEEE INTERNATIONAL SYMPOSIUM ON OBJECT/COMPONENT/SERVICE-ORIENTED REAL-TIME DISTRIBUTED COMPUTING (ISORC 2009.* IEEE Computer Society, 2009.

[6] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *In Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[7] C.L. Liu,James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973.

[8] C.M.Krishna. *实时系统*. 清华大学出版社, 2004.

[9] George Coulouris. *分布式系统概念与设计（原书第4版）*. 机械工业出版社, 2008.

[10] Daniel P. Bovet,Marco Cesati. *深入理解LINUX内核（第三版）*. 中国电力出版社, 2007.

[11] Eitan Frachtenberg, Uwe Schwiegelshohn. Job scheduling strategies for parallel processing. *14th International Workshop, JSSPP 2009*, 2009.

[12] Paul E.McKenney. Smp and embedded real time. *Linux Jour-nal, vol. 2007*, 2007.

[13] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IN PROC. 16TH REAL-TIME SYSTEMS SYMPOSIUM*, pages 152–161, 1995.

[14] H.M.Deitel. *操作系统（第三版）*. 清华大学出版社, 2007.

[15] Florian Huonder. Parallelization for multi-core, 2009.

[16] Intel. Intel i7 mobile processor. http://www.intel.com/products/processor/corei7/mobile/index.htm, 2010.

[17] IT168. 让处理器首次内置GPU. http://notebook.it168.com/a2010/0105-/832/000000832952_1.shtml, 2010.

[18] James Dundas, Trevor Mudge. *Improving data cache performance by pre-executing instructions under a cache miss.* In Proceedings of the 1997 International Conference On supercomputing, 1997.

[19] J.L.Hennessy. *计算机体系结构量化研究方法（英文版）*. 机械工业出版社, 2007.

[20] Linux kernel 2.6.31.4. Linux scheduling domain. Documentation/scheduler/sched-domains.txt.

[21] Linux kernel 2.6.31.4. CFS scheduler. Documentation/scheduler/sched-design-CFS.txt.

[22] Avinesh Kumar. Multiprocessing with the completely fair scheduler. http://www.ibm.com/developerworks/linux/library/l-cfs/, 2008.

[23] Butler W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11:347–360, 1968.

[24] Robert Love. *Linux内核设计与实现*. 机械工业出版社, 2004.

[25] Mark Weiser,Brent Welch,Alan Demers,Scott Shenker. Scheduling for reduced cpu energy. *USENIX Symp. Operating*, pages 13–23, 1994.

[26] Martin Schoeberl,Peter Puschner,Raimund Kirner. A single-path chip-multiprocessor system. In *In Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009), number LNCS 5860*, pages 47–57. Springer, 2009.

[27] Miquel Pericàs, Adrian Cristal, Francisco J. Cazorla, Ruben González, Daniel A. Jiménez, Mateo Valero. A flexible heterogeneous multi-core architecture, 2007.

[28] Mohan Rajagopalan,Brian T. Lewis,Todd A. Anderson. Thread scheduling for multi-core platforms. *In Proceedings of the 11th Workshop on Hot Topics in Operating System*, 2007.

[29] Paul E. McKenney. Smp and embedded real time. *Linux Journal*, 2007.

[30] Michael Pinedo. *调度：原理、算法和系统（英文影印版）（第2版）*. 清华大学出版社, 2005.

[31] Pradeep Kumar Yadav,M.P.Singh,Harendra Kumar. Scheduling algorithm: Tasks scheduling algorithm for multiple processors with dynamic reassignment. *Journal of Computer Systems, Networks, and Communications*, 2008.

[32] Dave Probert. 微软：需为多核芯片重新架构操作系统. http://server.zdnet.com.cn/server/2010/0323/1674346.shtml, 2010.

[33] Randal E.Bryant,David O'Hallaron. *深入理解计算机系统（修订版）*. 中国电力出版社, 2004.

[34] Robert D. Blumofe,Charles E. Leierson. Scheduling multithreaded computations by work stealing, 1999.

[35] Russinovich,Solomon. *深入解析Windows操作系统：第4版*. 电子工业出版社, 2007.

[36] Saisanthosh Balakrishnan,Ravi Rajwar,Mike Upton,Konrad Lai. The impact of performance asymmetry in emerging multi-core architectures. *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.

[37] Sarita V. Adve,Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[38] Shameem Akhter,Jason Roberts. Multi-core programming : Increasing performance through software multi-threading. *Intel Press*, 2006.

[39] Suresh Siddha. Multi-core and linux* kernel. Intel Open Source Technology Center.

[40] William Stallings. *计算机组织与体系结构性能设计（第7版）*. 清华大学出版社, 2006.

[41] Suresh Siddha,Venkatesh Pallipadi,Asit Mallick. Chip multi processing aware linux kernel scheduler. http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf.

[42] Andrew S. Tanenbaum. *分布式系统原理与范型（第2版）*. 清华大学出版社, 2008.

[43] Andrew S. Tanenbaum. *现代操作系统*. 机械工业出版社, 2009.

[44] Tong Li,Dan Baumberger,David A. Koufaty,Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[45] Zeljko Vrba. Implementation and performance aspects of kahn process networks. Doctoral Dissertation, the University of Oslo, 2009.

[46] Wikipedia. Architectural state. http://en.wikipedia.org/wiki/Architectural_state.

[47] Wikipedia. Multi-core. http://en.wikipedia.org/wiki/Multi-core_(computing).

[48] Wikipedia. Completely fair scheduler. http://en.wikipedia.org/wiki/-Completely_Fair_Scheduler, 2007.

[49] W.Richard Stevens,Stephen A. Rago. *UNIX环境高级编程（英文版·第二版）*. 人民邮电出版社, 2007.

[50] Thomas Zangerl. Optimization:operating system scheduling on multi-core architectures. *Seminar "Parallel Computing"*, 2008.

[51] 都思丹. 嵌入式多核处理器系统及视频信号处理技术研究进展. *南京大学学报(自然科学版)*, 2009年01期.

[52] 多核系列教材编写组. *多核程序设计*. 清华大学出版社, 2007.

[53] 哈里斯. *数字设计和计算机体系结构（英文版）*. 机械工业出版社, 2007.

[54] 互动百科. 计算机体系结构. http://www.hudong.com/wiki/计算机体系结构.

[55] 刘勃. Linux scheduling domains. http://www.ibm.com/developerworks-/cn/linux/l-cn-schldom/index.html, 2009.

[56] 汤小丹. *计算机操作系统*. 西安电子科技大学出版社, 2007.

[57] 田杭沛,高德远,樊晓桠,朱怡安. 面向实时流处理的多核多线程处理器访存队列. *计算机研究与发展*, 2009 46(10).

[58] 现在新闻. 抛弃显卡,笔记本步入换芯时代. http://cjsb.cnxianzai.com/keji-/2010/0115/184139.html, 2010.

[59] 新浪科技. Nehalem-ex将发布x86上攻关键计算市场. http://tech.sina.com.cn/b/2010-03-31/09051298754.shtml, 2010.

[60] 于渊. *自己动手写操作系统*. 电子工业出版社, 2005.

[61] 张振华,白中英,陈卉. 基于多核多线程处理器的网络设备设计与实现. *电子设计工程*, 2009年12期.

[62] 赵炯. *Linux 内核完全剖析*. 机械工业出版社, 2006.

[63] 周伟明. *多核计算与程序设计*. 华中科技大学出版社, 2009.

[64] 朱福喜. *并行分布计算机中的调度算法理论与设计*. 武汉大学出版社, 2003.

# 致　谢

值此论文完成之际, 谨向给予我无私帮助的老师和同学致以最诚挚的谢意!

首先衷心的感谢我的导师周宇斌教授！本文是在导师周宇斌教授的悉心指导和热心帮助下完成的。三年来, 导师无论从学习、科研和生活等方面都给于了我始终不渝的关怀和鼓励，使我克服了许许多多的困难, 从而顺利地完成了学业。周老师严谨的治学态度、渊博的知识和孜孜不倦的科研精神深深的影响着我。他以身作则、平易近人的风格时时感染着我，开明的学术观点和教育理念让我受益匪浅。

感谢兰州大学数学与统计学院的领导和教师的关心和鼓励,是他们为我创造了良好的学习和生活环境，使我顺利完成了学业。

最后, 我要感谢我的家人和朋友，是他们给了我巨大的物质支持和精神支持，使我能顺利完成学业。

朱文俊

2010 年 5 月于兰州大学

# 附　录

System.py 文件:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

"""
模拟整个系统，该模块中还有 System 和 Scheduler 两个类
"""

import sys
sys.path.append("modules")

from RU import *
from CU import *
from unit import *


class System(unit):
    """
    模拟系统
    """
    def __init__(self):
        """
        构造函数
        """
        self.cpus = []
        self.tasks = []         # 系统中新进入的任务，需要调度后分配到
各个核上。
        self.scheduler = None

    def addCPU(self, _cpu):
        """
        向系统中添加处理器
        """
        if  isinstance(_cpu, CPU):
            self.cpus.append(_cpu)
        else:
            raise TypeError(str(_cpu) + "类型须为 CPU")
```

```
def getCPUs(self):
    """
    取得系统中所有处理器的列表
    """
    return self.cpus


def addTask(self, _task):
    """
    向系统中添加一个任务
    """
    if  isinstance(_task, Task):
        self.tasks.append(_task)
    else:
        raise TypeError(str(_task) + "类型须为 Task")

def addTasks(self, tasks):
    """
    向系统中添加一组任务
    """
    for task in tasks:
        self.addTask(task)

def getTasks(self):
    """
    取得系统中要分配的任务
    """
    return self.tasks

def setScheduler(self, _scheduler):
    """
    设置调度器
    """
    self.scheduler = _scheduler

def go(self, isStep=True):
    """
    系统运行开始,调用调度器分配任务
    """
    if self.scheduler == None:
        raise AttributeError("未设置调度器")
```

```
        self.scheduler.allocate(self, isStep)

    def run(self):
        """
        系统运行任务
        """
        needBalance = False
        for cpu in self.cpus:
            for core in cpu.getCores():
                if len(core.getThreads()) != 0:
                    cp = core.getCP() / len(core.getThreads())
                    for thread in core.getThreads():
                        thread.setLoad((thread.getLoad()-cp<0 and [0]
                        or [thread.getLoad()-cp])[0])

                        if thread.getLoad()-cp<=0:  #若线程运行完成，
则从该核上删去该线程，并平衡负载
                            core.removeThread(thread)
                            needBalance = True
                core.run()
        if needBalance == True:
            self.scheduler.balance()


    def getStatus(self):
        """
        取得系统状态
        """
        status = {} # status 的数据结构为 {内核：[其上的线程列表]}
        for cpu in self.cpus:
            for core in cpu.getCores():
                status[core] = core.getThreads()
        return status

class Scheduler(unit):
    """
    模拟调度器
    """
    def __init__(self):
        """
        构造函数
        """
```

```
        self.system = None

    # 1、分配任务
    def allocate(self, _system, isStep):
        """
        分配任务中的各线程到各个核上
        """
        self.system = _system

        #print "任务 :" + str(self.system.tasks)
        for task in self.system.tasks:
            sortTask(task)  # 对任务中的线程进行排序
            threads = task.getThreads()
            #print "有线程 " + str(threads)

            for thread in threads:
                # 1.1、寻找负载最小的核心
                core1 = None; load1 = 0.0
                for cpu in self.system.getCPUs():
                    for core in cpu.getCores():
                        if core1 == None or core1.getLoad()
                         > core.getLoad():
                            core1 = core
                            load1 = core.getLoad()

                load1 += thread.getRC() / core1.getCP()
                # 遍历该任务中本线程之前的线程
                for pre_thread in threads[:threads.index(thread)]:
                    # 是否在同一个 CPU 中
                    if pre_thread.getCore().getCPU() == core1.getCPU():
                        load1 += thread.getCC(pre_thread)["thread"]
                    else:
                        load1 += thread.getCC(pre_thread)["cpu"]

                # 1.2、寻找可以使得当前线程负载最小(通信成本考虑已分配
的线程)的核心
                core2 = None; load2 = -1
                for cpu in self.system.getCPUs():
                    for core in cpu.getCores():
                        load_tmp = thread.getRC() / core.getCP()

                        for pre_thread in threads[:threads.index(thread)]
```

```
                        # 是否在同一个 CPU 中
                        if pre_thread.getCore().getCPU() == cpu:
                            load_tmp += thread.getCC(
                            pre_thread)["thread"]
                        else:
                            load_tmp += thread.getCC(
                            pre_thread)["cpu"]

                        if load2 < 0 or load2 > load_tmp:
                            load2 = load_tmp
                            core2 = core

                if core2 == None or load1 < load2:
                    core = core1
                else:
                    core = core2

                # 1.3 分配线程
                core.addThread(thread)
                core.addToLoad(thread.getRC() / core.getCP()) # RC
                for pre_thread in threads[:threads.index(thread)]:  # CC
                    if pre_thread.getCore().getCPU() == core1.getCPU():
                        core.addToLoad(thread.getCC(pre_thread)["thread"])
                        pre_thread.getCore().addToLoad(
                        thread.getCC(pre_thread)["thread"])
                    else:
                        core.addToLoad(thread.getCC(pre_thread)["cpu"])
                        pre_thread.getCore().addToLoad(
                        thread.getCC(pre_thread)["cpu"])

        # 清空任务列表
        del self.system.tasks[:]

        if isStep == True:
            return

    # 2、平衡负载
    def balance(self):
        """
        平衡负载，当有线程运行结束时进行。
        """
        moved = False
```

```
        load = {} # 内容为 {cpu:{'low':core1,'high':core2}}
        while True:
            # 2.1 在处理器间移动线程
            for cpu in self.system.getCPUs():    # 得到各个处理器最大
和最小负载的核
                low = high = cpu.getCores()[0]
                for core in cpu.getCores()[1:]:
                    if core.getLoad() > high.getLoad():
                        high = core
                    if core.getLoad() < low.getLoad():
                        low = core
                load[cpu] = {'low':low,'high':high}

            lows = []    # 最小负载列表
            highs = []     # 最大负载列表
            for cpu in self.system.getCPUs():
                lows.append(load[cpu]['low'])
                highs.append(load[cpu]['low'])
            sortCore(lows)
            sortCore(highs)

            if lows[-1].getLoad() > highs[0].getLoad():
                highCore = load[lows[-1].getCPU()]['high']
                lowCore = load[highs[0].getCPU()]['low']

                # 在处理器间移动线程，并标记
                lowMC = highCore.getThreads()[0].getMC(lowCore)
                threadMoved = highCore.getThreads()[0]
                for thread in highCore.getThreads()[1:]:
                    if lowMC > thread.getMC(lowCore):
                        lowMC = thread.getMC(lowCore)
                        threadMoved = thread
                if (highCore.getLoad() - threadMoved.getLoad()) >
                (lowCore.getLoad()
        + threadMoved.getLoad()
        * highCore.getCP()
        /lowCore.getCP()
        + threadMoved.getMC(lowCore)):
                    highCore.removeThread(threadMoved)
                    highCore.setLoad(highCore.getLoad() -
                     threadMoved.getLoad())
                    lowCore.addThread(threadMoved)
```

```
                    lowCore.setLoad(lowCore.getLoad()
+ threadMoved.getLoad() *
                     highCore.getCP() / lowCore.getCP()
+ threadMoved.getMC(lowCore))
                    threadMoved.setCore(lowCore)

                    moved = True


            # 2.2 在内核间移动线程
            for cpu in self.system.getCPUs():
                sortCore(cpu.getCores())
                cpu.getCores().reverse()    # 从大到小
                if cpu.getCores()[0].getThreads() == []:
                    continue
                lowMC = cpu.getCores()[0].getThreads()[0].getMC(
                 cpu.getCores()[-1])
                threadMoved = cpu.getCores()[0].getThreads()[0]
                for thread in cpu.getCores()[0].getThreads()[1:]:
                    if lowMC > thread.getMC(cpu.getCores()[-1]):
                        lowMC = thread.getMC(cpu.getCores()[-1])
                        threadMoved = thread
                if (cpu.getCores()[0].getLoad() -
                 threadMoved.getLoad() ) >
        ( cpu.getCores()[-1].getLoad()
        + threadMoved.getLoad() * cpu.getCores()[0].getCP()
        / cpu.getCores()[-1].getCP()
        + threadMoved.getMC(cpu.getCores()[-1])):
                    cpu.getCores()[0].removeThread(threadMoved)
                    cpu.getCores()[0].subtractFromLoad(
                     threadMoved.getLoad())
                    cpu.getCores()[-1].addThread(threadMoved)
                    cpu.getCores()[-1].addToLoad(
threadMoved.getLoad()
* cpu.getCores()[0].getCP()
/ cpu.getCores()[-1].getCP()
        + threadMoved.getMC(cpu.getCores()[-1]))
        threadMoved.setCore(cpu.getCores()[-1])

                    moved = True

            # 若没有线程移动，则退出
```

```
            if not moved:
                return


def sortTask(task):
    """
    给定一个任务，对其线程按照运行成本 RC 进行排序
    """
    if not(isinstance(task, Task)):
        raise TypeError("类型应为 Task")
    # 排序,从小到大
    task.getThreads().sort(compareThread)

    # 反序，从大到小
    task.getThreads().reverse()

def compareThread(thread1, thread2):
    """
    对两个线程按运行成本 RC 进行比较
    """
    if not(isinstance(thread1, Thread)
     and isinstance(thread2, Thread)):
        raise TypeError(str(thread1) + "和"
         + str(Thread2) + "的类型都应为 Thread")
    return cmp(thread1.getRC(), thread2.getRC())

def sortCore(cores):
    """
    给定一个核的列表，对核按照负载进行排序
    """
    # 排序，从小到大
    cores.sort(compareCore)

def compareCore(core1, core2):
    """
    对两个核按负载 Load 进行比较
    """
    if not(isinstance(core1, Core)
     and isinstance(core2, Core)):
        raise TypeError(str(core1) + "和"
         + str(core2) +  "的类型都应为 Core")
    return cmp(core1.getLoad(), core2.getLoad())
```