支持服务质量的 Linux 内核设计与实现*)

施笑安[1,2] 周兴社[1] 林 奕[1]

(西北工业大学计算机学院 西安 710072)[1] (空军工程大学电讯工程学院 西安 710077)[2]

摘 要 Linux操作系统对实时 QoS 支持较弱,但具有很好的实时应用前景。本文选择对 Linux 在内核级进行 QoS 扩展。利用 Linux 进程管理策略与机制分离的设计风格,设计了一个内核级的支持 QoS 的实时调度器。不需要修改应用,就能满足应用的 QoS 需求。选择 EDF 实时调度算法作为扩展目标,引入了"预留"对象这一数据结构,实现了进程间的共享 QoS。该新的 Linux 内核能更好地支持 QoS,满足了实时系统的 QoS 需求。

关键词 服务质量,实时,Linux,内核,操作系统

Design and Implement of Linux Kernel Supporting Quality of Service

SHI Xiao-An[1,2] ZHOU Xing-She[1] LIN Yi[1]

(Computer Institute, Northwestern Polytechnical University, Xi'an 710072)[1]

(The Telecommunication Engineering Institute, Air Force Engineering University, Xi'an 710077)[2]

Abstract Currently, Linux operating system, one of the most popular operating systems with good perspective in real-time computing, gives weak supports on real-time QoS. This paper choice to extend the kernel of Linux to support QoS. Taking advantage of the separation of process management policy and mechanism in Linux, design a kernel level real-time scheduler. This design enables to provide QoS supports without any changes to applications. In the design, choose EDF real-time scheduling algorithms as extendable target and develop a data structure named "reserve" object, so that QoS can be shared in Linux processes. This new Linux kernel with support of QoS could satisfy the QoS requirements of real-time applications better.

Keywords QoS, Real-time, Linux, Kernel, Operating systems

1 介绍

目前在通信网络领域服务质量 QoS(Quality of Service)的概念与技术已取得了重要的研究成果,使得网络环境能适应各类应用的优质服务需求。从基于服务(Service-Based)的系统设计思想出发,实时计算平台也同样存在服务质量问题。例如,对实时应用而言,时间、可靠性、安全性等重要的非功能需求可以看作是实时应用提出的应用 QoS 需求,而服务的提供者则是操作系统本身。操作系统处于硬件底层与上层其它系统软件及应用软件之间,其目标与功能在于有效地管理硬件资源,并向上层应用提供便利的调用接口。因此,操作系统的功能与性能对各类应用系统的影响是至关重要的。本文选择 Linux 系统作为实现 QoS 保证的操作系统,对其内核进行了扩充,使之能够在实时调度上支持 QoS,提高系统的实时性,更好地满足实时系统的需要,具有一定的典型性和普遍性。

2 Linux 的 QoS 需求分析

国际电信联盟将 QoS 定义为:一个或多个对象的集体行为的质量需求集。因为于 QoS 是一种集体行为,所以对象之间需要协商。但最终要保证每个对象所承诺的 QoS 服务,由此来确保总体的 QoS 实现。

2.1 实时系统的 QoS

实时系统与其他普通系统之间的最大的不同就是要满足处理与时间的关系。"在实时计算中,系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间"。对于实时系统来说最重要的要求就是实时操作系统必须具有在一个事先定义好的时间限制中对外部或内部的事件进行响应和处理的能力。此外它还需要有效的中断处理能力来处理异步事件和高效的 I/O 处理能力来处理有严格时间限制的数据收发应用。

将定义的 QoS 参数映射到操作系统级 QoS,例如,进程或线程的执行时间、周期,内存需求和缓存等。由得到的进程/线程的执行时间和周期来进行资源分配,并落实到具体的资源如 CPU 资源和内存资源等。在进行资源分配前要进行准入控制,即对于 CPU 资源,根据进程/线程的执行时间和周期依据调度算法判断进程/线程是否可调度,如果可调度则为它分配相应资源和响应的优先级。

对于操作系统而言,增加对 QoS 的支持,最关键的就是引入 QoS 强化落实机制,其主要内容是处理器资源的调度保证措施。针对协议处理的需求,操作系统需要构造在一个进程内周期性的处理行为机制和在多个进程间调度这些行为的调度机制。这一问题在传统的 Unix 系统中,并未得到很好的解决,它往往采用基于优先级的 RR(时间片轮转)算法来调度进程。为了处理好实时任务之间,实时任务与普通任务之间的调度,可采用一些新的周期性的实时调度算法,例如,RM(Rate-Monotonic)算法、EDF(Earliest Deadline First)算法等。这些方法各有优缺点,一般而言,实现的难度、性能效率以及通用性是考虑的重要因素。

由于 Linux 开放源码,价格低廉,它越来越广泛地应用于实时系统中。因此研究支持 QoS 的 Linux 实时系统极为必要。

2.2 Linux 内核对 QoS 的现有支持

Linux 内核支持 QoS 的基础具体表现在:

(1)实时任务调度策略 Linux 进程调度具有实时与分时兼顾的功能,采用了基于优先级的调度机制,实时进程采用

*)基金项目:获国防基础研究(K1800060504)、航天基金(2003CH100001)、航空基金(2000CB1103)、西北工业大学博士论文创新基金等资助和771所支持。施笑安 讲师、博士生,研究网络与分布式软件、实时嵌入式、分布式系统。周兴社 教授,博士导师。林 奕 博士生。

静态优先级，分时进程采用动态优先数。进程调度与时间事件驱动相结合，有关数据结构短小精悍，有利于快速切换。

(2)系统组织结构　Linux 采用了调度策略与实现机制分离技术。全局核心队列(就绪队列、等待队列、延时队列及事件表)及其管理程序构成了实现机制，而调度策略可依应用环境及特点进行定义。采用评估函数实现策略与机制的映射。该技术是面向应用可扩展操作系统的关键技术之一。

但 Linux 作为实时操作系统在支持 QoS 方面也存在不足。Linux 是在 Unix 基础上设计的，是一个单块式的操作系统，Linux 中的进程有一部分运行在用户模式，而另一些部分运行在内核模式或称系统模式。运行模式的变化是通过系统调用完成的。系统模式具有更高的 CPU 特权。调度过程类似一个黑箱操作，用户只能通过一个可调节参数——nice 值改变进程的动态优先级，但 Linux 系统在支持实时调度上，并不令人满意。它存在着问题如：内核方式下运行的进程不能被用户空间上更紧急的进程所抢占；中断延迟大；系统时钟的分辨率低；实时调度采用"时间片轮转＋先进先出＋优先级＋可抢占"的调度策略。然而，仅有这些是不够的，在大多数实时系统中，尤其是在军事实时系统中，要求计算系统尽可能根据有关应用程序和其执行环境的特定知识采用适当的调度策略。

2.3　Linux 内核剖析

Linux 核心采用类似 MACH 的机制和策略分离的进程调度方法。这是因为多媒体技术及分布式计算技术的发展，对操作系统提出了新的要求。传统的操作系统一般采用固定单一的编程模型和资源管理策略，不能满足复杂多变的应用需求。

目前流行的分布计算、并行计算、可靠计算及实时计算环境，尤其是多媒体应用环境，要求操作系统支持多种模型资源管理策略。为适应应用需求，首先在实时 MACH 中实现了机制和策略分离。该方法具有以下特点：资源管理的策略部分与机制部分分离；策略部分定义接口与资源管理器其它部分隔离；同一资源管理所有策略接口相同，以便动态替换；用户可根据具体应用增加或删除机制对象或策略对象；可继承重用现存策略。资源管理对象由三种对象组成：资源管理器、策略和机制。资源管理器对象为操作系统和用户提供抽象资源，抽象资源共享物理资源，策略决定物理资源的实际分配。策略对象封装了不同的对象，提供资源分配算法。例如，进程资源管理对象的策略对象包括处理机调度算法，即将进程抢占和选择的方法。机制对象由策略对象调用和共享，进程资源管理对象采用就绪队列来管理就绪运行进程。

Linux 进程调度具有机制与策略分离：Linux 采用机制与策略分离的进程调度算法。用户可以根据具体应用需求改变、插入及删除策略，而 Linux 的基于优先级的进程调度机制不能随应用改变。

3　支持 QoS 的 Linux 进程调度的设计与实现

3.1　实时任务的 QoS 定义

调度具有 QoS 保证的实时任务就是用通用的 QoS 参数来替换原先的优先级。

QoS 可由一组参数(CPU 利用率 u，周期 p)来定义，使得实时任务在吞吐率和时间性能上得以保证。例如，我们定义某个任务的 QoS 合同是(15%，20ms)，表示在每个 20ms 的 CPU 时间内至少给某个任务分配 3ms。这个 20ms 的时间段是周期发生的。以上表示的参数定义叫做 QoS 合同或 CPU 预留。

实时任务模型：实时任务具有时间约束，实时任务是由一组在时间上相互关联的执行单元组成，每一个单元称之为 job，多个 job 的序列构成 task(T 任务)。

任务又分抢占式和非抢占式。抢占式任务可以被高优先级任务中断；而非抢占式任务一旦执行不能被中断直到其自动放弃处理机，这样一来高优先级任务就必须等待低优先级任务完成，从而导致优先级倒置。实时调度算法性能评价标准之一是截止期保证率。设 GT 为在截止期内完成的任务数，RT 为待处理的任务集，则可调度任务集 T 保证率 R 满足公式 $R=GT/RT$。另一个评价标准是处理机利用率 U。对于周期任务集 $T(T1,T2,T3,\cdots,Tn)$，设 e_i 为周期任务 T_i 的执行时间，p_i 为周期任务 T_i 的执行周期，则任务调度集 T 的处理机利用率 U 满足公式：$U=\sum_{i=1}^{n} e_i/p_i$

3.2　支持 QoS 的 Linux 进程调度

3.2.1　调度类扩展　通过对 Linux 任务结构(PCB)的分析可知，原 Linux 不支持满足 QoS 的调度类。本设计主要针对实时任务进行扩展，因此要满足实时任务 QoS 保证，除 Linux 原有的调度类外，又增加了一种新的调度类 SCHED-QoS，采用时间作为给定任务的优先级，系统原有的 POSIX RT 调度器使用静态优先级，而在此调度器使用 EDF 算法，它的好处是支持动态的优先级变化。为了支持 EDF 算法，只需做一些改动。这种修改是为了增强对 QoS 的支持。EDF 算法最坏情况下的调度能力是 100%，也就是说如果实时任务的总负载率不超过 100%的话，那么就不会有任务错过截止期。因此剩于的时间就可用来处理尽力而为任务。可通过下面三种措施来支持 QoS：计算，调度策略，准入控制。QoS 合同一经建立，就为任务确定了优先级。EDF 算法要求任务的截止期越短，优先级就越高。扩展后的调度图如图 1 所示。

计算：要估计某一任务的 CPU 的使用率。

调度策略：它使用计算机制确保满足 QoS 合同。例如，当实时任务用完了 CPU 时间片，在被阻塞前别的任务就会立即进行抢占式调度，除非使用 fallback 调度策略让本该被阻塞的进程继续得以运行。有 QoS 保证的进程将比预先没有预约 CPU 时间的进程先得到调度。

准入控制：设置准入控制。

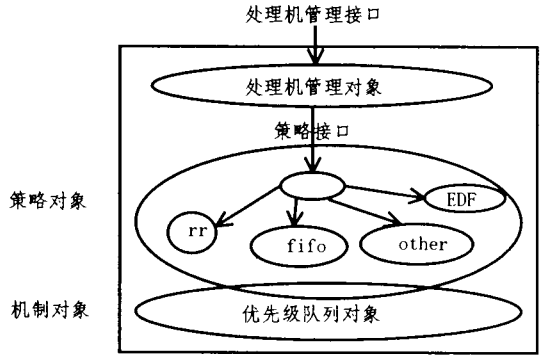

图 1　扩充后的策略对象与机制对象

3.2.2　双重调度策略　如果系统中没有实时任务运行，那么就用调度类(SCHED-OTHER)来处理尽力而为任务；如果有实时任务，由于它的优先级比尽力而为任务的优先级高，所以调度器转去调度实时任务；如果是具有 QoS 保证的任务(调度类 SCHED-QoS)，则使用 EDF 算法调度，EDF 算法要求任务在总负载率不超过 100%情况下才可被调度，这样可以使系统的利用率最大化，尽力而为任务使用剩余的利用率只要与实时任务的利用率之和不超过 100%，就不会有任务错过和任务不会"饥饿"。所谓双重是指当实时任务的处理时间没有超出预留的时间时，采用 SCHED-QoS 调度策略，而如

果出现过载情况下，则将 SCHED-QoS 调度策略转换成 SCHED-OTHER.，把实时任务当作尽力而为任务调度。

双重调度策略的具体实现：在系统调度参数结构里，包含了一个 limit 参数和一个 overren_ policy(fallback_ policy)参数，limit 的值等于分割 period 所得到的时间片，当一个任务用完了它的 QoS 分配时，它的调度策略就下降到 overrun-policy，一直到下一周期开始。

```
/ * 调度策略宏定义 * /
#define SCHED_ OTHER      0
#define SCHED_ FIFO       1
#define SCHED_ RR         2
#define SCHED_ QoS        3
#define SCHED_ IDLE       4
#define SCHED_ PAUSE      5
#define SCHED_ RSV        6
struct rsv_ RSV           6
struct rsv_ param
{int policy,overrun_ policy;(fallbackpolicy) / * 双重调度策略 * /
    int period;/ * 实时 QoS 任务的执行期 * /
    int deadline;/ * 实时 QoS 任务的截止周期 * /
    int slice;/ * 实时任务每个周期内分配的 cpu 时间 * /
    int rt_ priority;(static-priority) / * 使用 FIFO 或 RR 调度策略
        的实时优先级 * /
    int priority;(nice) / * nice 值；使用 OTHER 调度策略的优先级
        * /
}
```

图 2　实时任务数据结构

Linux 内核提供三种基本调度策略，并且采用策略和机制分离方法，因此便于用户进一步扩展调度策略。

3.2.3　*主要数据结构*　在原有任务结构基础上定义有关 QoS 的数据结构，具有 QoS 预留的实时任务数据结构如图 3。

为了支持 QoS，引入了 CPU 资源预留，在本系统中的具体实现是通过建立预留对象使得线程间能共享服务质量。预留对象封装了调度特征以及相关的保证。线程只要加入了预留对象就叫做该预留的成员。所有成员共享同样的调度参数和 QoS。预留对象和普通进程都由同样的内核调度器管理而不需要切换到单独的实时方式或采用多级调度器管理，避免了系统设计的复杂性。

```
struct reserve{ struct rsv_ param * rp;
                char* name;
                int members;/ * 分享预留对象的任务成员 * /
                struct timeval time_ used; / * In the range 0…cpu…
                   max… * /
                struct timeval next_ epoch;}
struct reserve_ list{struct reserve * rsv;/ * 预留对象构成了一个双向
                      链表 * /
                struct reserve_ list * next;
```

如果 rsv 不为空，则将调度策略设置为 SCHED_ RSV，在 task_ struct 中的字段 priority 和 rt_ priority 用 rsv 预留对象代替。因此，有预留的进程的 task_ struct 结构的字段如下：

```
p→rsv! =NULL
p→policy=SCHED_ RSV
p→priority / * 将其忽略不计 * /
p→rt_ priority / * 将其忽略不计 * /
p→ rsv→ policy(SCHED_ OTHER, FIFO, RR, EDF, QoS, IDLE or
    PAUSE) * /预留对象内部的字段如下 * /
p→rsv→priority
p→rsv→rt_ priority
p→rsv==NULL / * 如果 rsv 为空，则进程的 task struct 字段 * /
p→policy(SCHED_ OTHER,FIFO,RR,EDF,IDLE or PAUSE)
p→priority
p→rt_ priority
```

调度时判断不同的调度策略，例如，

```
if(p→policy! =SCHED_ OTHER){ / * 实时进程 * /
        if(policy==SCHED_ RR ‖ policy==SCHED_ EDF){
    weight=1000+(rsv? rsv→rp→rt_ priority:p→rt_ priority);}
```

图 3　预留对象的调度类和参数的数据结构

使用预留对象的优点：

首先它解决了无关联线程之间的共享问题，例如，如果在窗口管理器和虚拟终端控制台的 shell 之间建立预留共享，那么，采用多预留对象就完全没有必要并且浪费时间，因为在同一时刻只有一个控制台起作用。

其次，解决了多线程应用的不同线程具有不同的调度需求这一问题。例如，XMMS(X 多媒体系统)就具有声音、用户接口线程和频谱合成器线程，前者有 QoS 需求而后者用普通非实时调度就可满足需要。因此，如果整个程序只采用单一的优先级，而没有针对不同线程的不同的调度需求，就势必会降低系统的效率。

标准的 Linux 系统并没有提供一种方法让用户设置优先级或 QoS 参数，因此多媒体应用通常自己设置静态优先级，结果并不令人满意。而且，应用必须运行在 root 用户方式下才能设置优先级，这样会造成安全问题。

为了解决以上问题，在本系统中引了预留对象，应用可以建立不同类型的线程间与不同的预留对象间的联系，而这一切都由用户在应用运行前事先配置好。

预留对象的建立采用的是系统调用的方式。系统调用是进程使用内核管理功能的唯一“接口”，应用程序请求内核服务必须通过系统调用。Linux 提供了一组 syscall ()预处理宏指令来引导和完成系统调用。

图 3 数据结构用于描述组成预留对象的调度类和参数。

以上代码表示了如果 rsv 为空，则 weight＝1000＋p→rt_ priority 即 weight 等于普通实时进程的优先级，如果 rsv 不为空则 weight＝rsv→rp→rt_ priority。

用 C 语言简单描述调度策略的实现如图 4：

```
void implent(int pid){ / * 输入参数：pid 为待设置的进程标识。 * /
    if(normal_ flag){
    do_ normal(pid);
    return;}
    if(nice_ level) do_ nice(nice_ level,pid); / * 分时进程调度 * /
    if(rt_ revel) do_ fifo(rt_ level,pid);/ * 普通实时进程调度 * /
    if(rsv_ name) do_ reserve(rsv_ name,pid)} / * QoS 预留实时进
    程调度 * /
void do_ normal(int pid);
void do_ fifo(int pri,int pid);
void do_ nice(int n,int,pid);
void do_ reserve(char * rsv_ name,int pid);
```

图 4　调度策略实现算法

总结　本文通过全面地分析操作系统的实时 QoS 需求，从而定义了实时任务 QoS 参数，建立了通用的实时 QoS 实现模型。并对 Linux 内核进行扩充，利用 Linux 进程管理策略与机制分离的特点，设计和实现了支持 QoS 要求的实时调度器。从而不需要修改应用，也能满足应用的 QoS 需求。并选择 EDF 算法作为扩展目标，引入了“预留”对象这一数据结构，从而在进程间共享 QoS，建立了支持 QoS 的新的 Linux 核心。

参 考 文 献

1　Campbell A T. A Quality of Service Architecture:[Ddoctor Thesis of Computing Department of Lancaster University]. Jan. 1996

2　Goyal P. A Hierarchical CPU Scheduler for Multimedia Operating System. University of Texas at Austin. C. W. Mercer, S. Savage, and H. Tokuda . Processor Capacity Reserves: Operating System Support for Multimedia Application. In: Proc. of the IEEE ICMCS '94, May 1994

3　Barham P. Devices in a Multi-Service Operating System:[University of Cambridge Computer Laboratory Technical Report No. 403]. July 2001.

4　Beck M, Bohme H, Dziadzka M. Linux Kernel Internals. Second Edition. 1998

5　Gopalkrishnan R, Parulkar G M. A Framework for QoS Guarantees for Multimedia Applications within an Endsystem. Washington Univrsity, St. Louis, 2003