MASTER THESIS XIDIAN UNIVERSITY

西安电子科技大学

硕士学位论文

Linux实时调度研究及改进

作者姓名 吴振亚 学校导师姓名、职称 苏锐丹副教授

领　　域 计算机技术 企业导师姓名、职称 朱军高级工程师

申请学位类别 工程硕士 提交学位论文日期 2014年12月

学校代码 10701　　学　号 1203121883
分 类 号 TP31　　密　级 公开

西安电子科技大学

硕士学位论文

Linux 实时调度研究及改进

作者姓名： 吴振亚

领　　域： 计算机技术

学位类别： 工程硕士

学校导师姓名、职称： 苏锐丹副教授

企业导师姓名、职称： 朱军高级工程师

提交日期： 2014 年 12 月

Research and Improvement of Linux Real-Time Scheduling

A thesis submitted to

XIDIAN UNIVERSITY

in partial fulfillment of the requirements

for the degree of Master

in Computer Technology

By

Wu zhenya

Supervisor: Su ruidan Zhu jun

December 2014

摘要

随着信息技术的发展，实时操作系统具有广阔的应用前景，特别是在嵌入式系统领域。Linux 作为一个开源免费的操作系统，凭借其稳定的性能、广泛的适用性以及定制方便等优势，成为实时操作系统的一个理想选择。Linux 作为通用分时操作系统，在实时性支持方面已得到较大的发展。

本文以优化 Linux 实时性能为目标，详细分析了 Linux 可抢占内核以及 Linux 提供的进程调度算法，对已有的实时改进技术进行研究对比，并针对应用模式提出一种实时改进算法。本文所取得的主要研究成果主要有以下几方面：

1.对 Linux 内核的可抢占性进行研究。分析了 Linux 为了增强实时性而对内核源码进行的修改。为了实现可抢占，内核主要做了以下修改：在 thread_info 结构体中添加了一个抢占计数器，用来决定能否抢占；修改内核自旋锁 spinlock 代码段，使进程处于自旋锁保护的互斥区中时不可抢占。

2.对 Linux 提供的实时调度算法进行分析。解释了 Linux 调度器通过调度类的方式，提供多种进程调度策略，并详细分析了完全公平调度算法、实时调度算法以及调度中的关键数据结构和函数。

3.对现有 Linux 实时改进技术进行研究。主要包括针对调度框架、调度算法延迟、调度算法本身和内核接口四个方面的改进：对 Slot-Based Task-Splitting 调度框架与通用调度框架进行分析对比；分析 RT patch 为改进调度算法延迟而做的修改，包括中断线程化、优先级继承等；从问题模式的角度，对已有的硬实时、软实时调度算法进行分析比较；分析对比 RTAI、Xenomai 这类内核接口 API 改进方案。

4.针对一种特定的典型上层应用模式，提出一种实时性改进方案。首先，使用基于关联度的算法，由用户来对具有关联度的进程进行设置，将它们放在指定的 CPU 核运行。其次，在内核 SCHED_RR 调度策略的基础上，提出 Weighted Round Robin 调度策略，采用权重参数来指定时间片大小。这样采用 SCHED_WRR 调度策略的实时进程，不再是固定的时间片，而是根据进程代码中设置的权重来得到它的时间片大小。最后，测试并验证该实时调度算法所实现的功能，并与 SCHED_RR 调度策略进行性能比对，结果显示，在特定的应用模式下，该算法确实能提高实时性。

关 键 字：Linux 内核， 调度算法， 实时， 抢占

论文类型：技术报告

ABSTRACT

With the development of information technology, real-time operating system has a broad application prospect, especially in the field of embedded system. As a free open-source operating system, linux has become an ideal choice for real-time operating system due to its stable performance, wide applicability and easy customization. As a common time-sharing operating systems, Linux has been extensively developed in the aspect of real-time support.

In order to optimize linux real-time performance, this paper analyzes Linux preemptible kernel and process scheduling algorithms, compares the existing technologies of improving linux real-time performance, and proposes a real-time optimization algorithm for a specific application pattern. The main results obtained in this paper are outlined as follows:

1. Preemption of Linux kernel has been studied. The modifications of Linux kernel source code aiming to enhance real-time performance have been studied. In order to achieve a preemptive kernel, the main modifications of kernel source code has been done in the following aspects: a preemptive counter is added in thread_info structure, which is used to determine whether preemption is performed; kernel spinlock code is modified to assure that the process in critical section of kernel does not allow to be preempted.

2. The real-time scheduling algorithms provided by Linux have been analyzed. Linux scheduler offers a variety of process scheduling policies implemented by scheduling classes. Completely fair scheduling algorithm, real-time scheduling algorithm and related critical data structure and functions are also analyzed in detail.

3. The technologies of improving real-time performance have been studied. The improvement work is focused on scheduling framework, scheduling algorithm delay, scheduling algorithm itself and kernel interface： analyzes and compares the Slot-Based Task-Splitting scheduling framework and common framework; studies the modifications of RT-patch which is aiming to reduce delay, including threaded interrupt

and priority inheritance; compares the existing hard real-time, soft real-time scheduling algorithms from the perspective of the problem schema; reviews the optimization scheme of kernel interface API, such as RTAI and Xenomai.

4. An optimization scheme of real-time based on a particular application pattern has been presented. First, associated processes have been set to a specifed cpu core by users based on the correlation algorithm. Second, Weighted Round Robin scheduling policy has been proposed based on SCHED_RR scheduling policy, which can set time slice according to weight parameter. Therefore, the time slice of real-time process with SCHED_WRR scheduling policy is no longer fixed, but calculated by weight parameter defined in the process code. Finally, the function implemented by the real time scheduling algorithm has been tested and verified. Comparing with SCHED_RR scheduling policy, the result shows that SCHED_WRR scheduling policy has a better real time performance under the specific application pattern.

Keywords: Linux kernel, scheduling algorithm, real time, preemption

Type of Dissertation: Technical Report

插图索引

表格索引

目录

第一章 绪论

1.1 课题来源和研究背景

1.1.1 课题来源

本课题源自于国家“核高基”科技重大专项，2012 课题：开源操作系统内核分析和安全性评估支持（2012ZX01039-004）。

1.1.2 研究背景

从上个世纪 90 年代末开始，Linux 这个类 Unix 操作系统家族中的新成员就开始非常流行，并跻身于一些较知名的商用 Unix 操作系统之列。

Linux 作为一个开放源代码的操作系统，由 Linux 开源社区不断的维护和发展，集结了全世界的 Linux 开发者和爱好者的成果，不断取得进步。Linux 是开源的，它的所有成分都可以自由定制，通过内核编译配置，可以自由的选择自己真正需要的特征来编译内核。同时 Linux 对硬件的要求比较低，因此广泛用于嵌入式设备中。同商业化的操作系统相比，Linux 遵循 GPL，并且可以免费安装，具有很强的竞争力。

而目前实时操作系统，在信息技术快速发展的情况下，已经有了很大的需求。与一般的操作系统相比，实时操作系统的最大特色就是其实时性，如果有一个实时任务需要执行，那么系统会马上执行它，不会有很大的延迟。设计实时操作系统的首要目标不是高的吞吐量，而是保证任务在特定时间内完成。实时操作系统可分为硬实时和软实时两类：硬实时操作系统必须使任务在确定的时间内完成，而软实时操作系统能让绝大多数任务在确定时间内完成。系统从接收一个实时任务，到完成该任务所需的时间，其时间的变化称为抖动。硬实时操作系统比软实时操作系统有更少的抖动。

传统的商用嵌入式系统虽然实时性能可以满足嵌入式领域对实时操作系统的要求，但由于价格昂贵，使其应用受到自身的限制。因此 Linux 凭借其强大的功能、广泛的平台支持、充分的自由定制裁剪、丰富的开发工具支持以及非常低廉的价格等优势，渐渐成为嵌入式操作系统的优先选择。较好的实时性、系统可靠性、任务处理随机性是嵌入式操作系统追求的目标。而 Linux 作为通用分时操作系统，在实时性方面的表现越来越受到重视。

由于 Linux 在实时性方面存在着不足，因此对 Linux 的实时性进行分析研究具有重大意义。

1.2 研究现状

Linux 的开放性和低成本是实时 Linux 发展的优势。在 Linux2.5.4 以前的版本中，Linux 内核是不可抢占的，也就是说，如果当前有有一个进程运行在内核态，这时即使当前有一个优先级更高的进程需要运行，当前进程也不能被抢占。优先级更高的进程必须等到当前进程完成内核态的操作返回用户态后或者被某些事件阻塞而主动让出 CPU 后才能被考虑执行，这时进程的抢占延迟就比较明显。因此为了增强实时性，在 Linux2.5.4 之后的内核中，内核已经支持抢占。但是即使内核支持抢占，内核中仍有一些的区域是不可以抢占的，如由自旋锁（spinlock）保护的互斥区，以及一些显式使用 preempt_disable 失效抢占的互斥区。除此之外，为了得到更好的实时性能，已经有很多的研究机构和商业团体进行了实时 Linux 的研究与开发。

目前在 Linux 上已经有多种实时改进技术，一种是将 Linux 作为微内核与 RTAI、Xenomai 等实时子系统并行运行在硬件上；另外一种是对 Linux 内核源码修改，减少通用调度算法的延迟，例如在内核中使用优先级继承来解决优先级反转问题；各研究机构还提出各种适用不同场景的硬实时调度算法，如 EDF 调度算法。并且对 Linux 实时操作系统也提供了对应的 API 接口，便于设备驱动程序及应用程序的开发[1]。

本文为改善 Linux 实时性，首先对 Linux 的软实时内核进行研究，并分析内核中提供的实时调度算法以及当前已有的实时改进技术。在此基础上，本项研究从不同的角度，针对特定的上层应用模式提出一种实时性改进方案，对 Linux 中提供的实时调度算法进行针对性的修改，测试对比 Linux 内核中原有的调度算法，说明达到了增强性能的效果。

1.3 课题主要内容和文章结构

本文主要以 Linux 主线内核版本 3.5.4 源码为分析基础，论文中所列出的所有内核源码都是从 3.5.4 版本中节选出来的。

本文通过阅读相关文献以及内核源码，对 Linux 内核的实时调度和内核的可抢占性进行了阐述和分析，重点对其中相关的调度策略和数据结构进行了分析。并对当前在 Linux 上的实时改进技术进行了分类对比分析。最后，针对特定的应用模式，并根据已有的实时调度算法，提出一种新的实时调度算法，使得在这种特定的应用条件下的，应用的实时性能得到提高。

本论文主要从五个部分来阐述 Linux 实时调度算法的研究、分析与改进。

第 1 章绪论，主要是介绍本课题的研究背景、当前的研究现状还有本文的主

要工作。

第 2 章主要是对 Linux 内核的可抢占性进行分析，内核是如何通过提供抢占来增强实时性的，以及相关的重要数据结构与函数。

第 3 章是对内核的进程调度框架以及算法进行分析，从主要的数据结构出发，阐述了 Linux 怎样通过调度类提供调度算法，并对 CFS 调度算法和实时调度算法进行了分析。

第 4 章是对一些当前已经实现了的 Linux 实时改进技术进行分类分析，并针对问题模式来进行研究对比。

第 5 章提出特定的应用模式，并针对这种特定应用模式改进 Linux 调度算法，并对该算法进行测试验证，可以一定程度上增强实时性能。

最后第 6 章，对本文所做的工作进行总结，并指出本文工作的不足。

第二章 Linux 内核可抢占性分析

2.1 Linux 进程概述

进程是操作系统中最基本的概念，通俗的说，进程是一个正在执行中的程序实例。但进程不只是程序，它是在系统中动态运行的，还需要其他资源的支持，例如，进程中存放临时变量的栈，程序计数器，能够动态分配内存的堆以及保存全局变量的数据区等等，因此进程还有一个与当前状态相关的系统资源集[2]。进程是操作系统管理和分配系统资源的最基本单位。

2.1.1 进程描述符

内核为了方便管理进程，需要了解进程的实时状态，因此必须要有一个数据结构来对与进程相关的数据或事件进行记录，这就是进程描述符 task_struct。该结构中记录与进程相关的所有信息，例如，进程当前的状态是运行还是被阻塞，进程当前打开的文件，进程的优先级等等[3]。下面对其中一些关键成员进行分类说明。

1) 进程状态

表 2.1 常见进程状态说明

进程状态	数值	说明
TASK_RUNNING	0	可运行状态：进程当前在可运行队列中，可能正在运行，可能等待运行。
TASK_INTERRUPTIBLE	1	可中断等待状态：进程被阻塞，需要等待特定事件到达，事件到达后内核会把进程设置为可运行状态。
TASK_UNINTERRUPTIBLE	2	不可中断等待状态：进程被阻塞，与上一个状态相似，不同之处在于事件到达后并不会改变它的状态。这个状态通常在进程必须在等待时不受干扰或等待事件很快就会发生时出现。
__TASK_STOPPED	4	停止状态：进程暂停执行，进程不能投入运行。
__TASK_TRACED	8	跟踪状态：被其他进程跟踪的进程。
EXIT_ZOMBIE	16	僵死状态：进程终止执行，处于“僵尸”状态，进程已经死了，其资源已经释放，无法再运行，但是进程表中依然有对应的表项，等待父进程调用 wait4()系统调用来确认子进程的终结，释放子进程保留的资源。
EXIT_DEAD	32	僵死撤销状态：父进程确认了对子进程的终结，系统将删除进程。

进程描述符 task_struct 结构体定义在内核源码 include/linux/sched.h 中。

在 task_struct 结构体中，使用 state 成员来表示进程当前的状态。state 属于 long int 型，而进程状态被划分为不同的数值，因此可根据它的数值来得到进程的状态，并且进程状态之间是互斥的，进程同时只能设置一种状态，不能同时有多种状态。而每种状态的数值都是 2 的整数次幂，这样的话，只要对 state 的比特位进行检查就可知道进程的状态，查找效率更高。在内核源码 include/linux/sched.h 中有关于进程状态的宏定义。

表 2.1 中列出了常用了进程状态，数值表示在 Linux 内核中用来表示该状态的方式，通过对比位来确定进程的状态，并对常用的进程状态进行了解释说明。

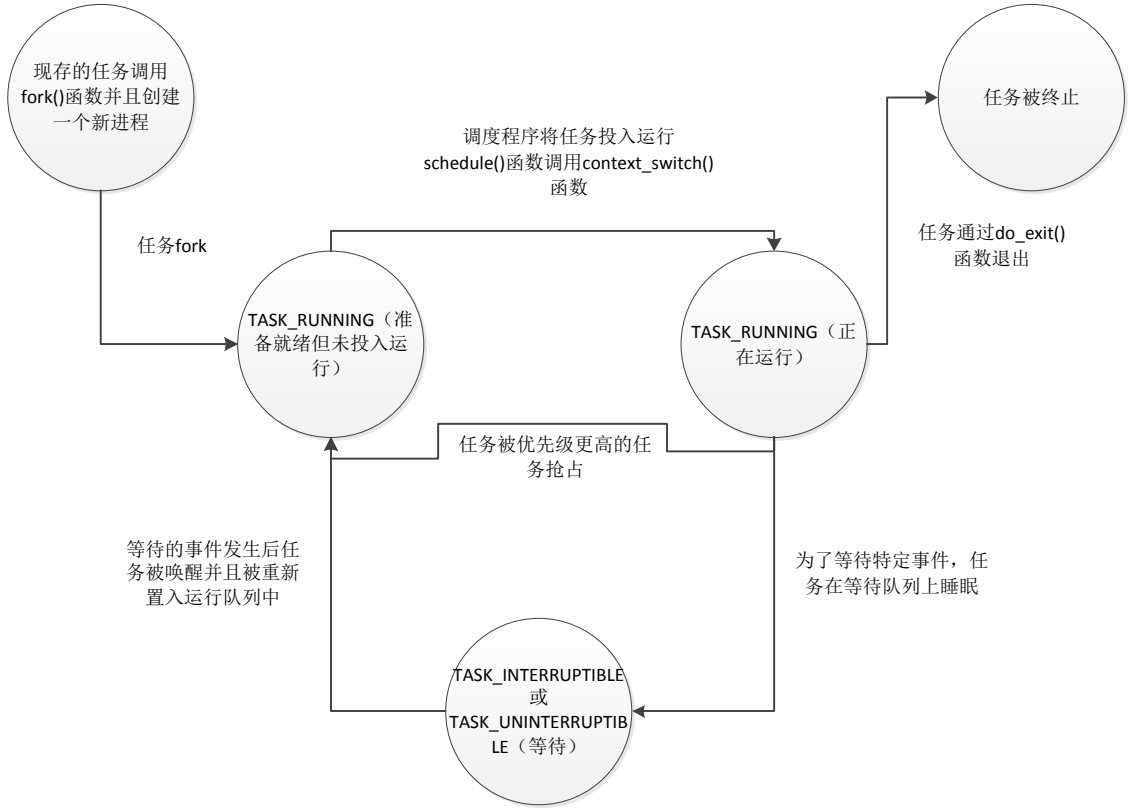


图 2.1 进程状态转移图

如图 2.1 所示，列出了常见的进程状态之间的迁移以及一般发生转换的条件。

2) 进程调度相关成员

1. 进程优先级相关成员（static_prio，normal_prio，prio，rt_priority）

static_prio 用于保存静态优先级，是进程启动时分配的优先级。该成员值可以通过系统调用修改（如 nice 函数），否则在进程运行期间会一直保持恒定。

normal_prio 表示常规优先级，根据 static_prio 和调度策略计算，计算函数为 normal_prio()。进程分支时，子进程会继承常规优先级。

prio 表示动态优先级，调度器使用的优先级就是 prio。由于某些情况下内核需要暂时提高进程的优先级，因此需要第 3 个成员来表示。这样的话，不对其他优先级产生影响。

rt_priority 用于保存实时进程的实时优先级，值范围[0,99]。值为 0 表示是非实时任务，1-99 表示实时任务，值越大，优先级越高。

一般对于实时进程：prio = normal_prio = MAX_RT_PRIO – 1 – rt_priority

一般对于非实时进程：prio = normal_prio = static_prio

因此在内核中使用的是 prio 表示优先级，因此图 2.2 显示了内核中的优先级标度，在调度器中，prio 值越低优先级越高，0-99 表示实时进程。普通进程中 nice 值[-20，20]映射到[100，139]。

图 2.2 内核优先级标度

2. 调度类与调度实体（sched_class，se，rt）

sched_class 结构体表示调度类，指向进程所属的调度类。在第三章 3.1.1 节中会详细介绍。

se 是普通进程的调度实体，用于普通进程。rt 则是实时进程的调度实体，用于实时进程。每个进程描述符中都包含这两个的实体。这样调度器可以操作比进程更一般的实体，每个进程描述结构中都嵌有调度实体结构，但并不是一个调度实体就等于一个进程，有时候将一组进程集合起来构成一个调度实体。调度实体结构将会在第三章 3.1.3 节中介绍。

3. 进程的调度策略（policy）

policy 用来指示进程的调度策略，包括了普通进程常用的 SCHED_NORMAL 调度策略和实时进程用的 SCHED_FIFO 和 SCHED_RR 调度策略，通常调度策略与调度类相对应。

表 2.2 中列出了进程常用几种调度策略，并对调度策略进行了解释说明。policy 通过数值来识别不同的调度策略。

表 2.2 常见进程调度策略说明

调度策略	数值	说明
SCHED_NORMAL	0	普通进程的调度策略，由完全公平调度类处理。
SCHED_FIFO	1	实时调度策略，先入先出调度算法。用于实现软实时进程，由实时调度类处理。
SCHED_RR	2	实时调度策略，轮转调度算法。用于实现软实时进程，由实时调度类处理。
SCHED_BATCH	3	用于非交互的处理器消耗型进程，由完全公平调度类处理。这类进程决不会抢占 CFS 调度器处理的另一个进程，因此不会干扰交互式进程。如果不打算用 nice 降低进程的静态优先级，同时又不希望该进程影响系统的交互性，此时最适合使用该调度类。
SCHED_IDLE	5	与 SCHED_BATCH 相似，用于系统负载很低时。

3) thread_info 结构体

thread_info 保存了体系结构的汇编语言代码需要访问的那部分进程数据[4]。该结构体的具体定义依赖于特定的体系结构，但是内容基本类似。表 2.3 列出了该结构体中的常用成员及含义。

表 2.3 thread_info 结构中常见成员说明

结构体成员	说明
task_struct* task	表示指向进程 task_struct 实例的指针
__u32 flags	该变量可以保存各种特定于进程的标志。其中两个特别感兴趣：如果进程有待决信号则置位 TIF_SIGPENDING。另一个 TIF_NEED_RESCHED 表示该进程应该或想要调度器选择另一个进程替换本进程执行。标志所对应的数值与平台相关，x86 下 TIF_NEED_RESCHED 为 3。
__u32 cpu	说明了进程正在其上执行的 CPU。
int preempt_count	该成员实现了内核抢占所需的一个计数器。0 表示可抢占，大于 0 表示不能抢占。
mm_segment_t addr_limit	指定了进程可以使用的虚拟地址的上限。该限制适用于普通进程，但内核线程可以访问整个虚拟地址空间，包括只有内核能访问的部分。

thread_info 结构体中 preempt_count 抢占计数器是其最重要的成员。thread_info 存在于进程内核栈中，记录了进程的一些信息，因为该结构中包含指向该进程描述符的指针，因此可以通过该结构体得到正在运行进程的进程描述符。如图 2.3 所示。

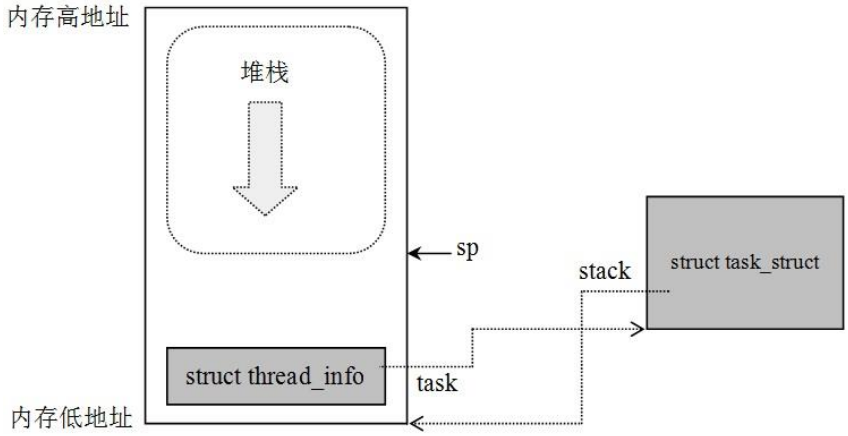


图 2.3 内核栈中的 thread_info 结构

Linux 内核源码所有体系结构中都有两个名为 current_thread_info 和 current 宏或函数，通过使用它们总是可以很方便的获得当前 CPU 上进程的 thread_info 结构体的地址以及进程描述符的地址。其函数说明具体如下所示：

1. current_thread_info 可获得指向当前执行进程的 thread_info 实例的指针。其地址可以根据内核栈指针确定，因为 thread_info 实例总是位于栈顶。因为每个进程分别使用各自的内核栈，进程到栈的映射是唯一的。
2. current 给出了当前进程 task_struct 实例的地址。该函数在源代码中出现非常频繁。该地址可以使用 current_thread_info() 确定：current=current_thread_info()->task。

2.2 可抢占内核概述

一般 Linux 进程管理中有两种进程执行模式：用户态和内核态。采用两种模式，这样就可以保护操作系统及重要的操作系统数据不受用户程序的干涉。内核态指的是操作系统的内核，这是操作系统中包含重要系统功能的部分[5]，进程在该模式下具有无限的权力。而在用户态下，进程只能访问自身的数据，无法干扰系统中的其它应用程序进程。进程通常都处于用户态。

进程如果想要访问系统数据或功能，就需要通过系统调用切换到内核态。系统调用就是在系统提供给用户切换到内核态的一种方式。系统调用是由用户进程

主动调用的。

还可以通过中断的方式由用户态切换到内核态。与系统调用不同不同，中断的发生一般是无法预测的。中断发生后，通常会有对应的中断处理程序来处理中断的操作。

普通进程在用户态运行时，总是可能被抢占。当一个优先级更高的进程变为可运行时，调度器可以决定是否立即执行该进程，即使当前进程仍然在正常运行。这种抢占对于实现良好的交互行为和低系统延迟有重要的作用。

在 2.5.4 之前，Linux 内核是不可抢占的，即当内核中有一个进程正在运行，除非该进程主动放弃 cpu 的使用权，否则高优先级的进程不可以抢占当前进程 cpu 的使用权，直至当前进程执行完毕或者退出内核态。但是进程退出内核态的时间未知，这对实时应用来说是不可接受的。

因此在 2.6 版的内核之后，内核引入了抢占能力。所谓可抢占内核，即当某个进程运行在内核态时，高优先级的进程变为可运行状态，如果当前内核允许抢占，执行抢占，运行高优先级的进程。内核抢占可以减少这样的等待时间，因而保证“更平滑的”程序执行，但该特性的代价是增加内核的复杂度，因为接下来有许多数据结构需要针对并发访问进行保护，即使是在单处理器系统上也是如此。

可抢占内核使 Linux 的实时性大大增加，为 Linux 应用于实时系统奠定了基础。在实时操作系统中，通常会给实时任务指定一个最后的期限，实时任务必须在限定的期限之前执行完成，因此当它被唤醒时，应该在限定时间内被调度执行。但是由于 Linux 内核是不可抢占的，在内核中无法确定需要执行的时间，该实时任务需要等到内核中运行的任务退出内核才能抢占，从而会产生延迟。因此不可抢占的内核不能满足实时任务的响应要求。Linux 实现可抢占的内核改善了对多媒体这种软实时应用的支持不好的缺点，并促进了 Linux 在桌面操作系统方面的发展[6]。

2.3 抢占原理和基础函数

Linux 为了实现内核可抢占对内核源码所做的第一个修改，就是在进程的 thread_info 结构体中加入一个抢占计数器成员 preempt_count。初始值为 0，当进程使用自旋锁时对其数值加 1，说明进程进入了互斥区，释放自旋锁的时候数值会减 1，说明退出互斥区。当其值为 0 的时候，内核才可以执行抢占。每次在内核中发生调度时机时，例如从中断程序返回的时候，如果当前有抢占请求，内核会检查 preempt_count 的值，如果值等于 0，说明内核当前是抢占安全的，内核就会执行抢占。如果值不等于 0，表示当前任务处在互斥区中，此时抢占是不安全的，所以不被允许。这样内核就不会执行调度程序，而是直接从中断返回到当前执行进程

中。

而内核中的抢占请求或者说是重调度请求由 thread_info 成员 flags 中给出，TIF_NEED_RESCHED 是重调度的标志，如果 flags 设置了该标志的话，说明需要重新调度。其中判断 flags 中是否设置了 TIF_NEED_RESCHED 标志可以通过 need_resched()函数。如图 2.4 所示，该函数将对 flags 标志进行比对。

```
static inline int need_resched(void)
{
    return unlikely(test_thread_flag(TIF_NEED_RESCHED));
}
```

图 2.4 判断重调度标志函数定义

内核抢占共分为两种：主动抢占和自愿抢占。主动抢占是指：只要正在占据 CPU 运行的低优先级的进程可以被抢占，那么，此时若存在一个就绪的高优先级的进程，则该高优先级的进程就会抢占低优先级的进程；自愿抢占是指：进程自愿放弃 CPU。

2.3.1 主动抢占

主动抢占技术的实现关键难点在于：抢占进程抢占后不会破坏内核临界区执行的正确语义。eg: 设 P1 抢占 P2 进程，而此时 P2 正在内核中的某个临界区 C（如内存分配）中，即 P2 目前持有该临界区的锁 L，而此时 P1 若不等 P2 安全释放 L，直接抢占，那么若此时 P1 依然同样执行临界区 C，最后造成的结果就是死锁。为了解决上述问题，Linux 对每个线程使用一个计数器，即 preempt_count，该计数器记录了该线程获得锁的数量。只有当被抢占线程所持有锁的数目为 0 时，抢占才可能发生。需要指出的是：preempt_count 的设计并未采用简单的布尔类型，而是使用整型。因为内核可能通过不同的入口点进入多个关键区，从而可能发生关键区嵌套，而在内核再次启用抢占之前，必须确认已经离开所有的临界区，所以若仅使用布尔类型区分当前进程/线程是否在一个关键区中运行是非常困难的。

preempt_count 结构是在 struct thread_info 结构体中。该结构在前面介绍进程时已经介绍过了，不再描述。每次当一个线程/进程，需要对一个关键区加锁时，加锁操作会对 preempt_count 值加 1。

为了避免不一致的发生，该值不能直接操作，只能通过函数 dec_preempt_count 和 inc_preempt_count 来对 preempt_count 的值分别进行加 1 和减 1 操作。每次内核进入某些区域需要禁止抢占时，就会调用 inc_preempt_count。退出该区域时，则调用 dec_preempt_count。

1) 处理抢占计数器的宏和相关函数

1. preempt_count()

该宏很简单，用来在 thread_info 描述符中选择 preempt_count 字段，可以很方便的读取到当前进程的抢占计数器的值。具体如图 2.5 所示：

```
#define inc_preempt_count() add_preempt_count(1)
#define dec_preempt_count() sub_preempt_count(1)

#define preempt_count() (current_thread_info()->preempt_count)
```

图 2.5 preempt_count 宏定义及加减操作

其中 inc_preempt_count 对抢占计数器做加 1 操作，dec_preempt_count 做减 1 操作。将会在其他宏中使用。

2. preempt_disable()

preempt_disable()使抢占计数器的值加 1，停止抢占，并在其后设置一个优化屏障。具体代码如图 2.6 所示：

```
#define preempt_disable() \
do { \
    inc_preempt_count(); \
    barrier(); \
} while (0)
```

图 2.6 preempt_disable 的宏定义操作

3. preempt_enable_no_resched()

preempt_enable_no_resched()使抢占计数器的值减 1，但是不进行重调度。其中开始时也插入了一个优化屏障。

```
#define sched_preempt_enable_no_resched() \
do { \
    barrier(); \
    dec_preempt_count(); \
} while (0)

#define preempt_enable_no_resched() sched_preempt_enable_no_resched()
```

图 2.7 preempt_enable_no_resched 宏定义操作

4. preempt_enable ()

```
#define preempt_enable() \
do { \
    preempt_enable_no_resched(); \
    barrier(); \
    preempt_check_resched(); \
} while (0)
```

图 2.8 preempt_enable 的宏定义操作

如图 2.8 所示，preempt_enable()通过调用 preempt_enable_no_resched()，使抢占计数器的值减 1，启用内核抢占，然后通过调用函数 preempt_check_resched()检

查在 thread_info 描述符的 TIF_NEED_RESCHED 标志是否被设置。如果设置了的话，进程切换请求是挂起的，preempt_schedule()函数将会被调用。

5. barrier()

在以上处理抢占计数器相关宏代码中大都包含了一个重要的函数 barrier()，该函数的作用是插入一个优化屏障，使得编译器必须先处理完屏障之前的读写请求，才能对屏障之后的请求进行处理。使用该函数的原因是现代编译器和处理器都会试图对代码进行优化，例如指令重排，这会提供程序运行时的性能。但是编译器或处理器往往很难判定重排的结果是否确实与代码原来的意图匹配。可能会导致错误的结果，而该影响也很难发现。因此在内核抢占机制中使用优化屏障就是为了防止抢占的时机因为代码优化而出现偏差。

在 preempt_disable()和 preempt_enable()或者 preempt_enable_no_resched()中间的代码应该是不能抢占的，在没有使用优化屏障 barrier()的时候，代码如果如下：

```
preempt_disable();
function_which_must_not_be_preempted();
preempt_enable();
```

那么编译器对代码优化可能代码重新排序如下：

```
function_which_must_not_be_preempted();
preempt_disable();
preempt_enable();
或者
preempt_disable();
preempt_enable();
function_which_must_not_be_preempted();
```

那么在上述两种情况下，都会把原本希望不可抢占的部分变得可抢占，因此必须在可抢占机制中使用优化屏障。preempt_disable()中在抢占计数器加 1 之后插入一个内存屏障，preempt_enable_no_resched()中在再次启用抢占之前出入一个优化屏障。

6. preempt_check_resched()

```
#define preempt_check_resched() \
do { \
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
        preempt_schedule(); \
} while (0)
```

图 2.9 preempt_check_resched 的宏定义操作

如图 2.9 所示，该宏定义用来检查在 thread_info 结构体成员 flags 中的 TIF_NEED_RESCHED 标志是否被设置，如果设置的话将会调用 preempt_schedule() 函数，看是否需要进行重新调度。

由于设置了 TIF_NEED_RESCHED 标志，并不一定保证可以抢占内核，内核可能正处于临界区，所以 preempt_schedule()函数对 preempt_count 进行判断，决定是否能进行抢占，可以的话再进行重新调度。该函数在 kernel/sched/core.c 中定义，如图 2.10 所示：

```
asmlinkage void __sched notrace preempt_schedule(void)
{
    struct thread_info *ti = current_thread_info();

    /*
     * If there is a non-zero preempt_count or interrupts are disabled,
     * we do not want to preempt the current task. Just return..
     */
    if (likely(ti->preempt_count || irqs_disabled()))
        return;

    do {
        add_preempt_count_notrace(PREEMPT_ACTIVE);
        __schedule();
        sub_preempt_count_notrace(PREEMPT_ACTIVE);

        /*
         * Check again in case we missed a preemption opportunity
         * between schedule and now.
         */
        barrier();
    } while (need_resched());
}
```

图 2.10 preempt_schedule 函数定义

该函数先对 preempt_count 的值进行判断，如果该值不为 0 或者该中断是关闭的，那么说明当前进程是不允许抢占的，直接返回。其中__schedule()是关键函数，为主调度函数。

值得注意的是，在使用调度器函数之前，将 preempt_count 设置为一个很大的值，该值用宏 PREEMPT_ACTIVE 表示。__schedule 函数通过该标识，知道调度由于内核抢占引发的，而不是普通方式。调度器函数结束后，要将抢占计数器减去该值，保持前后一致。并在 while 中会再次对抢占标志进行检查，以免在 schedule 和当前点之前错过了抢占的时机。

2) 主动抢占的时机分类

内核主动抢占时机分类有如下几种：

1. 当从中断处理程序正在执行，且返回内核空间之前。

这是最常见的内核主动抢占时机，最常见的中断处理程序是：时钟中断处理

程序。

2. 当内核代码再一次具有可抢占性的时候，如解锁及使能软中断等。

在这种情况下，调用的是 preempt_schedule()。例如在解锁的过程中，会调用 preempt_enable()，该函数中会调用 preempt_schedule()来判断是否进行抢占。

例如为了实现可抢占，要对用于保护互斥区的自旋锁进行修改，在加、解锁代码中加入对抢占计数器的操作。

自旋锁加锁函数 spin_lock()将对抢占计数器进行加 1 操作，具体如图 2.11 所示：

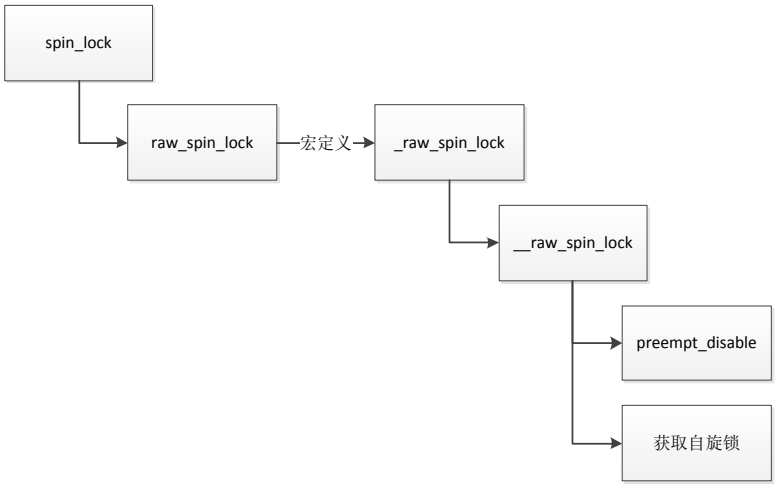


图 2.11 自旋锁加锁函数调用关系图

加锁函数进行了多次嵌套，最后在 include/linux/spinlock_api_smp.h 中的 __raw_spin_lock 函数中执行真正的操作，函数中的第一个操作就是 preempt_disable()，对抢占计数器加 1，来表明此时内核禁止抢占，然后再获取自旋锁。

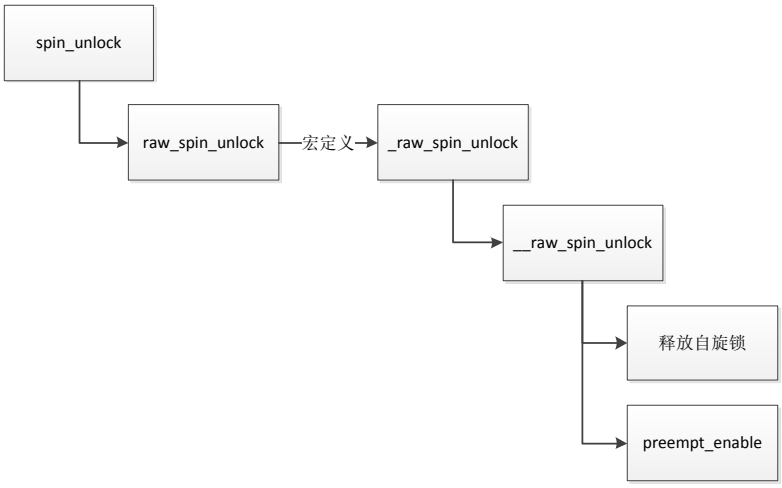


图 2.12 自旋锁解锁函数调用关系图

从图 2.12 可知，解锁函数的嵌套与加锁函数类似，不过解锁操作与加锁函数中相反。先释放自旋锁，随后使用 preempt_enable()对抢占计数器减 1，该宏将调用 preempt_check_resched()来判断当前是否能够抢占，是否需要进行重新调度。

因此，每次当进程退出用自旋锁保护的互斥区时，都会有一个抢占的时机，会判断当前是否需要抢占。

2.3.2 自愿抢占

自愿抢占的核心思想是：进程/线程执行某一段代码之前，若该代码段需要耗费大量的时间（Eg: 在 linux 内核代码中有大量的地方都通过调用 schedule()函数来进行重新调度，但是并不是每次都可以成功的实现进程的重新调度，比如：自旋锁被低优先级的进程 A 持有，而被调度的高优先级进程 B 需要该自旋锁 ，那么将会导致 B 长时间的等待），那么，在执行该代码段之前，主动放弃 CPU，让位于更高优先级的进程。一般自愿抢占的发生的时机包括：

1. 内核中的任务显式的调用 schedule()；
2. 内核中的任务阻塞，主调度函数也会被调用。

Linux 内核中的自愿抢占是靠 might_sleep()函数实现的，在执行需要耗费大量的时间代码段之前，Linux 内核会调用 might_sleep()，尝试自愿抢占。might_sleep 函数会去判断当前的上下文是否是一个原子操作，如果是原子操作，就会打印出错误信息。

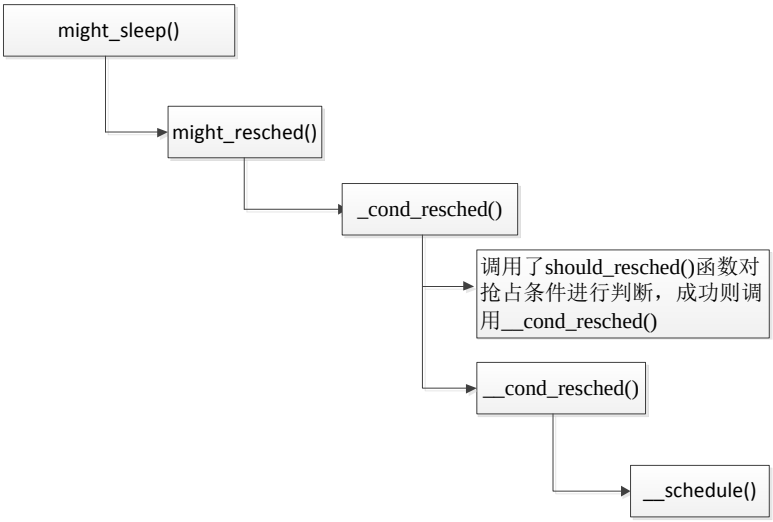


图 2.13 might_sleep 函数调用关系图

只有在定义了 CONFIG_PREEMPT_VOLUNTARY 的情况下 might_sleep()函数会形成一个显式的抢占调度点。如图 2.13 所示，该函数主要调用了 might_resched()函数。

在没有定义 CONFIG_PREEMPT_VOLUNTARY 时，mignt_resched()函数不做任何事情，如果定义了 CONFIG_PREEMPT_VOLUNTARY，might_sched()函数就会调用 _cond_resched()函数。

而在_cond_resched()函数中，调用了 should_resched()函数对抢占条件进行判断：

在 shoud_resched()函数中判断是否满足重新调度的条件：

```
static inline int should_resched(void)
{
    return need_resched() && !(preempt_count() & PREEMPT_ACTIVE);
}
```

先检查进程是否有重调度标识，同时 should_resched 再检查抢占计数器是否为 0，当前是否可抢占。如果满足的话再调用调度核心函数__schedule()，该函数将在第三章 3.2.1 节中重点说明。

第三章 Linux 内核进程调度算法分析

3.1 关键数据结构

Linux 是一个多任务分时操作系统，操作系统的关键任务就是管理系统资源，并调度各种进程使用这些资源，决定将哪个进程投入运行。因此进程调度就是一个基本的操作系统研究问题。下面先来介绍与调度相关的关键数据结构。

3.1.1 调度器类

Linux 通过调度器类实现调度器的模块化，调度器类就像是程序中的接口，特定的调度器类实例中实现该接口，将接口暴露给通用调度器，通用调度器将使用它的接口来处理进程，而该接口的具体实现则与特定的调度器实例相关。每种调度算法的实现都是通过调度器类来进行封装提供，因此通过实现调度器类的方式可以很方便的添加新的调度算法，并且各调度器类之间互不干扰。

在 Linux 内核源码 include/linux/sched.h 中定义了 Linux 调度器类 sched_class 数据结构。该结构中的主要成员都是函数指针，每个函数指针都对应着调度器中的操作。表 3.1 列出了调度类中常用的函数指针。

表 3.1 调度类中常用函数指针

函数指针名	操作说明
enqueue_task	添加一个新进程到可运行队列。例如在进程由其他状态变为可运行状态时，该操作就会发生。
dequeue_task	用来将一个进程从可运行队列删除。比如进程阻塞时，该操作就会发生。
yield_task	进程想要自愿放弃 CPU，使其他进程有机会得到运行。
check_preempt_curr	该函数检查是否需要用一个进程来抢占当前进程。当使用 wake_up_new_task 唤醒新进程时，该函数会被调用。
pick_next_task	该函数用于选择一个最适合的进程接下来运行，返回该进程的进程描述符 task_struct。
put_prev_task	在用另一个进程代替当前运行的进程之前调用该函数。
set_curr_task	该函数用来设置进程的调度策略，如果需要改变进程的调度策略，可以调用该函数。
task_tick	该函数由周期性调度器调用，用来统计周期性工作。

在调度器类结构体 sched_class 中，还定义了一个 next 指针，用来指向的下一个调度器类。每个调度类都会生成一个调度类实例，这样就可以通过该指针将实

现的调度类实例按优先级顺序连接起来。

图 3.1 调度类优先顺序图

如图 3.1 所示，内核中通过 next 指针将以上四种调度类实例联系起来，Linux 在进行调度的时候，会按照优先级顺序查找调度类，从最高优先级的调度类开始，尝试选择下一个将要运行的进程，如果没有，则使用 next 指向的调度类，直到选择出进程为止。从以上顺序中可以看出，实时进程调度类（rt_sched_class）的优先级比普通进程调度类（fair_sched_class）的优先级高，所以调度时，实时进程总是优于普通进程。

3.1.2 可运行队列

可运行队列（runqueue）是用来对可运行状态的进程进行管理的数据结构。每个 CPU 都有自身的可运行队列，各个活动进程只出现在一个可运行队列中。在多个 CPU 上同时运行一个进程是不可能的。

可运行队列是全局调度器许多操作的起点。但要注意，进程并不是由可运行队列的成员直接管理的。这是各个调度器类的职责，因此在各个可运行队列中嵌入了特定于调度器类的子可运行队列。

因为可运行队列 struct rq 中成员比较多，不对其一一介绍。最主要是包含了两个子可运行队列 struct cfs_rq 和 struct rt_rq，分别用于完全公平调度器和实时调度器。nr_running 表示 rq 队列上可运行进程的数目，不考虑其优先级或调度类。curr 指向当前运行的进程的 task_struct 实例。

内核中定义了一些便利的宏，根据对应条件来获取相应的可运行队列。

kernel/sched/sched.h

```
#define cpu_rq(cpu)          (&per_cpu(runqueues, (cpu)))
#define this_rq()        (&__get_cpu_var(runqueues))
#define task_rq(p)           cpu_rq(task_cpu(p))
#define cpu_curr(cpu)        (cpu_rq(cpu)->curr)
```

3.1.3 调度实体

由于调度器可以操作比进程更一般的实体，因此需要一个适当的数据结构来描述此类实体。

include/linux/sched.h

```
struct sched_entity {
```

```
    struct load_weight      load;
    struct rb_node          run_node;
    struct list_head   group_node;
    unsigned int            on_rq;

    u64             exec_start;
    u64             sum_exec_runtime;
    u64             vruntime;
    u64             prev_sum_exec_runtime;
    ……
    };
```

以上就是调度实体 sched_entity 中的主要成员，去掉了一些需要配置才有的成员，各个成员的含义如下。

load 指定了权重，决定了各个实体占队列总负荷的比例。该权重是完全公平调度的关键，进程的虚拟时间需要用它来换算。

run_node 是标准的树结点，使得实体可以在红黑树上排序。

group_node 是组调度中列表的结点。

on_rq 表示该实体当前是否在可运行队列上接受调度。

sum_exec_runtime 用于在 CFS 调度器中，记录进程消耗的 CPU 时间。跟踪时间是由 update_curr 不断累积完成的。调度器中许多地方都会调用该函数，例如，新进程加入可运行队列时，或者周期性调度器中。

vruntime 用来统计进程在执行期间虚拟时钟上消耗的时间。

prev_sum_exec_runtime 用在当进程被撤销 CPU 时，保存当前的 sum_exec_runtime 值。

在每个 task_struct 都嵌入了一个 sched_entity 的一个实例，所以进程是可调度实体。但是可调度的实体不一定是进程。本文中暂时将调度实体和进程视为等同。

而实时进程同样有它的调度实体 struct sched_rt_entity，该调度实体相比普通进程的较为简洁，不再详细列出结构体内容。

该结构中最重要的是 time_slice，它用来保存实时进程的时间片大小，run_list 是链表的一个结点，串联到链表中。

3.2 Linux 调度器概述

Linux 调度器通过使用上述 3.1 节所介绍的以及其他的一些数据结构，对系统

中的进程进行管理和调度。调度器的主要结构和工作方式如图 3.2 所示：

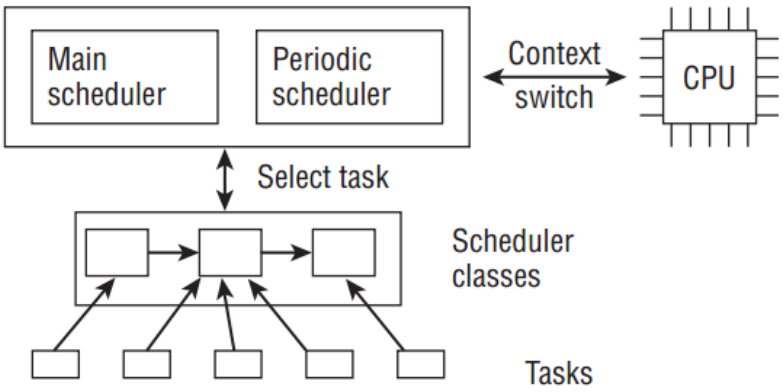


图 3.2 Linux 调度器的结构组成

调度器中主要有两种方法激活调度。一种是直接的，比如进程打算睡眠或出于其他原因放弃 CPU，从而发起调度请求，称为主调度器；另一种是通过周期性机制，按固定的频率运行，并会检查当前是否需要进行进程调度，称为周期性调度器[7]。这两种方法所在组件可以称为通用调度器，这是进程调度时的通用方法，与调度类无关[8]。

从图 3.2 中可以看出，Linux 调度器并不直接操作进程，而是将该操作交给进程所属的调度类来做。每个进程都必须关联到特定的调度类，通过 task_struct 中的 sched_class 成员来记录。调度器通过与调度类交互，由调度类来决定接下来运行哪个进程。

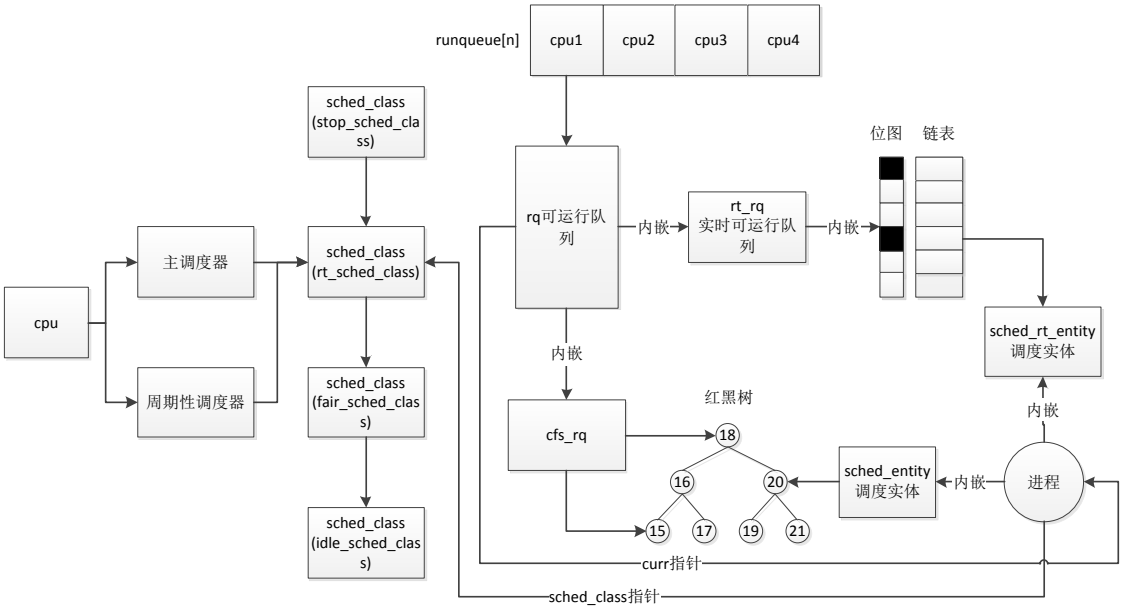


图 3.3 Linux 进程调度数据结构关系图

图 3.3 从整体上描述了 Linux 在进行进程调度时，各数据结构之间的关联关系，关于 CFS 和实时调度将在下面详细描述。

3.2.1 主调度器函数

在内核的许多地方，如果当前进程打算睡眠或者进程要被更高优先级的任务抢占，这时都需要调用主调度器函数 schedule()来处理，使用另一个进程来取代当前进程执行。该函数是 Linux 调度器中的核心函数。

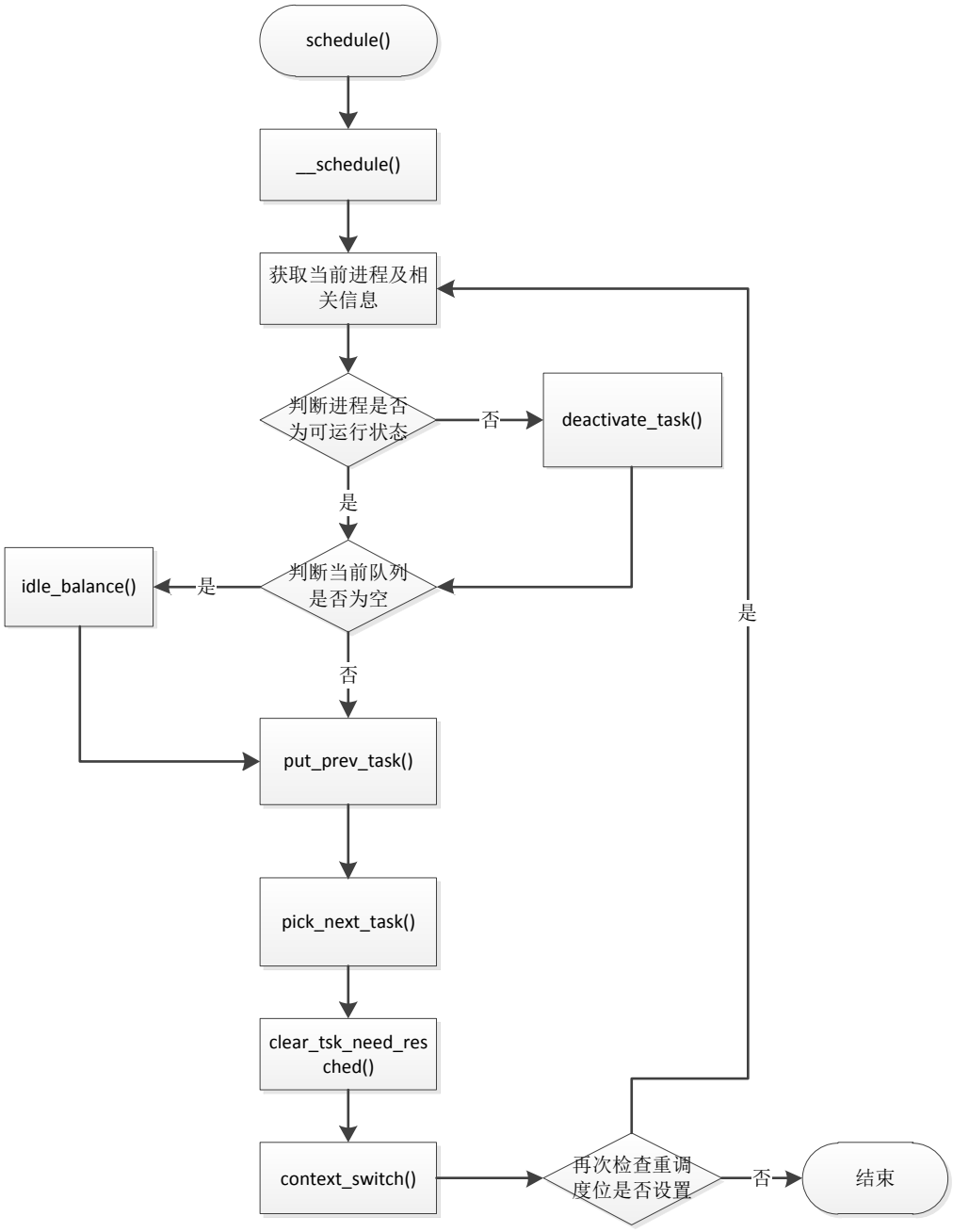


图 3.4 调度主函数 schedule 的代码流程图

图 3.4 为 schedule()的代码流程图，其中所调用的函数大多都会调用当前进程

所在调度类中的函数，由特定的调度类进行操作处理。

调度器会先确定当前可运行队列，并在 prev 中保存一个指向活动进程的 task_struct 的指针。

再判断该调用是否由抢占发起的调度：如果是由抢占发起的重新调度，那么当前运行的进程被抢占后，还将保持在可运行队列中；如果不是由抢占发起的，那么当前进程被抢占后，说明进程的状态发生变化不再是可运行状态，将被移出可运行队列。

schedule 函数中，被调用的函数说明：

1. deactivate_task()是将进程移出可运行队列的函数，该函数实际调用了调度类中的 sched_class->dequeue_task，具体操作依赖于调度类。
2. idle_balance()当前运行队列为空，调用该函数试图从其他的 CPU 运行队列拉来一些进程。
3. put_prev_task()在当前进程被其他进程代替前，需要对它进行一些操作。具体实现依赖进程所属的调度类。
4. pick_next_task()用来选择下一个应该执行的进程。在该函数中，对采用 CFS 调度策略的方法进行优化，如果队列中的可运行进程数量等于 CFS 类对应的可运行进程数。这样的话，直接调用该调度类中的方法 fair_sched_class.pick_next_task。否则的话，将以优先级为序，从最高的优先级调度类开始，遍历了每一个调度类。每一个调度类都实现了 pick_next_task()函数，调用该函数，它将返回指向下一个可运行进程的指针，没有时会返回 NULL，在 4.2.2 节中会详细说明。
5. clear_tsk_need_resched()清除要被替换进程 thread_info 结构体中的重调度标志。
6. context_switch()进行进程的上下文切换，这里不作为关注的重点。

3.2.2 周期性调度器函数

周期性调度器通过 scheduler_tick()函数来实现。该函数将会按照频率 HZ，被计时器代码自动调用。该函数主要有两方面的工作：对内核中与进程调度相关的统计量进行更新；调用当前进程所属调度类中的周期性调度方法。

图 3.5 所示为 scheduler_tick()的函数调用关系层次图。首先该函数会对该可运行队列加上自旋锁，该自旋锁会禁用抢占。

update_rq_clock()，用来对可运行队列时钟进行更新，更新时钟时间戳。

task_tick()的实现取决于特定的调度类。用来检查进程运行的时间，判断当前进程是否应该重新调度，如果需要重新调度，该调度类方法将当前进程 thread_info 结构体中的 flags 设置重调度标志。

scheduler_tick()函数在释放自旋锁时，会对该 flags 中的标志进行检查，决定是否要调用 schedule()。

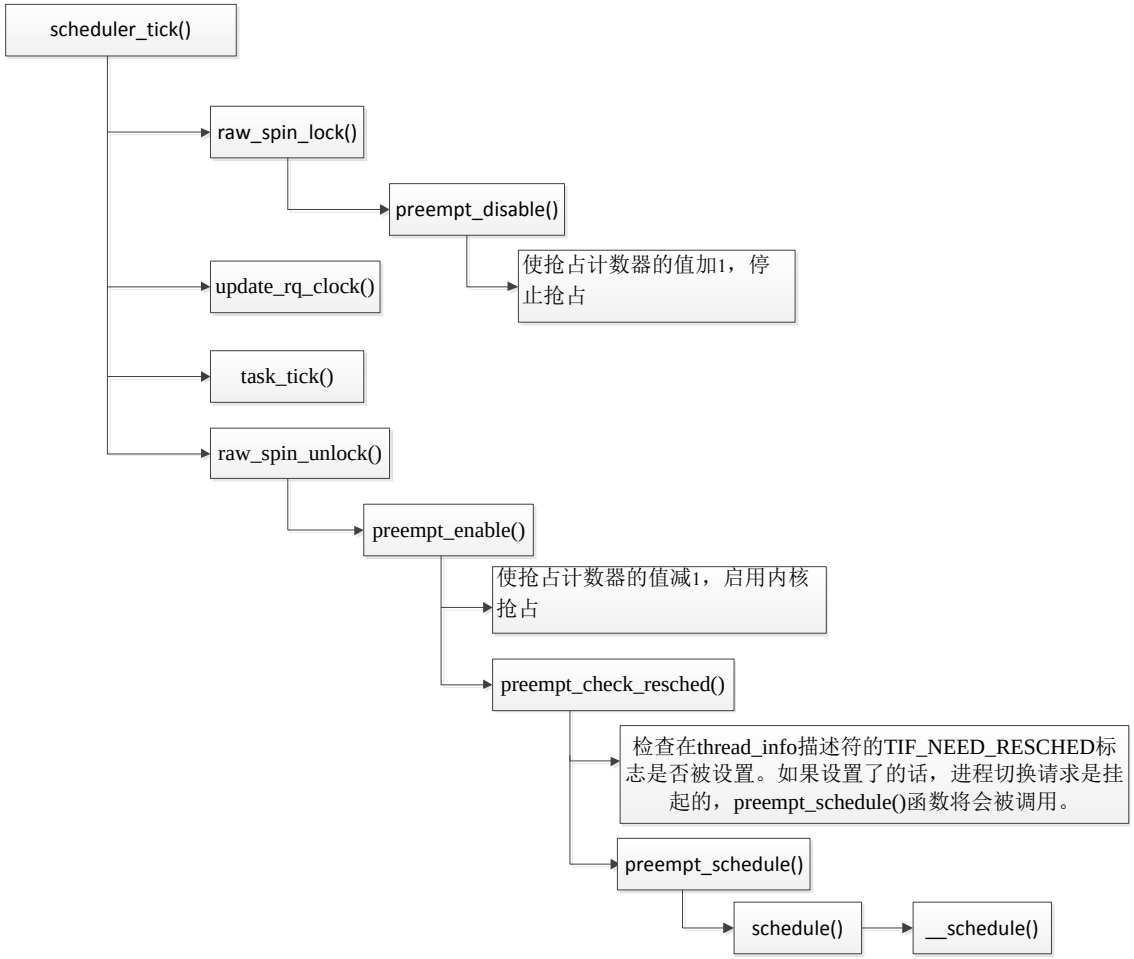


图 3.5 周期性调度函数 scheduler_tick 的函数调用关系图

3.3 CFS 调度算法

CFS（completely fair scheduler）完全公平调度算法是在内核版本 2.6.23 开发期间合并进来的，替代了原有的 O(1)调度算法，希望能模拟完全公平的情况。按照 Ingo Molnar 的说法，CFS 百分之八十的工作可以用一句话概括：CFS 在真实硬件上模拟了完全理想的多任务处理器。在“完全理想的多处理器”下，每个进程都能同时获得 CPU 执行时间[9]。

与之前内核中的调度算法不同，CFS 并没有 O(1)中的 active/expire 数组，而采用红黑树[10]来管理就绪进程。所有的就绪进程都将以其虚拟时间作为键值挂到红黑树的结点上，其中虚拟时间最小的进程将挂在红黑树的最左边叶子节点上。那么在进程调度时，CFS 调度器这个进程作为下一个运行的进程（红黑树最左边叶子节点上的进程）。

3.3.1 CFS 可运行队列

调度器的每个可运行队列中都嵌入了一个该结构的实例，下面介绍一下 CFS 的可运行队列，节选部分关键成员。

kernel/sched/sched.h

```
struct cfs_rq {
 struct load_weight load;
 unsigned int nr_running, h_nr_running;

 u64 exec_clock;
 u64 min_vruntime;

 struct rb_root tasks_timeline;
 struct rb_node *rb_leftmost;
 struct sched_entity *curr, *next, *last, *skip;
……
};
```

关键成员含义如下：

- load 记录队列中进程负荷权重的累积总和。
- nr_running 用来记录队列上可运行进程的数目。
- min_vruntime 跟踪记录队列上所有进程的最小虚拟运行时间。这个值是实现与可运行队列相关的虚拟时钟的基础。该值必须是单调递增的，并且该值会不断更新，每个队列的 min_vruntime 只有被树上某个结点的 vruntime 超出时才更新。
- tasks_timeline 是一个基本成员，用于在按时间排序的红黑树中管理所有进程。
- rb_leftmost 总是设置为指向红黑树最左边的结点，即最需要被调度的进程。
- curr 指向当前执行进程的可调度实体。

3.3.2 CFS 虚拟时间

在 CFS 完全公平调度中，会根据各个进程的权重分配运行时间，计算公式为：

进程的运行时间 = sysctl_sched_latency*weight/所有进程权重之和

sysctl_sched_latency 表示调度周期，一般为可运行队列上就绪进程都调度一遍的时间。weight 指的是进程的权重大小。所有进程权重之和在 cfs_rq 结构体中 weight 保存。

而在完全公平调度中，普通进程优先级会有对应的权重值，该对应关系保存

在 prio_to_weight 数组中。该数组大小为 40，数组下标 0—39 对应非实时进程的优先级-20—20，值越小说明优先级越高，权重值也越大，也就是说优先级为-20 的权重最大，值为：88761，优先级为 20 的权重最小，值为：15，默认普通进程优先级 0 的权重大小为：1024。而数组[0,39]是个等比数列，按因子 1.25 递减，具体该数组对应值如图 3.6 所示：

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

图 3.6 优先级到权重转换数组

采用这样的方式能保证高优先级进程获得较多的运行时间，通过 set_load_weight 函数来为进程计算它的负荷权重。进程根据它的优先级将对应的负荷权重保存在 task_struct->se.load 中。

对于某个给定调度实体表示的进程，分配到的时间使用 sched_slice()函数获取得到。但是进程中并没有保存下该时间片的大小，而是在每次周期性调度时，比较已运行时间是否超过该值，如果是，则需要重新调度。

而 CFS 理想公平主要通过虚拟运行时间(vruntime)实现，它用来保存进程已经运行的虚拟时间总和，该时间与调度实体中的 sum_exec_runtime 不同，并不是直接记录进程的运行时间，而是会将运行时间根据进程的权重放大或者缩小一个比例，再进行记录。根据实际运行时间，进程的虚拟运行时间长度为：

实际运行时间*NICE_0_LOAD/进程权重

以默认进程的优先级的权重 NICE_0_LOAD 为标杆，越重要的进程会有越高的优先级，会得到更大的权重，因此根据公式加权化，使得累加的虚拟运行时间会小一些。字段 vruntime 对该值进行累加。

而红黑树的排序过程也主要是根据虚拟时间 vruntime 的大小，累积的 vruntime 越小，在红黑树中排序的位置就越靠左，因此会被更快地调度。

在进程运行时，vruntime 会稳定地增加，因此它在红黑树中总是向右移动的。因为越重要的进程 vruntime 增加的越慢，因此它们向右移动的速度也越慢，这样其被调度的机会要大于次要进程。

如果进程进入睡眠，则其 vruntime 保持不变。因为每个队列 min_vruntime 同

时会增加，那么睡眠进程醒来后，在红黑树中的位置会更靠左，因为其键值变得更小了。

3.3.3 CFS 函数操作

1. 入队函数 enqueue_task_fair()

该函数用来向可运行队列放置新进程，也就是往红黑树中添加进程。图 3.7 为该函数的主要代码流程图。

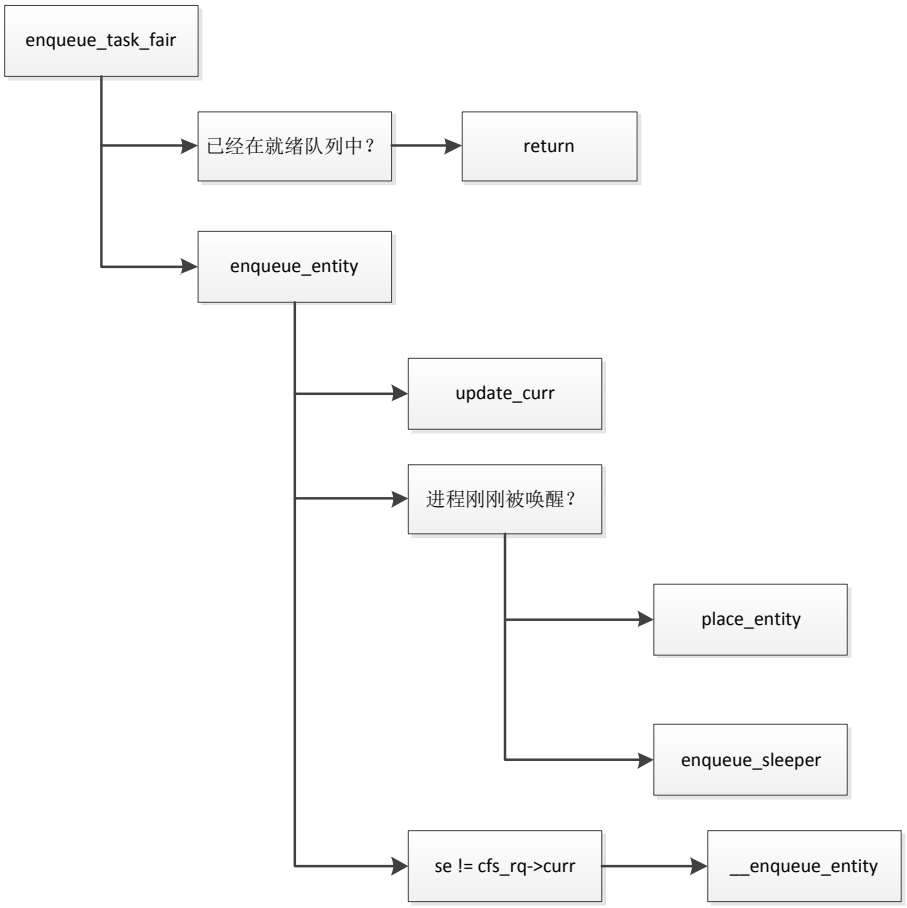


图 3.7 入队函数的代码调用关系图

该函数首先判断进程是否已在可运行队列上，若不在的话，才调用 enqueue_entity 函数进行插入操作，update_curr 函数用来更新统计量。place_entity 函数用来更新进程的虚拟运行时间 vruntime，使用__enqueue_entity 函数将进程加入到红黑树中。该函数根据进程的 vruntime 和队列中的 min_vruntime 值将进程置于红黑树中正确的位置。

2. 选择下一个进程函数 pick_next_task_fair()

该函数用于从红黑树中选择下一个需要运行的进程，因为 cfs_rq 中保存了指向红黑树最左结点的指针，所以只要队列不为空，那么该函数将调用

pick_next_entity 提取出进程。

选择好进程之后，还需要调用 set_next_entity()完成一些工作。该函数需要调用__dequeue_entity，该函数将设置 leftmost 指针指向下一个最左的结点。并且将当前要执行的进程从红黑树中删除。

3. 周期处理函数 task_tick_fair()

该函数的实际工作由 entity_tick()完成。首先调用 update_curr 更新统计量，随后调用 check_preempt_tick()函数判断是否应该被抢占。如果进程的运行时间比期望的时间间隔长，那么通过 resched_task()发出重调度请求，核心调度器会在下一个适当时机发起重调度。

3.4 实时调度算法

Linux 除了普通进程之外，还支持实时调度类 rt_sched_class。调度类结构使得实时调度类可以平滑地集成到内核中，而无需修改核心调度器，这就是调度类带来的好处。

实时调度类有两种调度算法先进先出 FIFO 和轮转 RR 如下所示：

1. SCHED_RR

该算法有时间片，其值在进程运行时会减少，如同普通进程。在时间片到期之后，会将该值重置为初始值，而进程则置为队列的末尾。当调度程序把 CPU 分配给进程的时候，如果存在几个优先级相同的 SCHED_RR 进程的话，那么它们总是依次执行。

2. SCHED_FIFO

该算法中没有时间片，当调度程序把 CPU 分配给进程的时候，除非出现更高优先级的进程，否则进程一直运行，直到它退出。相同优先级下，按先来先服务运行。

3.4.1 实时可运行队列

实时调度类中的关键数据结构为实时可运行队列 struct rt_rq，在可运行队列 rq 中嵌入了该数据结构。rt_rq 数据结构的定义在 kernel/sched/sched.h 中，不再详细列出，只对其中关键的成员进行分析。

而在 rt_rq 实时可运行队列中最关键的成员是 active，该成员表示实时进程的优先级队列，该成员的类型为 rt_prio_array。

因此 rt_rq 结构中关键的数据结构在于优先级队列 rt_prio_array 结构，该结构定义在 kernel/sched/sched.h 文件中，如图 3.8 所示：

```
struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
    struct list_head queue[MAX_RT_PRIO];
};
```

图 3.8 实时优先级队列结构定义

使用 DECLARE_BITMAP 生成一个位图 bitmap，该位图大小为实时优先级范围加 1。同时定义了一个对应的链表数组，active.bitmap 位图中的每个比特位对应于一个链表。位图中的每一位对应相应的优先级，如果存在该优先级大小的实时进程，则该将该比特位置 1（表示对应优先级链表有进程），并将进程链接到对应优先级链表中。具有相同优先级的所有实时进程会保存在一个链表中，表头为 active.queue[prio]，后面来的进程放到同一优先级队列的队尾。如果链表中没有进程，则该比特位不置位。使用位图可以提高搜索效率。

3.4.2 实时调度器的操作

1. 入队出队函数

由于实时调度器的队列设计，该调度类下的入队出队函数比较简单。入队函数 enqueue_task_rt()只需以 p->prio 为索引访问 queue 数组 queue[p->prio]，获得对应优先级链表，将进程加入链表末尾，如果链表之前没有进程，还要将位图中对应的比特位置 1；而出队函数 dequeue_task_rt()，同样通过进程优先级找到对应的优先级链表，将进程从链表中删除。删除完后，若链表中无进程，则将位图中对应的比特位置 0。

2. 选择下一个将要执行的进程函数 pick_next_task_rt()函数

该函数通过调用 sched_find_first_bit()函数，找到 active.bitmap 中第一个置位的比特位，这意味着高的实时优先级。取出所选链表的第一个进程，并将 se.exec_start 设置为可运行队列的当前实际时钟值。

3. 周期处理函数 task_tick_rt()

该函数首先会调用 update_curr_rt()函数用来更新当前时间统计。该函数对应于 CFS 调度中的 update_curr()。

然后判断如果当前进程是采用 SCHED_FIFO 的进程，由于这类进程没有时间片，所以直接返回。

如果当前进程是采用 SCHED_RR 策略的进程，则减少其时间片，在时间片未用完的情况下，进程可以继续执行。当时间片用完时，将值重置为 DEF_TIMESLICE，一般为 100 毫秒。如果该进程不是链表中唯一的进程，通过 requeue_task_rt()函数重新排队到末尾。并通过 set_tsk_need_resched()设置 TIF_NEED_RESCHED 标志，请求重新调度。

第四章 Linux 实时性改进技术分析

4.1 概述

本章从 4 个方面对现有的 Linux 实时改进技术进行了综述，由于现有实时性改进技术过于庞杂，所以，综述的角度的选取非常重要。首先，本章中所列举的改进技术均已在 Linux 上实现；此外，综述角度的选取采用如下方式，分为两部分说明：

1. Linux 内核的内部结构所进行的改进。
2. Linux 内核对外呈现的接口所进行的改进。

对于 1，分为三部分进行论述。首先论述的是组合各种调度算法的 Linux 内核实时调度框架改进方案；再论述适用于多种不同调度算法的通用改进方案（如：使用 mutex 实现 spinlock）。再论述针对问题模式的调度算法，包括硬实时、软实时算法。

4.2 调度框架改进方案

4.2.1 Slot-Based Task-Splitting 调度框架

该调度框架是将一个时间周期分为多个 slot，一个 CPU Core 上的一个 slot 只能运行一个进程，各个 CPU Core 上 slot 的开始和结束时间是一样的。并且，各个进程允许在不同的 CPU core 上迁移[11]。如图 4.1 所示。

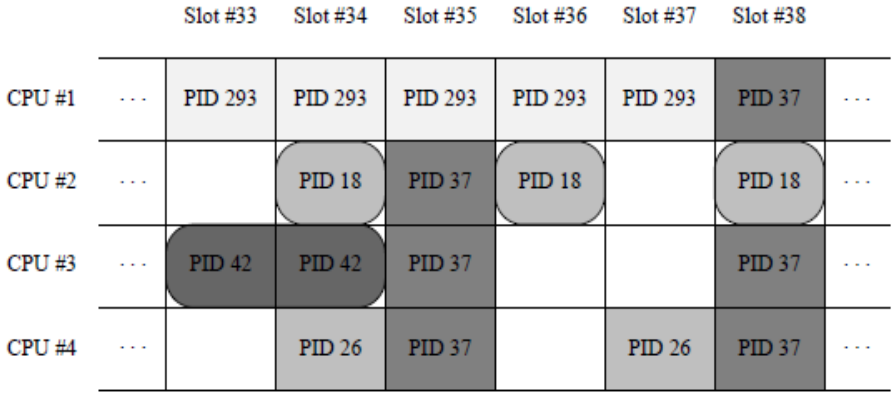


图 4.1 slot 调度序列

该调度框架的特点如下：

1. 允许多个调度算法同时存在。
2. 当系统中存在多个调度算法同时运行时，开发者能够较为容易的预则一个任务的完成时间（对于实时系统而言，就是判断该任务是否能够在时限前完成）。

与传统的 Partition 调度和 Global 调度相比，Task splitting[12]融合了 Partition 调度和 Global 调度的优点，避免了 Partition 调度和 Global 调度的缺点。即能够提高 CPU 的利用率，减少调度开销（对于多核体系架构，其中减少进程迁移代价的方法是利用多核体系架构中共享缓存和共享通信网络机制，使用缓存迁移），并且能够确保可预测性（大部分的任务是静态的划分到指定核上来减小缓存迁移，剩下的任务允许以一种预先决定的规则迁移到预先选择好的 CPU 核的子集上，来增加任务集调度的可预测性）。因为在 Partition 调度中，每个进程被固定分配到一个 core 上运行，每个 core 上使用一个调度器。而 Global 调度全局采用一个调度队列，允许进程在不同的 core 上迁移。Partition 调度虽然避免了进程迁移的开销，提高了可预测性。但会引发 bin-packing 这种问题，导致 CPU 的利用率低于全局调度，并且可能导致负载不均衡。而 Global 会引发全局调度队列的竞争问题，并且进程迁移会带来而外的开销，更重要的是：这种调度要预测任务的迁移时间非常困难。

4.2.2 Linux3.5.4 下的调度框架

如前面第三章中所述，在 Linux 调度框架可以分为两部分：第一部分为通用调度器，包括主调度函数和周期性调度函数；第二个为特定的调度类实例。其中调度类就是这两个部分之间的中间件，以接口的形式提供支持。这样 Linux 中的调度算法是调度类这种模块方式提供的，使得可以很方便的添加一个新的调度算法（只需要实现一个调度类实例），而不会对内核中已有的其它调度算法产生影响。并且进程也可以很方便的指定自身所属的调度类，对该进程进行调度时，所做的操作将由它所属的调度类中的方法来执行，调度的实际操作都将交由它来执行，比如选择下一个该运行的进程等等。

在进程中，通过进程描述符 task_struct 结构体中的成员：

const struct sched_class *sched_class;

来指明进程所属的调度类，来区别不同进程的调度方法。

因此，进程在创建之后需要设置进程的调度类。而在下面函数中，将对进程的调度类进行设置。

kernel/sched/core.c

```
static void
__setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
{
 p->policy = policy;
 p->rt_priority = prio;
 p->normal_prio = normal_prio(p);
```

```
    /* we are holding p->pi_lock already */
    p->prio = rt_mutex_getprio(p);
   /*在这里将通过判断 p 的优先级是否实时优先级来选择是实时调度类还是完全公平调度类*/
    if (rt_prio(p->prio))
        p->sched_class = &rt_sched_class; //实时进程设置调度类为实时调度类
    else
        p->sched_class = &fair_sched_class; //否则，进程采用完全公平调度类
    set_load_weight(p);
   }
```

如上例所示，对进程的优先级 prio 进行判断，如果进程的优先级在 0-99 之间，那么属于实时进程，则设置它的调度类为实时调度类。否则的话，就是普通进程，设置为完全公平调度类。

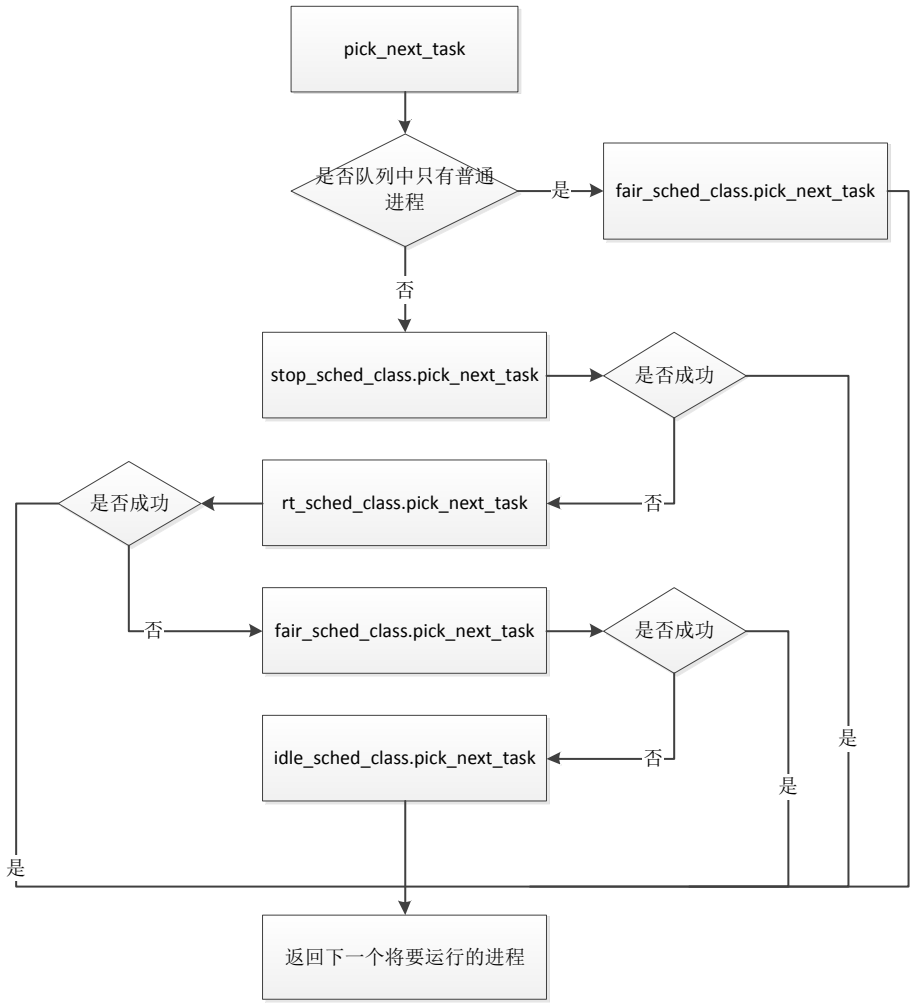


图 4.2 选择下一个进程函数的操作

首先由第三章图 3.1 可以知道，Linux 标准内核中生成了四个调度类实例，该实例按优先级顺序连接起来，可以看到实时调度类的优先级要高于普通进程。而实时进程总是优于普通进程先调度。

这是因为内核中调度发生时，需要选择下一个将要运行的进程，此时会调用 pick_next_task 函数，该函数的调用流程在图 4.2 中。

从图 4.2 中可以看出，该函数在选择下一个进程时，会按照调度类优先级顺序，从高到底来，调用该调度类中的 pick_next_task 函数，从而选出下一个进程，这就是为什么实时进程总是优于普通进程被执行。由于该函数中，可能需要遍历各个调度类，而系统中大多情况下可能只有普通进程，因此该函数会对普通进程做个优化，在遍历之前判断队列中是否只有普通进程，这样的话直接调用 fair_sched_class 中的方法，选出下一个进程。而这个优化往往也会起作用。

在实时调度类中因为有两种调度策略，所以在一些地方需要区分 RR 和 FIFO 的不同操作。

在获取时间片大小时，RR 的时间片是 RR_TIMESLICE，FIFO 的时间片为 0 。该函数为实时调度类中的成员，具体函数如下所示。

kernel/sched/rt.c

```
static unsigned int get_rr_interval_rt(struct rq *rq, struct task_struct *task)
{
 /*
  * Time slice is 0 for SCHED_FIFO tasks
  */
 if (task->policy == SCHED_RR)
        return RR_TIMESLICE;
 else
        return 0;
}
```

其次在更新时间片时，也会对 RR 和 FIFO 加以区分。因此 FIFO 调度策略是没有时间片概念的，所以该策略时，不对时间片进行操作直接返回。只有在 RR 时，才会递减或者重置它的时间片。task_tick_rt 是实时调度类实体中的成员，函数代码如下所示。

kernel/sched/rt.c

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
```

```
struct sched_rt_entity *rt_se = &p->rt;
update_curr_rt(rq);
watchdog(rq, p);
/*
 * RR tasks need a special form of timeslice management.
 * FIFO tasks have no timeslices.
 */
if (p->policy != SCHED_RR)     /*实时策略为 RR 才需要对它的时间片操作*/
        return;
if (--p->rt.time_slice)            /*减 1 后，时间片没用完将直接返回*/
        return;
p->rt.time_slice = RR_TIMESLICE;   /*时间片用完，重填该进程的时间片大小*/
……
}
```

内核采用这种调度类的机制能够很好的动态添加需要的调度类和调度策略，而不需要进行大范围的改动。

4.2.3 比较

表 4.1 调度框架对比

	Slot Based Task Splitting	Linux 3.5.4
是否可以组合不同的调度算法	是	是
预测实时任务对其它实时任务的影响的方法	基于 slot	基于优先级
预测实时任务对其它实时任务的影响的难易程度	易	难（很多情况下，很难知道高优先的任务究竟运行多长时间）

由表 4.1 可知，相比较 Linux 自身的调度框架，Slot Based Task Splitting 可以对实时任务的运行进行预测，能够很好的判断其他任务对它的影响，而 Linux 下很难做到这一点。因此，Slot Based Task Splitting 容易满足实时任务严格的时限要求。

4.3 通用调度算法延迟改进方案

RT patch 用于为 Linux 系统提供一些实时功能。RT patch 中所发展的功能有许

多已经加入主线内核。其中的一些功能包括：

1．使用 mutex 代替自旋锁

2．中断线程化

3．优先级继承

4．延后的工作

5．减少延迟的策略

4.3.1 使用 mutex 实现 spinlock

以往在 Linux 中，spinlock 是一个自旋锁，用来保护临界资源，由于进入该临界区的时间一般较短，所以 spinlock 在没有获得该临界区锁时自旋，而不是放弃 CPU[13]。为了避免一个进程使用 spinlock 加锁后，未解锁之前，出现另一个进程抢占该进程，重入该 spinlock 所保护的关键区将导致死锁情况的发生（这种情况称为失效抢占），所以，持有 spinlock 锁期间将会使抢占失效（如通过关中断等方法），因此会严重地影响着系统的实时性。比如：此时如果有一个根本不会加锁的高优先级的实时进程来到，那么，即便不会引发失效抢占，依然会导致 spinlock 无法使用。为此，在实时补丁 RREEMPT_RT 中，为了让 spinlock 也可以抢占，使用 mutex 互斥体来实现 spinlock。

但用 mutex 替换 spinlock 依然有可能产生失效抢占，为了避免失效抢占的发生，RT-Patch 采用等待队列来解决该问题：每个 spinlock 都会有一个等待队列，在进程请求 spinlock 锁时，如果当前锁被其他进程持有，那么该进程放弃 CPU，并添加到相应 spinlock 对应的等待队列中，等待队列是按进程优先级排序的，最高优先级的请求进程将会在队首，最低优先级的请求进程将在队尾。因此，当 spinlock 锁释放时，系统总是能很快的找到优先级最高的进程，并将它唤醒投入运行，减少延迟时间[14]。

4.3.2 中断线程化

PREEMPT_RT 补丁实现了上半部（Top-Half）中断线程化。以往的 Linux 处理，仅实现了下半部线程化。

所谓上半部和下半部是指：当一个中断发生时，中断处理程序分为两部分，上半部分是紧急需要立刻处理的，为了确保上半部分立刻执行，此时关闭中断。下半部是可以延时处理的，此时，将下半部变成一个线程处理。Linux 通过内核线程，tasklet 和 softirq 已经实现了下半部的线程化。

softirq 是一个服务程序，它的执行时机为：从一个 ISR 返回之后以及继续执行被中断的进程之前，如果有太多的 softirq 被排入队列，内核会唤醒一个具有高优先级的内核线程（ksoftirq）以便完成它们。

tasklet 跟 softirq 一样，它的执行时机也是在 ISR 之后以及继续执行被中断的进程之前。tasklet 跟 softirq 的差别在于，同一个 softirq 可以同时运行于两个不同的 cpu 之上，而 tasklet 不能。因此 softirq 必须使用上锁机制，以防止任何全局数据或其他资源的并行存取。而 tasklet 不必这么做。

内核线程的开销比 softirq 或者 tasklet 稍微多一些，不过它更为灵活。在 non-RT linux 内核之下，softirq 和 tasklet 无法被具有任何优先权的任何进程抢占。所以，尽管内核线程的开销比 softirq 和 tasklet 多一点，但是它可以使得内核调度程序更加灵活，而且可以让系统管理员获得更多控制权。

但在实际应用中，还存在着这样的情况，即：某些设备（注意：并不是所有设备）的上半部分也可以线程化，为了满足这种需求，RT Patch 提出了上半部线程化。为了区分出到底一个上半部是否需要线程化，RT patch 将该决定权交给驱动程序开发者决定。下面简要说明这一点的实现机制：

在使能 Linux RT-Preempt 后，默认情况下会强制透过 request_irq()申请的 IRQ 的上半部函数在线程中执行，在 Linux 标准内核中，request_irq 函数定义在 include/linux/interrupt.h 中，如图 4.3 所示：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

图 4.3 标准内核的中断申请函数

从图 4.3 可知，通过 request_irq()申请的 IRQ，在没有 Rt-Preepmt 的情况下，kernel 并不会为其创建 irq 线程，因为它在最终调用 request_threaded_irq()的时候传递的 thread_fn 为 NULL。

如果使能了 RT-Preempt Patch 的情况下，会强制 ARM 使用 threaded irq。但在使能了 RT-Preempt 的情况之下，此时仍然可以绕开上半部线程化的过程，只需要申请中断的时候设置 IRQF_NO_THREAD 标志。

4.3.3 优先级继承

在 linux 内核 2.6.18 版本中，优先级继承机制首次成为主线 linux 内核的一部分，主要用来解决优先级逆转问题。

优先级逆转可能会造成优先级较高的进程不能预测自己等待被占用的资源释放所持续的时间。eg：假设有三个进程：A，B 和 C，运行在一个仅含有一个 core 的 CPU 上，其中 A 的优先级最高而 C 的最低，C 先运行并获得一个锁然后被 A

抢占。A 试图去获取 C 所持有的锁，但是会被阻塞直到 C 释放它。这时，中间优先级的进程 B 到来，抢占了 C。这样，A 究竟需要等待多长时间，无法预测。

优先级继承的基本思想是：优先级低的进程继承任何与它共享同一资源的优先级高的任务的优先级。当高优先级的任务在资源上阻塞的时候，优先级立即更改。当资源被低优先级的任务释放时，这个改变结束。这样的话，以上 C 进程就会继承 A 进程的高优先级，从而不会被 B 进程抢占，一旦 C 进程释放该锁，它的优先级就会恢复，从而 A 进程就得到执行。

需要说明的是：解决优先级逆转的方案不止优先级继承这一种，RT patch 采用了优先权继承的方案。

4.3.4 延后操作

使用 mutex 实现 spinlock，这也意味着，当进程调用 spin_lock()后，进程有可能进入睡眠状态。如果进程在调用 spin_lock()前，系统处于关中断状态或者禁止抢占的状态，此时，就不能调用 spin_lock()。因为，如果在进程中使用 spin_lock()，造成进程进入睡眠状态，将使进程陷入死锁状态。然而，内核在实际中有这样的应用场景，即在关中断后，需要调用 spinlock()。RREEMPT_RT 对此的解决方案是：会延迟对 spin_lock()的申请直到抢占/中断又被重新开启为止，具体方法如下：

1. put_task_struct_delayed()将 put_task_struct()操作用队列管理起来，直到可以合法申请 task_struct 中的 spinlock_t alloc_lock 为止。
2. mmdrop_delayed()将 mmdrop()（该函数用来释放页表）操作用队列管理起来。

使用 TIF_NEED_RESCHED_DELAYED 标志可以进行重新调度，但是调度会延迟到进程准备好返回用户空间，或者下一个 preempt_check_resched_delayed()。两者的关键点是避免了无谓的抢占（将被唤醒的高优先级任务会等待当前任务释放一个锁）。如果不指定 TIF_NEED_RESCHED_DELAYED 标志，高优先级的任务将立即抢占低优先级任务，却很快又会阻塞在对低优先级占有的锁的申请上。解决的办法是：将后面紧跟着 spin_unlock() 的 wake_up() 替换为 wake_up_process_sync()。这样，如果将要被唤醒的进程会抢占当前进程，那么唤醒操作将通过指定 TIF_NEED_RESCHED_DELAYED 标记被延迟。

4.3.5 减少延迟的策略

PREEMPT_RT 对内核作出了一些修改来减少调度或者中断延迟。

1. 在 x86 系统上的 MMX/SSE 指令集，这些指令集在内核中的操作都是在内核禁止抢占的情形下使用的。大部分的 MMX/SSE 指令的操作都是非常迅速的，但是仍有一些 MMX/SSE 指令操作的过程比较慢，PREEMPT_RT 补丁对此作出了优化，尽量减少或者禁止使用这些操作比较慢的指令。

2. 在 slab 分配器中使用 per-cpu 变量，如：每个 CPU 都会维护一个空闲对象/页的链表，这样，在分配空闲空间时，就没有必要和其它 CPU 竞争，从而减少延迟。

4.4 调度算法本身的改进

目前，实时调度算法的数目繁多，分类的角度也极为庞杂，如：

1. 根据任务执行的周期，将调度算法分为：周期性（periodic）任务调度，非周期性（aperiodic）任务调度，零星（sporadic）任务调度。
2. 根据任务时限要求，将调度算法分为硬实时调度算法[15]和软实时调度算法。
3. 依据是否在多 CPU 上运行，分未单 CPU 调度算法和多 CPU 调度算法。
4. 依据是否具有容错性进行分类

因为所有的操作系统上的实时调度算法，本质上是对资源的分配算法，而分配所针对的目标，直接决定了资源分配算法的结构。为此，本文从实时调度算法满足任务所要求资源的程度，将资源分配算法分为：硬实时调度算法和软实时调度算法。硬实时调度算法和软实时调度算法的最大区别是：硬实时调度算法必须为任务分配足够的资源，以便其能够在时限前完成。而软实时调度算法为任务分配的资源数可以少于该任务为在时限前完成所需要的资源数。

由于即使将实时调度算法分为上述两大类，每类下的实时调度算法依然繁多，为此，本章仅列出一些典型的已经在 Linux 上实现实时调度算法。需要指出的是：这里的“典型”是指应用中反复遇到的问题模式所对应的一种解决方案。这是因为：

1. 所有的实时调度算法均和任务特性紧密相关（这也是实时调度算法如此庞杂的根本原因），即便是同一个问题，也有可能存在不同的解决方案。
2. 虽然这些算法已经在 Linux 上实现，但并不是说这些算法的实现已经合并到 Linux 内核主线中。
3. 在列举这些问题模式时，按照问题条件的严格程度，按照从严格到宽松的顺序一一列出。

4.4.1 硬实时调度算法

1) 问题模式 1：上层任务的调度时机确切

对于这种问题模式，上层应用的调度时机确切，即操作系统开发者明确地知道上层究竟要运行哪些应用，并且这些应用的调度时机也是明确的。对于这种问题，可以采用基于时间驱动（Time-Driven 也称 Clock-Driven）的进程调度算法。

总体来说，基于时间驱动的进程调度算法，是一种离线的静态调度方法[16]，

在设计之初确定。因此在进行系统设计的时候，就必须确定每个进程的调度时机，预先对每个进程的开始、结束以及切换时间做出正确的安排和设计。对于一些小型的传感器、自控系统、嵌入式系统等应用环境非常适合。基于时间驱动的调度算法优点就是对于任务的执行存在很好的可预测性。典型的实现有 RED-Linux[17]。

2) 问题模式 2：上层任务的运行时机未知，仅知其时限要求

在这种问题模式下，由于任务的运行时机未知，所以，为了保证突然出现的任务能够及时完成，为此，采用基于优先级的调度算法。这里仅举两例：EDF（Earliest Deadline First）[18]和 RMS（Rate-Monotonic Scheduling）[19]。

EDF 调度算法

EDF 调度算法动态地设置进程的优先级，根据它的资源需求，越接近时限的任务越先被调度[20]。该算法的好处是在资源分配和调度时有更大的灵活性。因此，在动态优先级调度算法中，EDF 算法是应用最多的、最常用的。

EDF 调度算法以任务 deadline 为基准，划分优先级，进程在被加入到可运行队列中时，以 deadline 排序，最近 deadline 的进程将得到执行。

由于内核采用调度类的方式可以方便的添加，因此可以将 EDF 实现为调度类实例，添加到内核中去，并对内核相关部分做修改，来提高标准 Linux 的实时性[21]。该算法支持多核平台，而且可以用于嵌入式体系结构（ARM）中，同时也适用于周期性或者非周期性任务。

EDF 调度算法在 Linux 内核中可作为一个调度类实现，在不影响其他调度类的正常行为的基础上，EDF 调度类还必须和已经存在的调度类进行正确的交互。事实上，每个调度类所完成的工作就是为系统提供一个可运行的任务。在 Linux 内核中，不同的调度类是通过不同的优先级顺序链接起来的。低优先级的调度类只有在高优先级调度类空闲的时候工作。其中一种实现方式是：EDF 调度类的优先级位于 sched_fair 和实施调度类 sched_rt 之间。这就意味着一个在 EDF 调度类中的任务可以抢占 sched_fair 调度类的任务，但是会被 sched_rt 调度类的任务所抢占。这样做的原因如下：

1. 基于兼容性的考虑：向后应用兼容以及 POSIX 标准都要求 SCHED_RR/SCHED_FIFO 的任务立即执行。
2. 在某些特定时间（系统掉电或者 CPU offline）的时候，系统需要通过简单的调度算法就能调度某一任务，而且这个任务不需要 deadline 的限制。
3. 因为 EDF 调度策略的性能比较高，所以即使是在 sched_rt 调度策略空闲的时候运行，也可以获取比较高的效率。

最后，需要指明的是：EDF 调度算法支持多核系统，每个 cpu 都有一个运行

队列，以红黑树的方式实现。而且在需要进程迁移的时候可以在多个 cpu 之间进行迁移。此外，EDF 调度算法可以用于周期性的，零散的或者非周期的任务。

静态优先级调度算法（RMS）

顾名思义，这种算法思想就是静态地给系统中的所有进程都分配一个优先级。根据任务的属性来分配静态优先级，可以使用的属性例如任务的执行周期或者其它的方式[22]。RMS 属于静态优先级调度算法中的典型实现，在实时系统中也比较常用。在 RMS 算法中，最短周期的任务具有最高优先级，次短周期的任务具有次高的优先级，以此类推。当有多个任务时，最短周期的任务被优先执行。因此这种静态调度算易于保障稳定性，尤其适用于工业应用中。RMS 算法会假定任务有以下特点：

1. 进程间没有资源共享，不允许进程交互；
2. 进程的 deadline 恰好等于它的周期；
3. 采用静态的优先级，具有最高优先级的进程一旦变为可运行状态，马上抢占其他进程。
4. 根据速率单调约定来划分静态优先级，最短周期的进程拥有最高优先级；
5. 进程的切换是瞬时的[23]。

3) 问题模式 3：上层任务的运行时机未知，运行时间难以预测

通常这种问题模式为实时调度带来的最大困扰是：难以预测一个任务是否满足时限要求。运行时间难以预测的原因主要有如下几点：

1. 交互式任务，任务的完成依赖于外界的输入，那么任务的运行时间就依赖于：a.外界输入完成的时间；b.处理该输入所执行的特定路径。实际中经常出现的情况是：a 无法预知，b 不能精确预知。从而导致任务运行的时间难以预测
2. 对于 Power-Aware 系统，为了达到降低能耗的目的，有时会降频运行任务。

所以，此时实时调度算法所要解决的关键问题就是如何提高系统运行的可预测性。目前解决该问题的较好方式是采用资源预留策略[24]。下面将给出一个这样的算法：

GRUB-PA（Greedy Reclamation of Unused Bandwidth—Power Aware）

目前提出的很多调度算法都采用了动态调整处理器的电压和频率的方法来提高系统的性能。这些调度算法的目标在于不仅仅是选择需要调度的任务，而且需要选择处理器的频率，从而在不影响系统实时任务调度性能的基础上最小化系统的能耗。

一种实现上述思想的调度算法为基于 power-aware 的 GRUB（Greedy Reclamation of Unused Bandwidth）算法，该算法能够应用于硬实时和软实时任务

共存的系统中。该算法可以回收那些系统分配给周期性任务而没有使用的带宽，或者零散任务（到达的频率比较低）的带宽，根据这些信息，我们可以降低系统的 CPU 频率，从而达到减少系统能耗的目的。

基于 Power-Aware 的 GRUB 的算法是对普通 Power-Aware 调度算法的改进。Power-Aware 调度算法是通过降低系统的电压（这种技术称为 DVS（dynamic voltage scheduling）），达到节省能量的目的。但是这种策略会导致一些问题：很多现存的处理器都能够动态的降低电压从而减少系统的能耗，但是，降低电压会增加系统的门延迟，从而导致处理器必须降低自己的频率以及处理器的速度，这样，所有的任务的执行时间就会增加。在实时系统中，如果频率的改变不恰当的话，就会使得很多实时任务不能在 deadline 之前完成。

为了解决该问题，Power-Aware 调度算法在采用 DVS 时，在每个时间点上，不仅选择需要调度的任务，而且指定所选处理器的运行频率。

Power-Aware 调度算法在混合了实时任务，非实时任务以及零星任务等的操作系统上，该算法的实现变得比较困难。由于在混合了实时任务，非实时任务以及零星任务等的操作系统上，常常采用资源预留框架解决问题。在资源预留框架中，服务器会为每个任务分配周期 P，以及预留资源 Q，即在周期 P 的时间段内，该任务可以执行的时间长度为 Q。对于这种资源预留框架有很多已经存在的实现算法，包括固定优先级以及动态优先级算法。如果任务执行的时间比预期的时间短，我们可以利用未使用的时间减少零星任务的响应时间。这种将为使用时间加以利用的算法称为回收技术，GRUB 就是一种采用该技术的资源预留框架算法。

其实回收技术跟 Power-Aware 算法的实现思想是类似的。这两种算法都可以分为两部分完成。第一部分就是计算未使用的时间，这两种算法在该部分上的实现是一致的。第二部分的实现有些差别。对于回收技术而言，它主要是利用这些未使用的时间执行一些零星的任务，而对于 Power-Aware 算法而言，它采取的措施是在未使用的时间上降低处理器的频率。即，Power-Aware 算法可以与 GRUB 算法相融合，解决 Power-Aware 调度算法在混合了实时任务，非实时任务以及零星任务等的操作系统上的实现问题。

GRUB-PA 算法是用于运行硬周期性任务以及软非周期性任务的系统。GRUB 算法的实现主要是基于资源预留的框架来实现的。GRUB-PA 算法在保证任务“运行时独立”的特性之外，相比较于其他的 Power-Aware 性能上有了明显的提高，相比较教于普通的 Linux 系统，采用 GRUB-PA 改进后的算法性能提高了百分之三十以上[25]。

4.4.2 软实时调度算法

1) 问题模式：多个任务均超时限

本节论述的软实时调度算法所针对的问题模式是：软实时调度中多个任务均超时限[26]。在此问题模式下，最重要的就是保证各个人任务能够公平使用资源。其中一个算法是：基于比例共享的调度算法。

由于硬实时调度算法并不在任何情况中都适用：比如在视屏会议这类的软实时应用中。因此，在这种应用环境下，应该采用基于比例共享的资源调度算法。

该算法是一种基于 CPU 使用比例的共享式的调度算法，其实现的核心是：调度一组任务时，是根据一定的比例或者说是权重，从而使任务的执行时间正比于它们的权重。

基于比例共享的调度算法最常使用的情形就是系统中有固定的数据流，比如上面提到的多媒体应用。比如系统正在接收网络数据包，共享比例的算法会将每个数据包看成独立的逻辑队列，共享比例算法会搜索所有的非空队列然后从每个非空队列中取出数据进行处理，取出的数据的量因各自权重的不同而不同。

该算法实现的方式通常分为两种：第一种方法是对已就绪进程出现在调度队列队首的频率进行调节，调度并执行队首的进程；第二种方法就是将可运行队列中的各个进程依次调度运行，但每个进程的运行时间片时间片大小是由分配的权重决定[27]。

由于比例它不存在任何优先级的概念，任务共享 CPU 资源是根据任务申请的比例，如果系统当前负载过重的话，所有任务的执行就会按比例地变慢。所以一般会采用一种动态调节进程权重的方法，来保证系统中重要进程能够及早处理完毕。

4.4.3 比较

表 4.2 各实时调度算法的比较

	硬实时/软实时	调度时机	优先级	时限	运行时间	Power-Aware
Time-Driven	硬实时	预知	无	预知	预知	无
EDF	硬实时	动态确定	动态确定	预知	预知	无
RMS	硬实时	动态确定	预先静态确定	预知	预知	无
GRUB-PA	硬实时	动态确定	动态或静态确定	未知	未知	有
比例共享	软实时	动态确定	无	预知或未知	预知或未知	无

如表 4.2 所示，对各实时调度算法进行比较，按各调度算法在调度实际、优先级、时限、运行时间等方面的效果进行综合对比。这样将各调度算法在各方面的特性显示出来，对于在特定问题模式下选择合适的调度算法提供便利。

4.5 内核接口 API 的改进方案

如果仅有上述的章节中的改进机制，而没有利用这些机制实现的为开发者提供的 API，那么，开发者开发实际的实时应用时就不够方便。目前，主流的 Linux 内核接口 API 的改进方案是：RTAI（Real-Time Application Interface）[28]和 Xenomai[29]。下面将分别对其论述。

4.5.1 RTAI

RTAI 是对 Linux 的扩展，它为 Linux 提供了大量的实时接口。其设计核心思想是：把 Linux 内核当做一个普通任务，该任务的优先级较低，在调度的时候，如果当前存在实时进程，那么运行实时进程，否则的话，执行 Linux 内核。RTAI 的实现主要经历了两个阶段：i. 使用 RTHAL（Real-Time Hardware Abstaraction Layer）实现 RTAI。ii. 使用 ADEOS（Adaptive Domain Environment for Operating Systems）实现 RTAI。RTHAL 和 ADEOS 这两种技术本质上是相同的，只是因为版权问题，才使得 RTAI 后来的实现转向了 ADEOS。

1. 基于 RTHAL 的实现方式

RTAI 被设计为 3 个基本组成部分：中断分发器，实时调度器，实时通信器。实时调度器和实时通讯器均采用 Linux 的模块机制实现。即，RTAI 的设计结构更像一个实时地微内核结构，将其完成的功能分解成为一些基本的服务，如上面提到的实时调度器，也可加载 FIFO，共享内存等实现实时通信器。

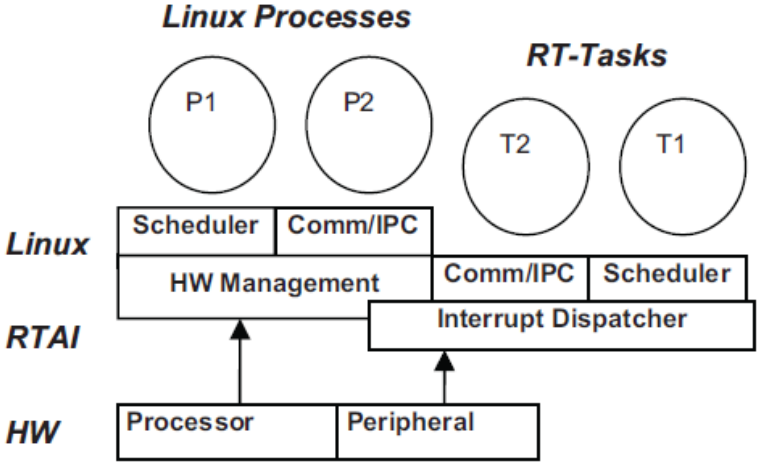


图 4.4 基于 RTHAL 的 RTAI 体系架构图

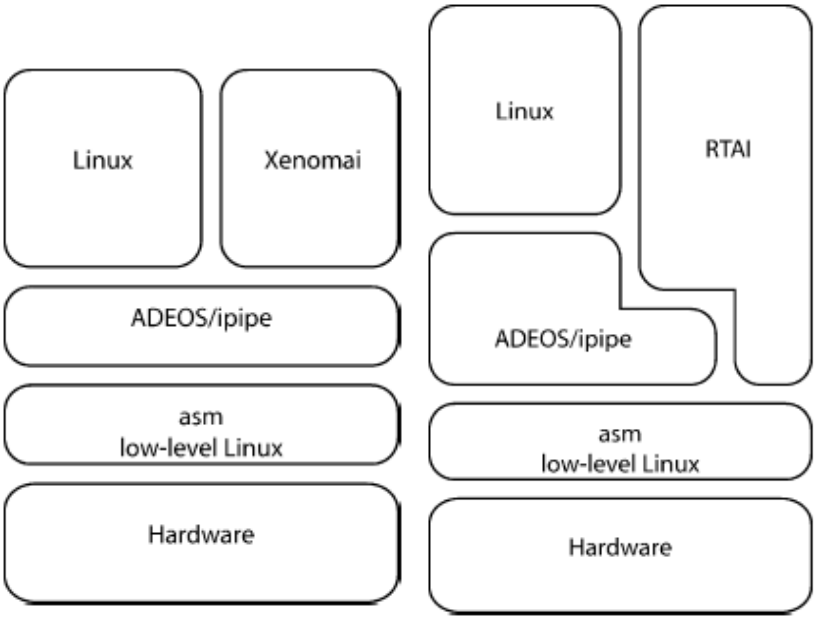
如图 4.4 所示，其中 RTHAL 的主要作用是：捕获硬件传来的中断，交给 RTAI 中的对象处理。定义 RTHAL 目的是大大减少了 Linux 内核源码中需要修改部分，从而提高 RTAI 和 Linux 内核代码的可维护性。将要修改的部分组合成一个接口界面，通过这个界面 Linux 与 RTAI 进行通信[30]。

2. 基于 ADEOS 的实现方式

ADEOS 是一个资源虚拟化管理器，是作为一个 Linux 补丁实现的，它允许一个实时内核和 Linux 同时运行。这种实现方式和 RTHAL 的实现共同之处在于：帮助 RTAI 中的对象截获中断。不同之处主要在于 ADEOS 的实现要比 RTHAL 复杂，因而基于 ADEOS 的调度延迟要比基于 RTHAL 的调度延迟稍微高一点。

4.5.2 Xenomai

Xenomai 的实现也是基于 ADEOS，即 Xenomai 和 RTAI 的实现本质上是相同的，其实现架构的不同非常小，如图 4.6 所示：



(a) Xenomai 系统架构　　(b)RTAI 系统架构

图 4.5 Xenomai 和 RTAI 的系统架构图

从图 4.5 中对比可看出，Xenomai 不能绕过 ADEOS 与硬件交互而 RTAI 可以，因此 RTAI 可以对激活实时进程的中断进行直接处理，相比 Xenomai 它的中断延迟更短。

而 Xenomai 的特点在于，提供了丰富的 API，它可以为开发者提供其它 RTOS 上的 API 调用，如：Vxworks，VRTX 等 RTOS 上的 API 调用。甚至 Xenomai 还可以提供 RTAI 的 API 调用。如图 4.6 所示。

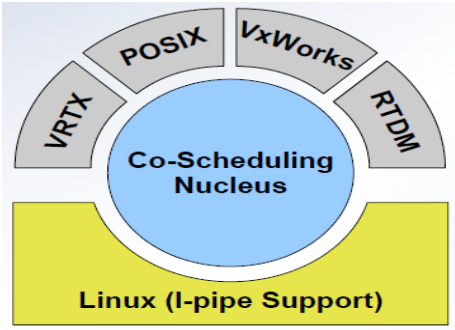


图 4.6 Xenomai 的 API 支持

从图 4.6 中显示了 Xenomai 支持的部分 API，Xenomai 提供多套与主流商业 RTOS 兼容的 API 以及对硬件的广泛支持，使得在其之上构建的应用系统能保持较高实时性。

第五章 Linux 实时调度算法改进

5.1 针对的应用模式

由于不存在一个适用于任何场景下的通用的实时调度算法，而目前，有关融合不同实时调度算法的框架已经基本成熟，为此，本项研究针对一种特定的上层应用模式提出一种实时性改进方案。

该应用模式描述如下：

应用进程可以分为两个集合：$S_1=\{P_1,P_2,..P_m\}$，$S_2=\{P_{m+1},P_{m+2},\ldots P_n\}$。$S_1 \cap S_2=\Phi$

1. S_1 中的进程存在着依赖关系，S_2 中的进程存在着依赖关系，但 S_1 中的进程与 S_2 中的进程相互独立。
2. 对于 S_1 集合中的每个进程，CPU 每个应用进程所分配的时间片的大小是不一样的，即每个进程的权重是不一样的。
3. S_2 中进程出现的时机不能确定。但 S_1 与 S_2 中的进程数目是已知的。
4. S_1 中进程的优先级相同。

符合上述应用模式的常见的典型应用场景是：驾车使用手机 GPS 导航的同时，接入电话；硬件平台是双核 CPU。

其中，GPS 导航至少完成如下功能：

1. 从 GPS 定位设备的获取当前位置点
2. 与 mysql 数据交互，获得当前所在区域的地图
3. 与绘图子系统交互，将所在区域地图画在 GPS 导航仪上。

也就是说：GPS 导航应用至少由三个进程构成:

1. 导航业务逻辑进程
2. Mysql 数据库进程
3. GPS 绘图进程。

这三个进程和打电话进程不相关。但这三个进程相互之间有依赖关系，并且这三个进程的优先级是一样的，但显然这三个进程进行一次定位服务计算时所要耗费的时间是不相同的。

5.2 实时调度算法设计

实时调度算法设计如下：

1. 将相互依赖的进程放在一个 CORE 上一起运行。将无相互依赖的进程放在不同的 CORE 上运行
2. 为了充分利用资源，当无相互依赖的进程不存在时，设此时相互依赖的进

程在 CORE X 上运行，那么此时就将相互依赖的某个或几个进程放在另一个 CORE Y 上运行；一旦无相互依赖的进程出现时，那么无相互依赖的进程会占领 CORE Y，并将 CORE Y 上的其它进程放到 CORE X 上运行。

3. 挂起那些与最终任务目标无关的进程。
4. 采用 WRR（weighted round robin）

采用上述设计方案的原因是：

对于 1 和 2：将相互依赖的进程放在一个 core 上运行，可以减少因放在多个 core 上运行导致某个 core 必须等待其它 core 上的进程所造成的 core 上时间资源的浪费，并且减少对其它 core 上运行的实时任务的影响。

对于 3，这是因为：调度存在相互依赖关系的 S_1 中的进程时，为了提高其实时性，那么最好的策略是：仅当需要调度其它进程为本进程提供服务时，再调度其它进程。但这样做，也就意味着操作系统必须知道进程的依赖关系图，虽然可以让应用提供该依赖关系图，但这一点在开放系统中是非常难以做到的（eg: 导航进程可能可以进行配置，可以和多种数据库进行操作，而其中的某些数据库是闭源的，所以无法获取该数据库的行为，从而无法得到依赖关系图。即便数据库是开源的，想从源码中获取进程依赖关系图也绝非易事）。另一种可能的实现方式就是让操作系统自行发现应用的依赖关系。但这样也使不现实的，比如，一个进程在获取用户空间的 spinlock 锁时陷入忙等待，操作系统此时是不能知道哪个进程获得了这把锁。因为操作系统不知道上层应用的业务逻辑。唯一可能的现实方法就是人工规定需要调度进程的最小集合，利用操作系统已有的机制部分发现进程所存在的依赖管理。Eg: 当一个导航进程需要通过 socket 读取 mysql 进程的数据时，因数据还未到，从而陷入睡眠状态，此时调度其它进程。注意，此时不能保证一定调度 mysql 进程，因为操作系统不知道导航进程是因为 mysql 进程睡眠，为了增大调度 mysql 进程的概率，采用的方式就是削减所要调度的进程的最小集。

对于 4：由于每个进程的权重不一样，所以，对于这些进程，需要采用 WRR，而不是单纯的 round robin，以减少进程切换的开销。

5.3 算法具体实现

5.3.1 基于关联度的算法实现

该算法的实现主要思路是在 sys_execv 函数处预先埋入“钩子”（hook），一旦发现所关心的那些进程被载入，立刻将其放入指定的 CORE 上运行。由用户指定哪个进程放在哪个核上运行，并且能够钩住线程：即应用进程运行时，可能会启动多个线程，该线程可能会调用 sched_setaffinity，绕过嵌入的钩子，设置自己的

cpu affinity，而该功能能够让进程中的线程无法绕过钩子，只能按照用户定义的亲和性运行。可将具有关联度的进程设置在同一个核上运行，有效利用等待时间。

首先通过创建一个内核模块 hooker，在虚拟文件系统/proc 下生成名为 hooker 的文件，作为内核与用户进行交互的接口。用户通过向/proc/hooker 文件中写入内容，让指定的进程运行在指定的核上。

然后在 x86 环境下，对内核源码对其中 arch/x86/kernel/process.c 中的 sys_execve 函数进行修改，首先会读取我们编写的内核模块/proc/hooker 文件中的内容，如果要产生的新进程名存在/proc/hooker 文件中，那么按照该文件指定的要求设置它的亲和性。如果进程名不在其中，则按照标准的内核进行操作。这样用户就可以通过在用户空间下，设定具有关联度的进程。具体函数的修改如下：

```
        char *filename;
+       struct cpumask cmask;
+       int count=0;


        filename = getname(name);
+       read_lock(&hookerlock);
+       if(proc_mnum==2)
+       {
+               while(count<proc_mnum)
+               {
+                       if(strcmp(filename,proc_m[count].proname)==0)
+                       {
+                               printk(KERN_ERR "ker_db: %s Intercept it!\n",filename);
+                               cpumask_clear(&cmask);
+                               cpumask_set_cpu(proc_m[count].cpumask,&cmask);
+                               proc_m[count].pid=current->tgid;
+                               sched_setaffinity(current->tgid,&cmask);
+
+                       }
+                       count++;
+               }
+       }
+       read_unlock(&hookerlock);
```

5.3.2 Weighted round robin 算法实现

添加了一种全新的调度策略 SCHED_WRR。该策略类似 RR，但是由于使用 RR 调度策略的话，任务分配得到的时间片大小会是固定的。所有任务执行的时间片大小都是一样的，在本机中配置为 100ms。而 WRR 策略的意思是根据进程的权重来设置进程的时间片大小。它定义一个最小时间单元，每个进程的所分配的时间片均是该时间单元的整数倍。不同的进程所分配的时间单元数是不一样的（注：由程序员通过 sched_setscheduler 系统调用指定）。

```
@@ -1216,6 +1219,7 @@
   * Timeslices get refilled after they expire.
   */
 #define RR_TIMESLICE          (100 * HZ / 1000)
+#define WRR_TIMEUNIT          (10 * HZ / 1000)
```

从该 patch 中可以看出，定义了一个最小时间单元 WRR_TIMEUNIT，该时间片大小是原有 RR 策略下时间片大小的 1/10。

```
diff -uNr ws/linux-3.5.4/include/linux/sched.h uplinux/include/linux/sched.h
--- ws/linux-3.5.4/include/linux/sched.h  2012-09-15 06:28:08.000000000 +0800
+++ uplinux/include/linux/sched.h 2014-02-27 09:42:10.830425165 +0800
@@ -39,6 +39,7 @@
 #define SCHED_BATCH           3
 /* SCHED_ISO: reserved but not implemented yet */
 #define SCHED_IDLE            5
+#define SCHED_WRR             6
 /* Can be ORed in to make sure the process is reverted back to SCHED_NORMAL on fork */
 #define SCHED_RESET_ON_FORK     0x40000000

@@ -46,6 +47,7 @@

 struct sched_param {
        int sched_priority;
+       int wrrunits_num;
 };
```

WRR 策略的意思是根据进程的权重来设置进程的时间片大小。因此我们需要给内核传递进程的权重参数。修改内核中 sched_param 的结构，在其中添加了一个

表示权重的成员 wrrunits_num，称为 WRR 的权重。用户可以在系统调用时，通过它设置单元时间片的多少。并在内核源码中宏定义调度策略 SCHED_WRR。

```
diff -uNr ws/linux-3.5.4/kernel/sched/core.c uplinux/kernel/sched/core.c
--- ws/linux-3.5.4/kernel/sched/core.c  2012-09-15 06:28:08.000000000 +0800
+++ uplinux/kernel/sched/core.c  2014-02-27  09:40:25.681432354  +0800  @@  -4385,6 +4386,16 @@
        oldprio = p->prio;
        prev_class = p->sched_class;
        __setscheduler(rq, p, policy, param->sched_priority);
+
+       if(p->policy == SCHED_WRR)
+       {
+               p->rt.wrr_time_slice=(param->wrrunits_num)*WRR_TIMEUNIT;
+               printk(KERN_ERR "ker_db: pid=%d, rt.wrr_time_slice=%d\n",p->pid,p->rt.wrr_time_slice);
+       }
+       else
+       {
+               p->rt.wrr_time_slice=0;
+       }
```

判断进程如果使用的策略为 SCHED_WRR，对采用该策略的进程的时间片大小赋值时：使用单位时间片乘以权重的大小，这就是进程的时间片大小。所以在一些比较紧要的进程中可以它的权重设置大一些，而在一些不太紧要的进程中，可以设小一些。

```
diff -uNr ws/linux-3.5.4/kernel/sched/rt.c uplinux/kernel/sched/rt.c
--- ws/linux-3.5.4/kernel/sched/rt.c  2012-09-15 06:28:08.000000000 +0800
+++ uplinux/kernel/sched/rt.c  2014-02-27  09:40:25.744432347  +0800  @@  -1996,13 +1996,21 @@
         * FIFO tasks have no timeslices.
         */
        if (p->policy != SCHED_RR)
-               return;
-
```

```
+       {
+               if (p->policy != SCHED_WRR)
+                       return;
+       }
        if (--p->rt.time_slice)
                return;

-       p->rt.time_slice = RR_TIMESLICE;
-
+       if (p->policy == SCHED_RR)
+       {
+               p->rt.time_slice = RR_TIMESLICE;
+       }
+       else   //p->policy==SCHED_WRR
+       {
+               p->rt.time_slice =p->rt.wrr_time_slice;
+       }
```

在进程时间片用完，重新分配时间片的时候，需要使用判断进程是否为 WRR 调度策略，如果是，将之前计算好的时间片赋给进程。

5.4 测试验证

5.4.1 测试环境

该测试环境是在已修改的 linux-3.5.4 内核—linux-3.5.4-WRR 上进行的，实验机为 x86 平台，用的 linux 发行版为 fedora17，CPU 为 AMD A6-3650 APU with Raden(tm) HD Graphics，内存为 4GB DDR3 1333 MHz。

Systemtap 是一种脚本语言，它是使用 kprobes 生成的工具，可以对内核中的数据进行提取、过滤和总结，以便更好地分析性能或者是问题。在这里我们使用 systemtap 脚本来动态的监视 CPU 核上相关的进程切换，统计它们的次数，并且查看它们每次运行的时间。

测试说明：

在 5.4.2 功能性测试第一节，基于进程依赖的调度算法中，采用的进程为 t1、t2 为/root/hooker/下的 t1、t2 进程。

而 5.4.2 功能性测试第二节及之后的 5.4.3 性能测试中，所使用的 t1,、t2 进程

为/root/ws/下的 t1、t2 进程。

5.4.2 功能性测试

1) 基于进程依赖的调度算法

在已经打完补丁的 Linux-3.5.4 内核上，先加载内核模块 hooker：insmod hooker.ko ，然后向产生的文件/proc/hooker 中写入以下内容，/proc/hooker 是用户用来指定 hooker 钩住哪些进程的地方：

```
0    /root/hooker/t1
1    /root/hooker/t2
```

其中，第一列用来指定想要运行在哪个核上，第二列用来指定所要绑定的进程名，该文件可能有多行设定。测试的主要目的是根据写入/proc/hooker 中的设置，是否将指定的进程放指定的核上运行。

而测试使用的进程 t1、t2 都是相同源代码生成的 demo 进程，其中都能生成多个线程，并对线程进行亲和性设置。进程的源码如下：

```
cpu_set_t cpuset;
void* th(void *p)
{
        pthread_t t;
        t=pthread_self();
        pthread_setaffinity_np(t,sizeof(cpu_set_t),&cpuset);
        while(1)
        {
        sleep(1);
        }
}
int main()
{
        int s,i,j;
        pthread_t myt;
        CPU_ZERO(&cpuset);
        CPU_SET(2,&cpuset);
        for(i=0;i<10;i++)
                pthread_create(&myt,0,th,0);
        while(1)
```

```
        {
            sleep(1);
        }
    }
```

运行这两个进程，并使用 ps 命令查看进程 t1、t2 的是否如用户所设置的，绑定在指定的核上。

```
[root@localhost hooker]# ps -eo pid,psr,cmd | grep hooker
 2861   0 /root/hooker/t1
 2862   1 /root/hooker/t2
```

图 5.1 显示进程运行所在核

如图 5.1 所示，t1、t2 已如/proc/hooker 中所设置的一样，分别绑定在核 0 和核 1 上。

再通过 ps 查看 t1、t2 所生成的线程是否也绕过了线程中的亲和性设置，在 t1、t2 所在核上运行，如图 5.2 所示。

```
[root@localhost hooker]# ps -eL -o pid,psr,cmd | grep hooker
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2861   0 /root/hooker/t1
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
 2862   1 /root/hooker/t2
```

图 5.2 显示各线程所在核

使用参数 L 来显示线程，可以看到 t1 所产生的所有线程都在核 0 上运行，而 t2 所产生的所有线程都在核 1 上运行，绕过了线程中的亲和性设置，这样通过对

/proc/hooker 进行设置，有关联度的进程将只会运行在指定的核上。这样相互依赖的进程将在同一个核上轮转运行，有效利用了进程等待数据的时间。

2) Weighted round robin 时间片

通过对内核的修改，添加了一种全新的调度策略 WRR。在本机中 RR 的时间片大小为 100ms，我们设置 WRR 的最小时间片为 10ms。

测试进程 t1、t2 为/root/ws/下的 t1、t2 进程，不同于/root/hooker/下的 t1、t2 进程，这里的进程属于实时进程，并设置它们采用的实时调度策略 RR 或者 WRR。

进程源码中对实时调度策略设置片段如下：

```
struct sched_param param;
int sched_method=SCHED_WRR; //设置为 WRR 调度策略或者 SCHED_RR
param.sched_priority=80; //设置进程优先级
param.wrrunits_num=50; //设置时间片权重
sched_setscheduler(getpid(),sched_method,&param);
```

在测试之前，会通过内核模块/proc/hooker 进行设置，将/root/ws/t1、/root/ws/t2 都绑定在 CPU 核 0 上。

使用 systemtap 脚本 cswmon_spec.stp 可以来得到进程每次执行的时间，进而算出它的单位时间片大小，代码如下所示。

cswmon_spec.stp

```
global time_offset
global switch_num
probe begin { time_offset =gettimeofday_us();switch_num =0 }
probe scheduler.ctxswitch{
    if(next_task_name =="t2" || prev_task_name == "t1" || next_task_name == "t1" || prev_task_name == "t2") {
        t=gettimeofday_us()
        printf("time_off (%8d)%20s(%6d)(prio=%4d)(state=%6d)---> %20s(%6d)(prio=%4d)(state=%d)\n",t-time_offset,prev_task_name,prev_pid,prev_priority,(prevtsk_state),next_task_name,next_pid,next_priority,(nexttsk_state))
    }
}
probe scheduler.process_exit
{if(execname() == "t2")
printf("task:%s PID (%d) PRI(%d) EXIT \n",execname(),pid,priority);
```

```
printf("Switch_num(%d) \n",switch_num);
}
probe timer.s($1)
{
        printf("----------------------------------------\n")
        exit();
}
```

关于脚本中涉及的打印的一些参数的含义如下：

time_off： 标志

t-time_offset：当前时间

prev_task_name:将要唤出的进程名称

prev_pid：将要唤出的进程号

prev_priotiry: 将要唤出的进程优先级

prev_state：将要唤出的进程状态

next_task_name：将要唤入的进程名称

next_pid：将要唤入的进程 id

next_priority：将要唤入的进程优先级

next_state:将要唤入的进程状态

而脚本在运行时还需要输入一个参数，表示脚本要运行多少秒。

我们使用脚本 ms.sh 来执行 t1、t2 进程，用来测试采用 RR 或者 WRR 策略时的时间片大小。

ms.sh 脚本的内容如下，主要用来近似同时启动两个进程。

```
#!/bin/sh
chrt –p 80 $$
/root/ws/t2 &
/root/ws/t1 &
```

该脚本中使用了 chrt 命令，该命令可以更改进程的调度策略和优先级，并且可以通过指定目的进程的 PID 来更改正在运行的进程的调度策略。使用该命令必须以 root 权限进行。

而环境变量$$表示的是当前 bash 的 pid。该命令的作用在于更改执行这个 shell 脚本的 bash 的优先级，这样做的好处在于可以避免这种情况：如果 bash 的优先级低于 t2，并且 bash 也是被放到和 t2 绑定的核 0 上执行，那么 shell 中，在 t2 得到执行后，t2 将会抢占 bash 得到执行，并在它执行完后，bash 才重新得到执行，而

这时 t1 才得到执行，这样就无法形成 t1、t2 采用 RR 或者 WRR 策略交替执行的效果了。

先执行 cswmon_spec.stp 脚本，随后在执行 ms.sh 脚本。

stap ./cswmon_spec.stp 90

/root/ws/ms.sh

确保 cswmon_spec.stp 的参数时间，要比 ms.sh 脚本的运行时间要长。我们确定 ms.sh 的运行时间不会超过 90 秒，并且将脚本的输出重定向到一个文件中，已便观测。

a) RR 策略的时间片

首先 t1、t2 均采用调度策略 RR，并且设置的实时优先级也都为 70，根据前面说明，先后运行 cswmon_spec.stp 和 ms.sh 脚本，进行测试，并将输出结果重定向到 rr.txt 中，以便之后进行分析。得到 RR 的执行结果并进行分析：

```
.....
time_off ( 8946304)                    t2(  4729)(prio=  29)(state=      0)--->
t1(  4730)(prio=  29)(state=0)
time_off ( 9046304)                    t1(  4730)(prio=  29)(state=      0)--->
t2(  4729)(prio=  29)(state=0)
//通过分析上面两行得出 t2 的调度时间为：100ms
time_off ( 9146304)                    t2(  4729)(prio=  29)(state=      0)--->
t1(  4730)(prio=  29)(state=0)
time_off ( 9246304)                    t1(  4730)(prio=  29)(state=      0)--->
t2(  4729)(prio=  29)(state=0)
//通过分析上面两行得出 t1 的调度时间为：100ms
time_off ( 9346306)                    t2(  4729)(prio=  29)(state=      0)--->
t1(  4730)(prio=  29)(state=0)
.....
```

从观测得到的结果可知，采用 RR 策略的 t1、t2 的时间片大小一样，都为 100ms，如之前内核中配置的一样。

b) WRR 策略的时间片

再将 t1、t2 均采用调度策略 WRR，并且设置的实时优先级也都为 70，其中 t1 的权重设置为 50，t2 的权重设置为 10，然后进行测试，并将结果重定向到 wrr.txt 中，以便之后进行观测。

得到 WRR 的执行结果，并对结果进行分析：

```
.....
time_off ( 4458784)                    t1(  4073)(prio=  29)(state=      0)--->
t2(  4072)(prio=  29)(state=0)
time_off ( 4558785)                    t2(  4072)(prio=  29)(state=      0)--->
t1(  4073)(prio=  29)(state=0)
//通过分析上面两行得出 t2 的调度时间片为：4558785-4458784；约为 100ms
time_off ( 5058785)                    t1(  4073)(prio=  29)(state=      0)--->
t2(  4072)(prio=  29)(state=0)
time_off ( 5158784)                    t2(  4072)(prio=  29)(state=      0)--->
t1(  4073)(prio=  29)(state=0)
time_off ( 5658784)                    t1(  4073)(prio=  29)(state=      0)--->
t2(  4072)(prio=  29)(state=0)
//通过上面两行得出 t1 的调度时间片为：5658784-5185784；约为 500ms
.....
```

通过对 t1、t2 运行时间片的计算发现，它们的时间片大小确实如设置的权重一般，t1 时间片的大小是 t2 的 5 倍。

说明 WRR 策略的确正确设置了进程的时间片大小。

5.4.3 性能测试

1) 进程的周转时间

该测试可以依据 5.4.2 第二节 Weighted round robin 时间片实验中得到的 cswmon_spec.stp 输出结果。

a) RR 策略下的周转时间

t1、t2 均采用调度策略 RR，并且实时优先级也都为 70，根据以上 5.4.3 中 RR 策略时间片的测试输出结果文件 rr.txt，分析：

```
//t2 进程开始时间
time_off ( 8148181)                    t2(  4729)(prio=  29)(state=      0)--->
ms(  4730)(prio=  19)(state=0)
//t1 进程开始时间
time_off ( 8247305)                    t1(  4730)(prio=  29)(state=      0)--->
t2(  4729)(prio=  29)(state=0)
……
//t2 进程结束时间
time_off (42932701)                     t2(  4729)(prio=  29)(state=     64)--->
```

```
t1(    4730)(prio=    29)(state=0)
      //t1 进程结束时间
      time_off (43125598)                    t1(    4730)(prio=    29)(state=       64)--->
kworker/0:0(    2750)(prio= 120)(state=0)
```

从以上结果分析中可以看出：

t1 进程的执行周期约为： 34s

t2 进程的执行周期约为： 35s

从而可以看出：在采用 RR 调度的两个进程的周转时间基本相等，相同优先级的进程在同一核上运行时，周转时间相等。

b) WRR 策略下的周转时间

t1、t2 均采用 WRR 策略，优先级均为 70，且其中 t1 的权重设置为 50，t2 的权重设置为 10。根据以上 5.4.3 中 WRR 策略的时间片实验得到的 cswmon_spec.stp 输出结果 wrr.txt 进行分析。

```
      //t1  进程开始时间
       time_off ( 2059788)                    t1(    4073)(prio=    29)(state=         0)--->
t2(    4072)(prio=    29)(state=0)
      //t1 进程结束时间
      time_off (23260904)                    t1(    4073)(prio=    29)(state=       64)--->
t2(    4072)(prio=    29)(state=0)
      //t2 进程开始时间
      time_off ( 1961523)                    t2(    4072)(prio=    29)(state=         0)--->
ms(    4073)(prio=    19)(state=0)
      //t2 进程结束时间
      time_off (37347534)                    t2(    4072)(prio=    29)(state=       64)--->
kworker/0:2(    2384)(prio= 120)(state=0)
```

从分析结果中可以看出：

t1 进程的执行周期约为：21s

t2 进程的执行周期约为：36s

发现进程 t1 的执行周期明显的变短，说明：通过采用 SCHED_WRR 调度策略，通过给定权重参数，可以对进程重新划分时间片大小，可以使得权重大的进程每次得到的时间片更大，从而执行周期明显缩短。这样，当进程在同一个核上时通过权重可以使得紧要的进程能更快的执行完成。

2) 切换次数

t1、t2 进程的切换次数，在 cswmon_spec.stp 脚本中，有一个全局变量 switch_num，在 t1、t2 的换入、换出的同时，对它们的切换次数进行统计。

因此在分别使用 RR 和 WRR 策略的测试后，得到的切换次数的数据如下。

RR 策略（t1、t2 实时优先级也都为 70）

```
……
time_off (36521618)                    t1(   2692)(prio=   29)(state=      64)--->
t2(   2691)(prio=   29)(state=0)
task:t2 PID (2691) PRI(29) EXIT
Switch_num(372)
```

从得到的结果可以看出，RR 策略下，t1、t2 总的切换次数为 372。

WRR 策略（t1、t2 设置实时优先级为 70，t1 权重 50、t2 权重为 10）

```
……
task:t2 PID (2517) PRI(29) EXIT
Switch_num(92)
time_off (37168119)                    t2(   2517)(prio=   29)(state=      64)--->
kworker/0:2(   1324)(prio= 120)(state=0)
```

从以上结果中可以看出，在 WRR 策略下，t1、t2 总的切换次数仅为 92，相比 RR 策略下，次数大大减小。这样也减少了切换所带来的系统开销。

3) 基于进程依赖的调度算法的执行时间

该测试的目的是证明：当一个 CPU 上单独专门运行某一个进程，要比将该进程和其它进程放在一起运行要快。将具有关联度的进程（比如具有数据依赖关系）绑定在一个核上，而不相关的进程将放在另一个核上运行。这样能有效的提高进程的周转时间。

首先，在/root/ws/目录下，除了 t1、t2（t1、t2 具有关联度）外，我们需要第三个进程 t3（与 t1、t2 无关联度），这三个进程的实时优先级都为 98，通过使用/proc/hooker 或者设置亲和性来决定 t3 进程要运行的核。

这三个进程都采用 RR 调度策略进行测试，首先将 t3 设置在 t1、t2 所在的核 0 上进行测试。再次将 t3 设置在与 t1、t2 不同的核上进行测试。通过在进程中，开始段和结束段的时间点，来计算进程的运行时间。运行的脚本 ms.sh，从而得到进程的运行结果。

运行的结果如图 5.3 所示，从图中可以看到 t3 单独放在另一个核中的时候，执行的时间明显的减小了。验证了之前的假设。当一个 CPU 上单独专门运行某一

个进程，要比将该进程和其它进程放在一起运行要快。将具有关联度的进程绑定在同一个核上，而无关的进程放在不同的核上将有效的利用进程的等待时间，减少进程的周转时间。

```
[root@fedora-t410 ws]# ./ms
t2 CPU 1 is set
[root@fedora-t410 ws]# t1 CPU 1 is set
t3 CPU 1 is set
The t3: 3549 is done,a=3000000000,time:53s
The t2: 3547 is done,a=3000000000,time:53s
The t1: 3548 is done,a=3000000000,time:53s
^C
[root@fedora-t410 ws]# ./ms
t2 CPU 1 is set
t3 CPU 2 is set
[root@fedora-t410 ws]# t1 CPU 1 is set
The t3: 3646 is done,a=3000000000,time:16s
The t1: 3645 is done,a=3000000000,time:34s
The t2: 3644 is done,a=3000000000,time:34s
```

图 5.3 测试进程运行结果

5.5 对比说明

综合 5.4.3 节中测试得到的数据，将得到的结果绘制成表格，以便观测结果并进行综合对比。

表 5.1 RR 策略下进程的测试结果

	调度策略（权重）	周转时间	进程切换次数
进程 t1	SCHED_RR	34s	372
进程 t2	SCHED_RR	35s	

表 5.2 WRR 策略下进程的测试结果

	调度策略（权重）	周转时间	进程切换次数
进程 t1	SCHED_WRR（50）	21s	92
进程 t2	SCHED_WRR（10）	35s	

如表 5.1、表 5.2 所示，显示了进程在分别在调度策略 SCHED_RR 和 SCHED_WRR 下得到的结果，对比分析，由于 RR 下所有进程的时间片大小都是固定的，因此采用 WRR 调度策略，能让同一优先级的进程保证轮转运行的同时，紧要的进程能够得到更大的时间片，更快的完成执行。并且在时间片大小得到设置的同时，进程的切换次数也会变小，减少了换入换出的系统开销，有利于增强系统的实时性。因此，当对关联度进程设置在同一个核上时，采用 WRR 策略可以

满足关联度进程有更紧要的进程的需求，并且减少切换次数。

5.6 结论

本章针对特定的应用模式，提出了一种实时改进方案，对具有关联度的进程进行设置，并对实时改进算法进行改进，提出 SCHED_WRR 调度算法。结合以上方案，通过测试验证了 SCHED_WRR 的功能，并在此特定应用模式下进行测试对比，将具有关联度的进程设置在同一核上，结果显示进程采用 SCHED_WRR 调度策略，比采用 SCHED_RR 调度策略，进程的周转时间和切换次数都要少。因此，在这种应用模式下，采用 SCHED_WRR 确实能提高实时性。

第六章 总结与展望

本文在 Linux3.5.4 内核源码的基础上，分析了 Linux 可抢占内核的实现：为了增强内核的实时性，对内核通过对内核互斥区加锁的方式，实现内核的可抢占，从而减少延迟的时间。并对 Linux 的进程调度框架进行了分析，展现了 Linux 是如何通过调度类的方式提供完全公平调度算法和实时调度算法的，进程是如何在所属调度类下被调度的。对当前 Linux 上的实时改进技术进行分类综述对比，从不同的角度来说明这些技术的原理，并在此基础上，提出一种基于应用模式的实时调度算法改进模型。本文的主要工作包括：

1) 根据 Linux 内核代码，分析了为了增加内核的实时性，而对内核进行的可抢占的修改实现。
2) 详尽分析了调度中的关键数据结构、Linux 调度器的组成模块以及 Linux 调度核心函数 schedule，核心调度器与调度类的联系。
3) 对 Linux 已有的实时改进技术进行分类分析，并进行对比。
4) 根据对实时改进技术的分析，针对特定应用模式，根据关联度对进程进行设置，并提出 WRR 实时调度算法，并对该算法进行测试，符合该调度策略的预期。

本文在分析成果及改进算法符合预期的同时，也存在一些不足：

1) 首先，内核源码比较庞大，对内核的一些分析存在不够且不全面的地方。
2) 对实时性改进技术分析时，并没有全面覆盖，只是选取了几种典型的技术。
3) 具有关联度的应用，目前仅依靠用户自己识别设置，未能自动识别。
4) 实时改进方案并没有以形成独立的调度类形式提供，只是在内核源码的基础上进行了修改。

参考文献

[1] 刘慧双. Linux 实时操作系统定制及设备驱动开发[D]. 华中科技大学, 2013.

[2] Robert Love.Linux 内核设计与实现[M]. 机械工业出版社, 2013:20-24.

[3] 贺永红. 基于 Linux 的嵌入式实时研究与改进[D].贵州大学, 2007.

[4] Daniel P. Bovet, Marco Cesati. 深入理解 Linux 内核[M]. 中国电力出版社，2007:89-91.

[5] William Stallings.操作系统-精髓与设计原理（第七版）[M]. 电子工业出版社. 2012: 96-97.

[6] 徐晓磊,董兆华,吴建峰等. Linux 可抢占内核的分析[J]. 计算机工程, 2003, 29(15): 115-117.

[7] 张国琛.基于 Linux 的低功耗手持设备系统的设计与实现[D]. 武汉理工大学, 2011.

[8] Wolfgang Mauerer.深入 Linux 内核架构[M]. 人民邮电出版社, 2010: 67-83

[9] 刘明. Linux 调度器发展简述[EB]. [2014-11-15]. http://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/index.html.

[10] Thomas H.cormen, Charles E.Leiserson, Ronald L.Rivest et al.算法导论[M]. 机械工业出版社, 2013: 173-176.

[11] Brandon Hall. Slot Scheduling: General-Purpose Multiprocessor Scheduling for Heterogeneous Workloads[D]. The University of Texas at Austin, 2005.

[12] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad. Semi-Partitioned Hard-Real-Time Scheduling Under Locked Cache Migration in Multicore Systems[C]// Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on. Pisa, Italy: IEEE Comp Soc, 2012: 331-340.

[13] 郭韬. 基于 LINUX 的实时语音雷达同步记录仪[D]. 四川大学, 2006.

[14] 肖振华,徐玉斌,解辉等.基于嵌入式 Linux 2.6 的实时优化[J]. 计算机技术与发展, 2008, 18(11): 83-86.

[15] C. Liu, J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM, 1973, 20(1): 46–61.

[16] R. I. Davis, A. Burns, R. J. Bril, et al． Controller area network (CAN) schedulability analysis. Refuted, revisited and revised[J]. Real-Time Systems, 2007, 35(3): 239–272.

[17] KWEI-JAY LIN, YU-CHUNG WANG. The Design and Implementation of Real-Time Schedulers in RED-Linux[C]// Proceedings of the IEEE, 2003,91(7): 1114-1130.

[18] M. Bertogna, S. Baruah. Limited preemption EDF scheduling of sporadic task systems[J]. IEEE Trans. Ind. Informat., 2010, 6(4): 579–591.

[19] Alia Atlas, AzerBestavros.Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux[C]//Real-Time Systems Symposium, 1999: 272 –276.

[20] R.Bril, J.Lukkien, W.Verhaegh. Worst-case response time analysis of real-time tasks under

fixed-priority scheduling with deferred Preemption[J]. Real-Time Systems, 2009, 42(1-3): 63–119.

[21] 王宇英.嵌入式 Linux 实时化及其应用[D]. 西北工业大学, 2003.

[22] 贺炎,刘曙霞. Linux 的进程调度策略[J]. 电子科技, 2004, (5):31-34,38.

[23] 陈敏,周兴社.基于嵌入式 Linux 的实时优化方案[J]. 计算机应用研究, 2005, 22(3): 235-237.

[24] Claudio Scordino, Giuseppe Lipari. Using Resource Reservation Techniques for Power- Aware Scheduling[C]// EMSOFT '04 Proceedings of the 4th ACM international conference on Embedded software. Pisa, Italy: ACM New York ,2004:16-25.

[25] Giuseppe Lipari, Sanjoy Baruah. Greedy reclamation of unused bandwidth constant-bandwidth servers[C]// Euromicro-RTS'00 Proceedings of the 12th Euromicro conference on Real-time systems.Stockholm, Sweden: IEEE Computer Society, 2000: 193-200.

[26] G.Yao, G.Buttazzo, M.Bertogna. Comparative evaluation of limited preemptive methods[C]// Proc. 15th IEEE Int. Conf. Emerging Techonol. Factory Autom. (ETFA'10), Bilbao,Spain: IEEE Comp Soc, 2010: 1–8.

[27] 董晓峰.嵌入式 Linux 及其调度策略的研究[D]. 西安电子科技大学, 2004.

[28] Karim Yaghmour. The Real-Time Application Interface[C]// In Proceedings of the Linux Symposium. Boston, Massachusetts, USA, 2001: 245-260.

[29] Philippe Gerum. Xenomai - Implementing a RTOS emulation framework on GNU/Linux[EB]. [2014-11-15]. http://www.xenomai.org/documentation/xenomai-2.1/html/xenomai/.

[30] 须文波,张星烨,欧爱辉等.基于 RTAI-Linux 的实时操作系统的分析与研究[J]. 现代计算机（专业版）, 2003, (5):19-21,31.

致谢

转眼间，两年半的研究生生活即将结束了。在此学习期间中，我得到了实验室很多老师和同学的帮助。这篇论文是在他们的帮助和关心下完成的，值此论文完成之际，向他们表示衷心的感谢。

衷心感谢我的导师苏锐丹副教授以及马志欣副教授。他们知识渊博、专业知识深厚，在学术上给了我悉心的指导，为我创造了良好的工作环境和自由的学习氛围。导师从论文选题到搜集资料，从写稿到反复修改都倾注了大量的心血。得益于老师们在学术领域的帮助，使我的专业水平迅速提高，顺利完成研究生的科研工作。老师们勤奋的工作作风，正直无私的品格，和蔼可亲的态度，深深地感染了我，使我终生受益。

感谢多媒体研究所的各位老师，为我们创造了良好的实验室环境，提供各种学习资源，使我们能够专心科研工作，努力提高自己的专业知识。感谢实验室的所有同学，在学习和生活中对我包容、给予我鼓励，使我能够与你们一起学习、一起进步。

感谢西电，给我带来的美好时光，不仅让我在学业上收获了知识，也让我结识了感情至深的朋友和同学。

感谢曾经教育和帮助过我的所有老师和同学。

深深地感谢我的家人，是你们给我无尽的关怀和爱护，默默地支持我完成学业。感谢你们对我的理解和无条件的支持，这一直是我不懈努力的动力。

最后，衷心地感谢为审阅本论文而付出宝贵时间和辛勤劳动的专家和老师。

西安電子科技大學

地址：西安市太白南路2号
邮编：710071
网址：www.xidian.edu.cn