兰州大学

硕士学位论文

Linux实时抢占补丁的研究与实践

姓名：吴章金

申请学位级别：硕士

专业：计算机软件与理论

指导教师：Nicholas Mc.Guire;周庆国

20100501

摘 要

本文总结了过去两年作者在扩展“Linux 实时抢占补丁”方面所从事的研究与开发工作，包含了作者对该项目所作的贡献。本文在深入分析其实现原理之后，介绍了其在 MIPS（龙芯）平台的移植与优化，并给出了评测结果与性能分析。

Linux 实时抢占补丁项目由 Ingo Molnar 于 2006 年发起，旨在整合其他团队（KURT，RED-Linux，low-latency）的工作，通过修改 Linux 让其支持完全抢占以提供实时性能。它不仅提供了 POSIX 的 API，继承了 Linux 对文件系统、网络和图形的良好支持，源代码以 GPL 协议发布，可以自由获取与修改，有良好的可移植性，目前已支持 X86、PowerPC、ARM 等平台。该项目还在开发中，不支持 Linux 所支持的所有平台，通过本文的工作，MIPS（龙芯）将得到支持。

本文在调研该项目研究进展与开发趋势后，基于其最新源码对其实现原理进行了深入学习，分析了其低延迟/自愿抢占技术、抢占技术、中断线程化、高精度时钟、实时调度策略、临界区抢占、优先级继承等实时改造技术以及 Ftrace、Perf 等实时调试与优化技术，从而更深入地理解了实时操作系统的原理与特点。

本文所采用的目标平台是龙芯处理器，它由中国设计，自第一代于 2002 年面市以来，已经发展到了第三代。市面上广泛采用的是第二代的龙芯 2F，该处理器采用 RISC 架构，基本兼容 MIPS，达到中等奔四的性能，已应用于桌面、上网本、小型服务器等领域。虽然其功耗低，但是在工业自动化、数字控制、汽车电子等领域的应用还有待拓展，而这些领域都需要实时操作系统的支持。

本文成功地移植了 Linux 实时抢占补丁到龙芯处理器平台，不仅充分验证了 Linux 实时抢占补丁的高度可移植性，而且为龙芯平台提供了一款安全、可靠、高效的实时操作系统，潜在地拓展了龙芯在上述实时领域，甚至是在国防、航空航天等领域的应用。

该工作得到了江苏龙芯梦兰科技股份有限公司的支持，相应的研究成果已经被该项目官方接收：

git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git rt/2.6.33

最新的研发进展以 GPL 协议发布于以下两个站点：

1. http://dslab.lzu.edu.cn/dslabwiki/index.php/Real_Time_Preempt_Patch(RT_PREEMPT)_for_MIPS
2. http://dev.lemote.com/code/rt4ls

关键字: Linux，实时抢占，Ftrace，实时操作系统，MIPS，龙芯

ABSTRACT

This thesis summarizes the research and development efforts conducted in the past 2 years on extending Preempt-RT. It covers the contribution of this work along with the analysis of the technological principles behind Preempt-RT - notably in the context of MIPS. Further this thesis covers the specifics of the porting to the Loongson 2F platform and concludes with benchmark results and a performance analysis.

The Preempt-RT effort, launched by Ingo Molnar in 2006, has been a patch to the mainline Linux kernel. This patch has been incorporating efforts by other groups (KURT, RED-Linux, low-latency) along the way and in many ways is a unification effort to bring real-time to mainline Linux. With the current Preempt-RT patch, mainine Linux truely is entering the RTOS domain and provides a full POSIX API. Preempt-RT is a full featured Linux extension, providing the full feature set of GNU/Linux including the network and graphical environment, extensive filesystem support, etc. Its souce code is released under GPL, freely downloadable, open to modification and portable - with support for x86, PowerPC, ARM, etc. Preempt-RT is still under development, and its full merge into mainline will still take some time, as such it currently does not yet support all platforms that Linux supports, but with this work MIPS has been added back to the list of architectures supported in Preempt-RT.

Based on the investigation of its research progress and development trend, this work studied the latest source code of Preempt-RT and analyzed the core real time technologies notably: low-latency patch/voluntary preemption patch, preemption patch, interrupt threading, high resolution timer sub-system, real time scheduling policy, preemptible critical sections, priority inheritance and the related real time debugging and optimizing technologies: Ftrace and Perf. Through this work the author has achieved a well founded understanding of the principle and features of real time operating system in general and the specific implementation in mainline Linux more deeply.

The target CPU for this work is the Loongson processor which is a processor developed in china. Its 1st revision, the loongson 1A, was released in 2002 and now stands at the transition to the third generation extending the Loongson to 64bit multicore CPUs. The revision most used in the market currently is the Loongson-2F, a MIPS compatible RISC architecture with a performance comparative to the Pentium V though at conciderably lower power-consumption. It has been used in the market of desktop, netbook and small servers, including NAS. Although it has low power consumption and thus is potentially suitable for the application in industrial automation, digital control as well as automotive electronics and other areas,

exploitation in these areas all need the support of a real time operating system.

The main goal of this work was porting Preempt-RT to the Loongson platform. This not only proved that Preempt-RT patch is highly portable but also provided a safe, reliable and efficient real time operating system for the Loongson 2F CPU. This not only brings MIPS support back to Preempt-RT but also potentially expanded the application of the MIPS processor and specifically the Loongson 2F in the above areas and even others such as national defence and Aero-space.

This research project has been supported by Lemote and has in the meantime been accepted by the maintainer of Preempt-RT:

git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git rt/2.6.33

The latest research results are published and maintained under terms of GPL on the following sites:

1.http://dslab.lzu.edu.cn/dslabwiki/index.php/Real_Time_Preempt_Patch(RT_PREEMPT)_for_MIPS

2.http://dev.lemote.com/code/rt4ls

Keywords: Linux, Preempt-RT, Ftrace, Real Time Opearting System, MIPS, Loongson

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，均已明确著名出处。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：吴章金　　日期：2010年5月24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。

保密论文的解密后应遵守此规定。

论文作者签名：吴章金 导师签名：[signature] 日 期：2010年5月24日

第1章绪 论

1.1 研究动机

实时操作系统发展非常迅速，特别是随着嵌入式计算机的出现，以及其在工业自动化、数字控制、汽车电子等领域的广泛应用，促进了实时操作系统的发展。而包括POSIX 1003.1b、Real-Time Specification for Java、OSEK(automotive RTOS standard)、Ada83、Ada95在内的各种实时标准则加速了这一发展。

最近几年，随着低延迟补丁、抢占补丁、O(1)调度器、优先级继承Mutex、高精度时钟系统、可抢占RCU进入Linux，Linux的实时性能不断提升，由于其不仅功能丰富，集成POSIX编程环境，支持多种硬件平台，支持模块化，可剪裁与定制，有强大的社区支持，而且遵循GPL协议，因此相对于其他价格昂贵、源代码受到专利保护的商业实时操作操作系统而言，Linux赢得了很多公司和科研院所。

但是Linux本身还无法满足硬实时操作系统的所有需求，所以才出现了各种各样的实时Linux扩展。当前比较流行的实时Linux扩展莫过于RTAI[4]、Xenomai[5]、XtratuM[6]和实时抢占补丁[7]。资料[1][2][3]对上述四种实时操作系统进行了测试，其结果表明，四者都具备了一定的硬实时能力：

- RTAI

虽然在实时响应时间方面RTAI比实时抢占补丁有一定优势，但是RTAI在高负载下出现死机，这对一个严格的硬实时操作系统而言是不允许的，另外RTAI还不支持POSIX。RTAI有较大的开发社区和较多用户，但是支持的平台有限，而且支持的内核版本较老。

- Xenomai

Xenomai在实时性能方面都不如RTAI，但在高负载下比RTAI更稳定，它还提供了一种叫"皮肤"(Skin)的技术，更容易从其他实时方法或者实时操作系统中迁移实时应用程序到Xenomai，这意味着Xenomai支持更多API，比如POSIX、RTAI、甚至商业的Vxworks。Xenomai在可确定性方面比实时抢占补丁有优势，虽然在平均性能方面略逊一筹。Xenomai在社区支持和用户群方面跟RTAI有些类似。

- XtratuM

XtratuM不仅支持POSIX，在实时性能方面比实时抢占补丁更有优势，而且本身被实现为一个轻量级Hypervisior，容易在可靠性方面进行形式化验证，其上可以运行多个实时与非实时操作系统，实现不同操作系统间的隔离，提高实时任务的安全性，并且基于XtratuM的Hypervisor特性，还能通过操作系统冗余实现系统容错。不过XtratuM目前仅支持X86、LEON2等很有限的几个平台，而且其用户与开发团队相对比较薄弱。

- Linux实时抢占补丁

Linux实时抢占补丁的实时性能不如前三者，但已具备一定的硬实时能力。它有更强大的社区支持，支持更多平台，并且越来越多的实时抢占补丁不断被官方Linux接收，预计不久后将完全进入官方Linux。由于其支持POSIX，各种外部设备驱动无需重写，实时应用程序与普通应用程序开发过程差异很小，因此，其上的实时应用开发更容易。另外，其平台相关性小，具有高度可移植性。

就实时技术而言，前三者在一定意义上都是对RTLinux[8]的继承，通过引入中断虚拟化技术和双核技术实现实时与非实时系统的分离，本质上是在Linux之外重新实现了一个实时操作系统，而实时抢占补丁则不一样，它通过修改Linux本身，让其支持完全抢占以原生提供实时支持，是对一个通用操作系统的实时改造，并未重新开发一个系统。

通过对近几年Real Time Linux Workshop[9]论文集的分析，并根据自己一年多来参与Linux自由软件社区的经历发现，Linux实时抢占补丁不仅有庞大的内核开发团队的支持而且有大量的企业用户，因此具有非常广泛的发展前景。

另外，通过对近几年国内优秀硕士、博士论文的检索发现，国内对RTLinux（包括后续的RTAI、Xenomai以及XtratuM）等都有研究，但对Linux实时抢占补丁的研究较少，因此，研究该项目将有以下几方面的意义：

- 深入分析它的各种实时改造技术，了解实时操作系统与通用操作系统的异同，理解实时操作系统的原理与特点。
- 针对特定平台移植与优化验证其可移植性，并通过总结移植与优化的过

程，为其他平台的相关工作提供潜在的借鉴。

- 对移植与优化结果进行全方位的性能评测，评估特定平台上实时抢占补丁的实时性能，并为相关人员在选择实时系统时提供一定的参考数据。

1.2 文章结构

本文的章节安排如下：

第一章 绪论

提出了本文的研究动机并规划了相应的文章结构。

第二章 实时操作系统概述

在介绍实时操作系统的基本概念之后分析了其性能指标，进而讨论了其基本需求与 POSIX 兼容性。

第三章 实时抢占补丁研究

在分析传统 Linux 不足之后，比较了各种 Linux 实时解决方案，调研了 Linux 实时抢占补丁项目的研究进展与开发趋势并总结了其优缺点；基于最新源码深入分析了其低延迟/自愿抢占技术、抢占技术、中断线程化、高精度时钟、实时调度策略、临界区抢占、优先级继承等实时改造技术以及 Ftrace、Perf 等实时调试与优化技术。

第四章 实时抢占补丁移植

深入分析了 Linux 实时抢占补丁的平台相关性，并基于最新源码详细讨论了 Ftrace 在 MIPS 平台的实现以及 Linux 实时抢占补丁往龙芯处理器平台的移植。

第五章 实时抢占补丁优化

讨论了实时操作系统的基本优化方法以及 Linux 实时抢占补丁在龙芯平台的优化过程。

第六章 性能评测与分析

探讨了实时操作系统的性能评测原理，调研了相应的评测方法与工具，并对本文移植与优化的龙芯平台的 Linux 实时抢占补丁进行了性能评测，最后对评测结果进行了系统地分析并与 VIA 平台上的 Linux 实时抢占补丁的实时性能进行了比较。

第七章 总结与展望

对本文工作进行了总结并展望了后续工作。

第2章实时操作系统概述

本章将阐述本文的理论基础，包括实时操作系统的基本概念，性能指标，以及达到这些性能指标的基本需求与相应的 POSIX 兼容性。

2.1基本概念

2.1.1 实时系统

资料[10]给出了实时系统的定义：

“实时系统是指计算的正确性不仅依赖于逻辑的正确性而且依赖于产生结果的时间，如果系统的时间限制不能得到满足，系统将会产生故障。”

如果这种故障是灾难性后果，例如造成重大人员伤亡、财产损失或者环境污染，那么该系统可称为硬实时系统。比如用于控制航空器的嵌入式系统，如果不能及时处理任务，将可能造成航空器的失控进而造成航天员的牺牲与航空器的损毁；对于核反应堆，化学电场的控制系统，如果不能及时处理某些任务，则可能造成严重的环境污染。

如果这种故障不带来灾难性后果，只造成性能等方面损失，则可称为软实时系统。例如播放高清视频，如果系统不能在规定时间内解码一定数量画面，那么播放效果会很差但不会造成灾难性后果；而对于个人 PC 的桌面系统，如果不能及时处理用户按键，用户体验会很差，但并不会造成灾难性后果。

因此，相比于软实时系统，硬实时系统要求更加严格的时间限制以防止灾难性后果发生。图 2-1充分反映了硬实时、软实时之间的这种区别[1][11]。

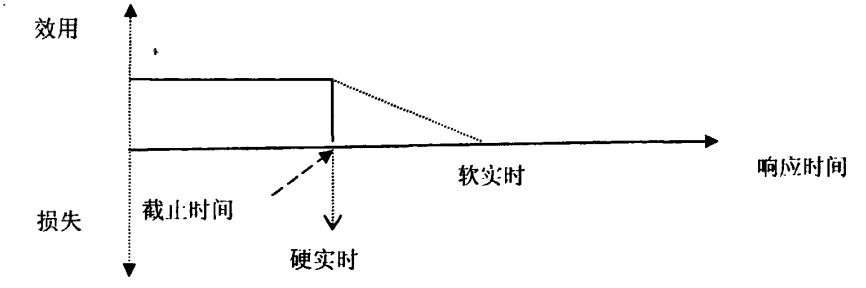

图2-1 硬实时与软实时示意图

2.1.2 实时操作系统

实时操作系统作为实时系统的核心组成部分，它应该遵循实时系统的要求，POSIX标准 1003.1b[12]给出了其定义：

“该操作系统有能力提供一个指定范围内的服务响应时间。”

那么到底什么是系统服务响应时间，该服务响应时间包括哪些部分，又如何

界定呢？下面通过分析“服务响应时间”和“指定范围”两个术语来理解实时操作系统的概念并引出实时操作系统的性能指标、基本需求以及 POSIX 兼容性。

2.2 性能指标、基本需求和POSIX兼容性

下面从实时操作系统的概念入手，以 Linux 操作系统为例，介绍不同系统任务的服务响应时间的定义、构成与界定方法，进而总结实时操作系统的性能指标。

2.2.1 服务响应时间

响应时间即Latency，被定义为“the time that elapses between a stimulus and the response to it”[13]即从外部刺激发生到作出反应之间经过的时间。

外部刺激对于实时操作系统来说可能是各种外部中断事件、任务本身设置的定时器到期通知或者是其他任务发出的信号。

作出反应意味着该任务得到运行并执行完，不仅关系到任务启动时间也关系到任务完成时间。任务启动时间是指从外部刺激发生到任务得到调度，而任务完成时间等于启动时间加上执行时间，因此整个服务响应时间可以分为两部分来考虑，一部分是任务启动时间，另一部分是任务执行时间。

2.2.1.1 任务启动时间

任务启动时间跟外部刺激源有关，下面从三种不同刺激源来讨论。

1. 外部中断事件

如果任务由外部中断事件触发，那么这类任务通常是非周期性任务，其系统服务响应时间从外部中断发出到相应的任务启动，这段时间包括中断延迟、中断处理时间、调度器延迟、任务调度时间，如图 2-2[14]。

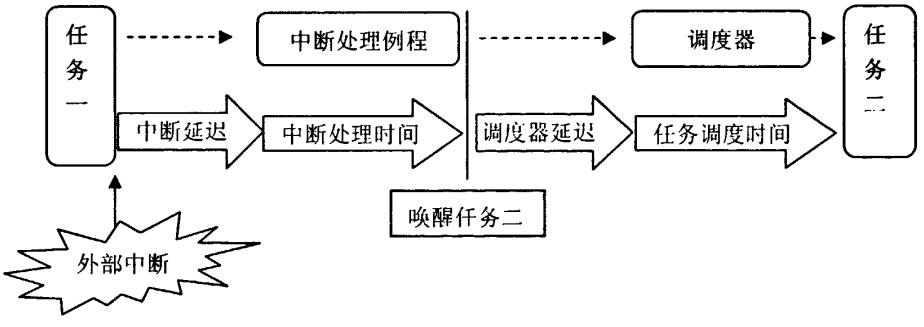

图2-2 外部中断触发的任务响应过程

- 中断延迟

中断延迟从外部中断发生到中断处理例程得到执行，主要包括中断的硬件响应和软件响应两部分。

对于X86，硬件响应过程包括保存中断（或异常、系统调用[1]）发生时的相应地址、状态寄存器、栈指针，如果有硬件错误码也保存到栈中，然后找到中断（异常、系统调用）的入口地址，转到软件处理部分，软件处理返回后，依次恢复保存在栈内的各类信息，返回到之前保存的地址处继续执行。如图 2-3所示[15]：

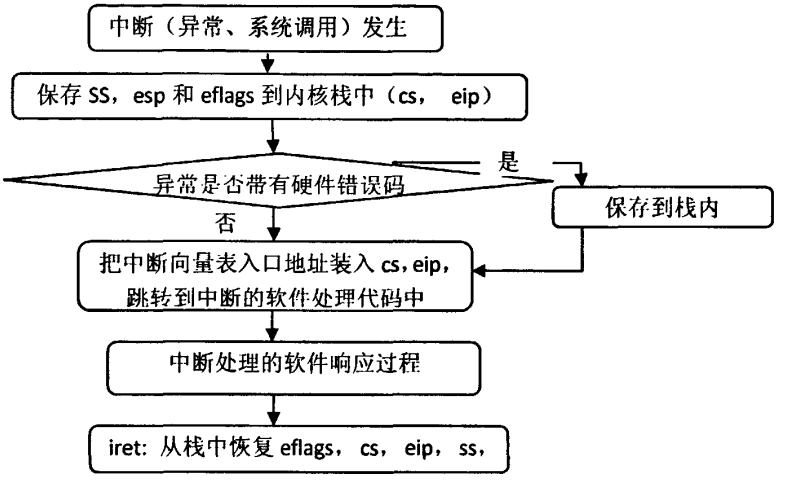

图2-3　X86 平台中断（异常、系统调用）的硬件响应过程

至于软件处理过程，先保存中断（异常、系统调用）上下文，后进行中断（异常、系统调用）处理，根据中断号（异常号、系统调用号）执行相应的处理例程。如果是中断与异常，从处理例程返回后恢复中断（异常）上下文，直接回到硬件响应部分；如果是系统调用，判断是否有任务需要抢占，如果需要则启动调度器调度任务，否则判断是否有信号要处理，如果有需要则处理信号否则恢复上下文，返回硬件响应部分，进而恢复整个处理过程[2]。如图 2-4所示[15]。

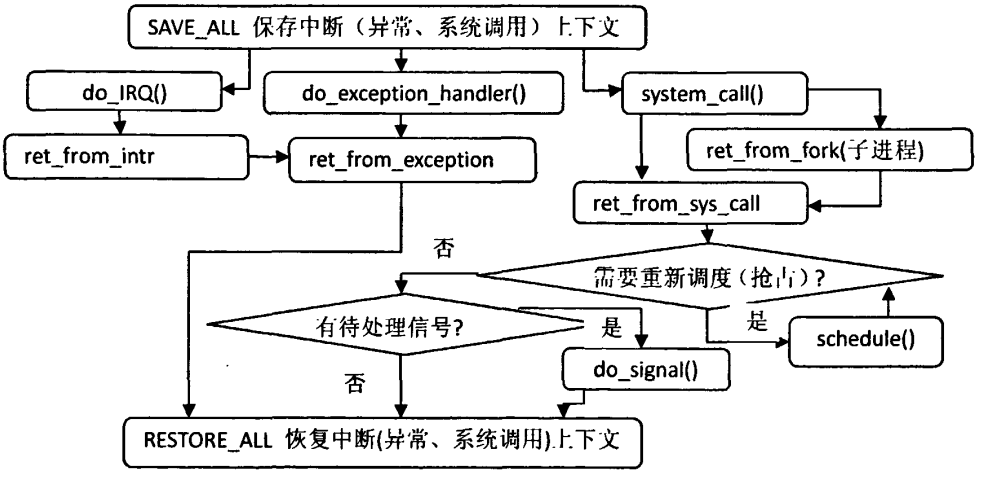

图2-4　X86 平台中断（异常、系统调用）的软件响应过程。

[1] 由于中断、异常和系统调用的处理过程类似，而且后文需要相关基础，所以一并讨论。

[2] 这里仅考虑配置为 PREEMPT_NONE 的情况，对于配置为 PREEMPT 的情况会在下一章详细介绍

- 中断处理时间

指中断处理例程中确切地用于处理中断的时间。对于Linux而言，即用户通过setup_irq()，request_irq()或者request_threaded_irq()注册的中断处理结构体irqaction中的中断处理函数handler所执行的时间[3]。

- 调度器延迟

指从中断处理例程执行完到刚进入到调度器所经过的时间。如图 2-4所示，当内核配置为PREEMPT_NONE时，中断处理后直接返回，期间没有启动调度器的机会而是直接执行被中断的下一条指令。此时，调度器只有在下一次时钟中断发生或系统调用返回时才有机会启动。

- 任务调度时间

包括调度器选择任务并切换到该任务所花费的时间，主要有两部分：一部分是调度器调度策略（算法）从任务就绪队列中选择一个任务的时间开销，对于 Linux 内核，则是 pick_next_task()所花费的时间。

当调度策略选择下一个任务后，会发生正文切换，进而启动新任务。因此，另外一部分是正文切换的时间开销，对于 Linux 内核，即 context_switch()的开销，而 context_switch()主要完成 switch_mm()和 switch_to()，前者换进相应进程的页表，换进页表后需要刷新 TLB（页表缓存）甚至是 Cache，后者换进寄存器和栈。这两部分都跟特定处理器相关，包括寄存器个数，Cache 与 TLB 的组织，TLB 的表项规模（跟内存和页面大小有关）等。

2. 任务本身设置的定时器到期通知

通过以上介绍，对于由外部事件触发的非周期性任务，其服务响应时间主要包括中断延迟、中断处理时间、调度器延迟、任务调度时间。而对于某些周期性任务，通常是通过任务本身设置的定时器到期通知触发，例如：

```
while (1) {
    /* do something */
    sleep(2);
}
```

上述任务要求每隔 2s 周期性地运行。任务是否能够每隔 2s 及时地启动，跟底层 sleep()函数的实现有关。

Sleep()函数会引起任务进入不可运行状态，允许处理器挂起任务并执行其他任务直到睡够指定时限再唤醒任务继续执行。至于睡眠时限的设置，关联到一个定时器，先为该定时器设置一个到期时限，当定时器计数到期后，任务得到通知继续执行。定时器是一个硬件或软件设施，允许一些函数在指定时限后激活。通常，除了硬件定时器以外，还需要软件定时器以满足多任务的定时要求。而定时器需要底层硬件时钟设备（时钟计数器）的支持。

因此，任务能否通过 sleep()函数在指定时限后唤醒，关系到系统定时器的管

[3] 对于线程化的中断，可以主动放弃处理器资源，因此中断处理时间是中断开始执行到开始调度的时间。

理，而定时器的管理由时钟管理系统完成，因此，时钟管理系统的精确性决定了周期性任务的服务响应时间，下面讨论时钟精确性。

- 时钟精确性

绝大部分实时任务的执行周期都很短[16]，要求有高精度时钟管理系统以获取高精度时间并精确地睡眠。例如一个任务期望在下一个 200us启动，可以容忍100us延迟，那么实时操作系统必须准确地获取当前时间并精确地控制调度时限。

关于时间的获取，涉及到时间获取函数的实现。不仅涉及到时间获取函数本身的开销而且涉及到时钟源的精度。如果时间获取函数本身的开销很大，则获取到的当前时间跟实际时间偏差很大。如果时钟源的精度很低，比如基于传统的Jiffies，精度为时钟滴答的精度，即 1/HZ，如果 HZ 被设置为 1000，那么时钟源的精度只有 1ms，即每次获取到的时间跟实际时间可能存在 1ms 左右的偏差。如果直接访问硬件时钟源，那么时间精度决定于硬件时钟的计数频率，比如硬件时钟频率为 400MHz，那么可精确到 2.5ns。

关于睡眠时限的设置，关系到定时器的实现。如果软件定时器通过时钟滴答来维护，即通过时钟滴答的计数来实现定时，而时钟滴答根据底层硬件时钟的计数来产生，那么软件定时器的精度也决定于时钟滴答的精度，如果时钟滴答的频率是 1000，那么软件定时器精度也只有 1ms，类似地，如果直接使用硬件定时器，那么对于 400MHz 的硬件时钟，定时精度能够达到 2.5ns。

睡眠时可能设置定时器，释放处理器资源，定时器到期后产生时钟中断，触发处理器重新调度该任务，因此，睡眠的精确性不仅关系到定时器的实现，还跟非周期性任务类似，需要考虑时钟中断延迟、中断处理时间、调度器延迟以及任务调度时间，如图 2-5所示。

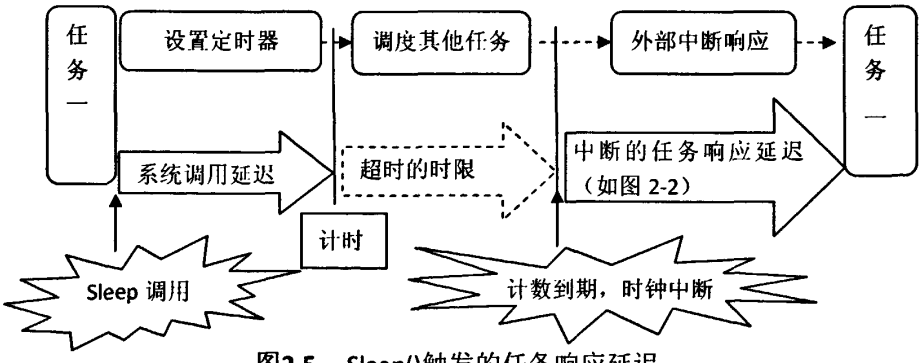

图2-5　Sleep()触发的任务响应延迟

3. 其他任务发出的信号

以上仅考虑了单任务，但具体应用可能涉及到多任务协作，整个系统服务响应时间不是取决于单一任务的响应时间，而是取决于第 n 个任务的响应时间。

对于多任务的应用，由于任务间存在潜在的事件等待、资源共享与通信，第 n 个任务需要等待前面的任务发出信号、释放资源与传输数据后才能启动，因此第 n 个任务的启动时间从其他任务发出信号（释放资源、发送完数据）到该任务接收到信号（获得资源、接收到数据）。因此服务响应时间还应考虑多任务间各种同步、互斥与通信措施的延迟。另外，在高优先级与低优先级任务共享资源时，

可能存在优先级反转现象带来不可预测的延迟，需要采取措施避免不可预测的优先级发转问题。

- 同步、互斥和通信延迟

例如两个任务共同协作完成某个工作，之间存在通过信号等待事件、通过信号量共享资源、通过消息队列传输数据，那么相应的延迟如图 2-6所示。

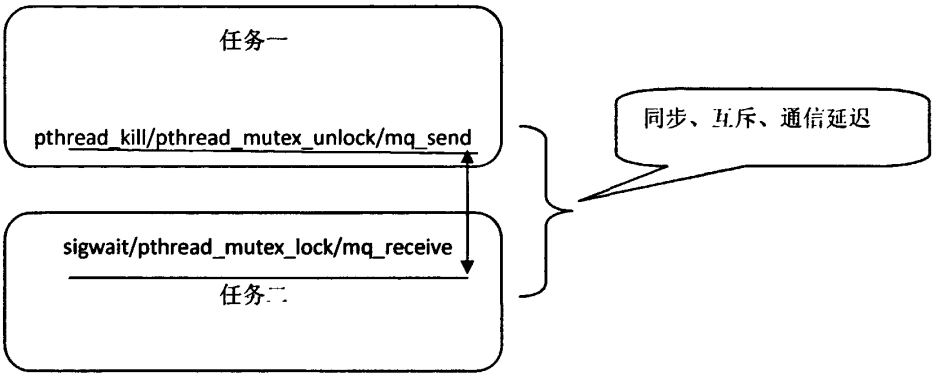

图2-6　同步、互斥和通信延迟

对于同步、互斥和通信的具体处理过程，仅以实时信号为例进行描述。当任务一通过pthread_kill()发送信号后，会进行系统调用，在信号队列中插入该信号，从系统调用返回时，选择优先级最高的信号唤醒对应任务，如果发给任务二的优先级最高，则唤醒任务二，设置任务二为就绪状态，之后检查是否有任务需要调度，从而有机会唤醒任务二，其过程如图 2-7。

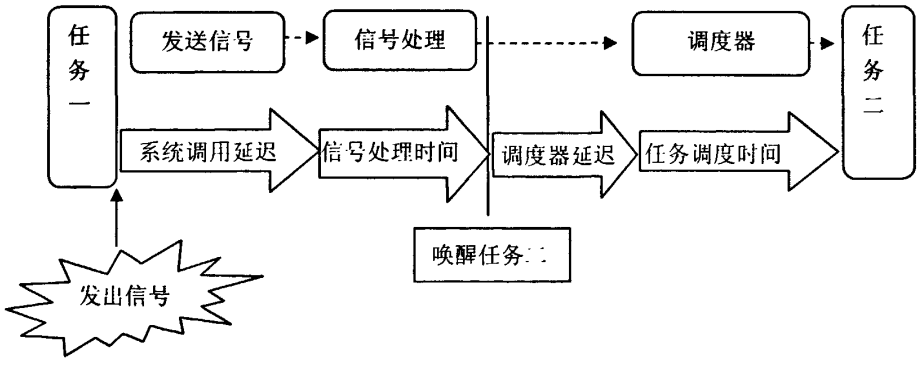

图2-7　信号触发的任务响应延迟

- 优先级反转

优先级反转[18]是指高优先级任务运行时必须等待低优先级任务。这种情况通常在高优先级与低优先级任务共享资源时发生。当高优先级任务试图获得低优先级任务已占有的资源时将被阻塞。这种优先级的反转现象是无法避免的，需要避免的是不可预测的优先级反转现象（如图 2-8）：当高优先级任务(A)被阻塞时，如果有中间优先级任务(B)要执行，它将立即抢占低优先级任务(C)，此时低优先级任务由于被抢占而无法释放资源，因此，高优先级任务将一直被阻塞到中间优

先级任务执行完，由于中间优先级任务的执行时间不可预测，因此，这种优先级的反转将导致高优先级任务的响应时间不可预测。

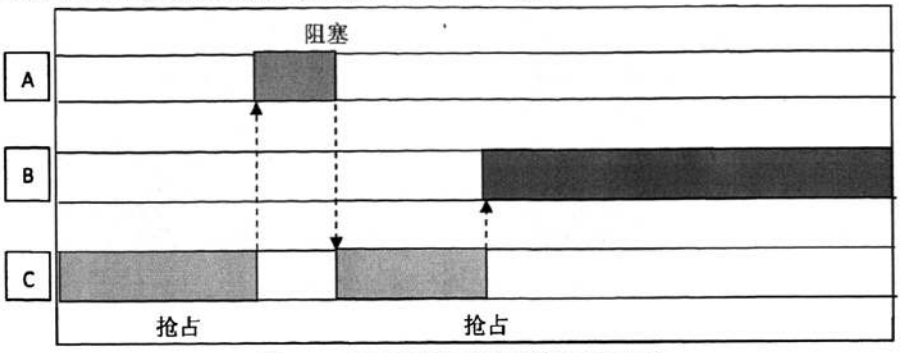

图2-8　不可预测的优先级反转现象

一般的实时操作系统都会采用优先级继承协议[17]来避免这种不可预测的优先级反转，例如，优先级继承协议会在高优先级任务发现锁被其他进程拥有后，提升其他进程的优先级为自身优先级，从而防止中等优先级任务抢占，让低优先级任务尽快执行，释放资源后恢复原优先级，如图 2-9；而优先级天花板协议[19]则根据静态分析确定一个资源锁的可能拥有者的最高优先级，然后把该资源锁的优先级顶棚值设置为该确定值，每当进程获得该资源锁时，就把进程的优先级设置为该资源锁的优先级顶棚值。

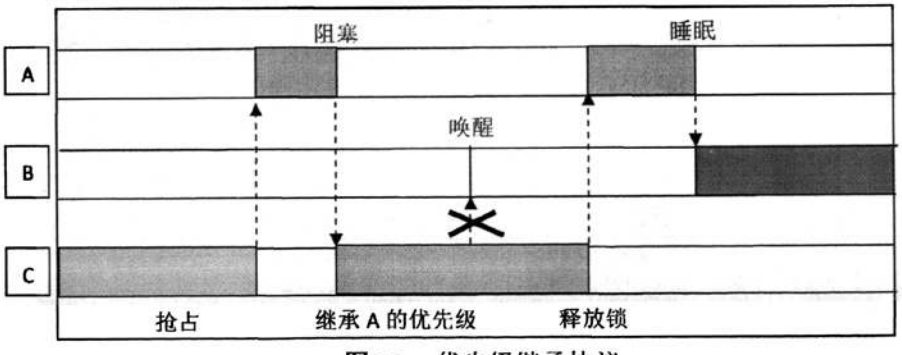

图2-9　优先级继承协议

出现优先级反转时的同步、互斥和通信的延迟情况如图 2-10所示：

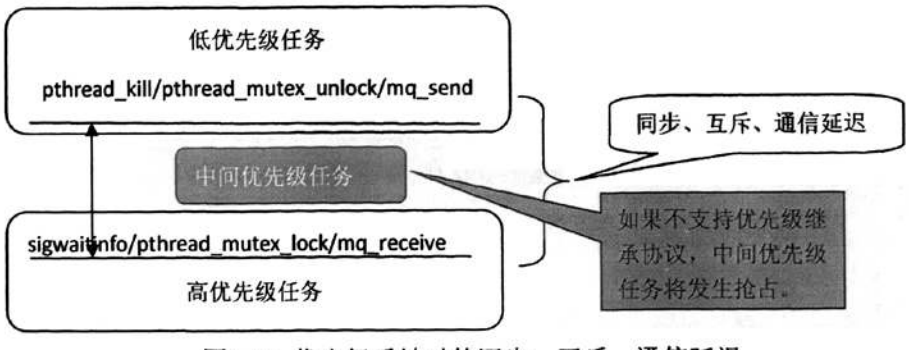

图2-10 优先级反转时的同步、互斥、通信延迟。

2.2.1.2 任务执行时间

任务执行时间与具体应用相关，比如，内存中有 1G 左右的数据待保存到磁盘中，涉及到大量磁盘 I/O 操作（或者 DMA 操作），执行时间会很长，又比如通过并口打开某个外部设备的电源通常只需要很少的几个 I/O 操作，执行时间会很短。这两个例子都是 I/O 紧密型任务，而另外有些是 CPU 紧密型任务，比如大数乘法，如果简单地计算两个数的乘积，执行时间可能比较短，又比如气象预测，则可能涉及到大量数据计算，需要消耗大量处理器资源，执行时间会很长。那么任务的执行时间到底包括哪些部分，跟哪些内容相关呢？

首先看一下 Linux 操作系统的基本结构。

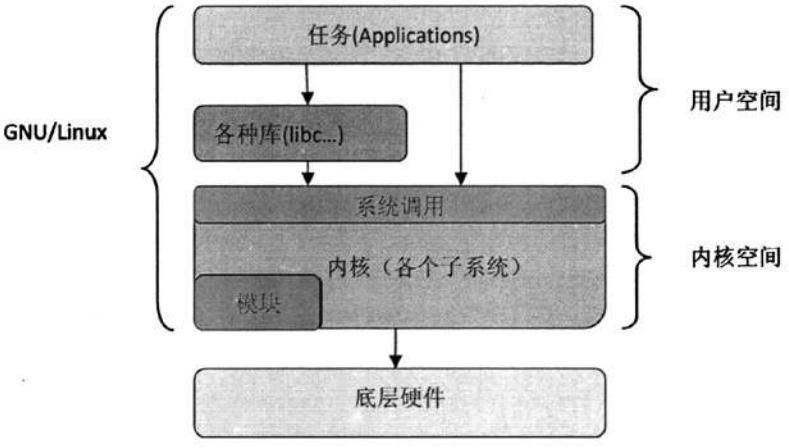

图2-11 GNU/Linux 基本结构

通过上图发现，任务执行时不仅执行任务本身还涉及到对各种系统库的调用，并通过系统调用进入到内核空间访问到各个内核子系统以及模块中的各种设备驱动，最后访问到底层硬件。因此，任务的执行时间包括如下几个部分。

1. 任务本身的执行时间

首先与任务的运算类型有关，比如四则运算、条件判断、赋值、I/O 操作中的一种或者多种运算的叠加。

其次与任务的规模有关，即任务处理的数据量，数据量越大，执行时间越长。

再者，与任务计算时采用的算法有关，算法时间复杂度越大，执行时间越长。

2. 系统库

系统库封装了各种公共操作，比如标准输入输出、线程库、数学函数库以及实时库等，对于实时任务的开发，为节省开发时间、节约开发成本，一般都会基于现有系统库来开发。因此，任务执行时间也包含相应库函数的执行时间，库函数的实现效率越高，执行时间越短。

3. 内核

操作系统用于管理所有底层硬件资源，并按照一定策略为上层任务分配这些

资源。上层任务通过内核提供的系统调用接口来访问底层资源。因此，任务执行时间还包括任务通过系统调用进入到内核后访问相关资源时所花费时间，执行时间的长短与系统调用的开销、在内核中访问的资源类型和数量以及内核对这些资源的管理方式有关。

- 系统调用

系统调用作为上层应用与内核间的接口，是上层应用与各种系统库访问内核所管理资源的唯一方式。系统调用过程在“中断延迟”部分作了介绍，发生系统调用时，将从用户空间陷入到内核空间，需要保护系统调用上下文，进行参数传递，执行完内核空间的工作，进行系统调用返回时，需要把运行结果返回给用户空间，并恢复系统调用上下文。有些平台通过VDSO机制[40]把系统调用映射到进程空间从而避免复杂的系统调用开销。

- 调度（访问处理器资源）

对于任务启动时间，其调度器延迟部分要求任务及时地申请到处理器资源，对于任务执行时间，由于内核负责对任务进行调度，假设任务已经启动，内核如果能够保证任务持续执行而不被其他任务抢占，那么该任务执行时间将会得到保障，但是，如果任务执行时持续地被其他任务抢占，那么任务执行时间将变得很长甚至不可预测。

- 内存管理（申请内存资源）

任务请求内存资源时，内存资源的分配时间也将影响任务执行时间，因此要求内核提供可确定性的内存分配算法，比如静态分配算法或者是可确定性的动态内存分配策略。

- 虚拟内存管理（访问内存资源）

虚拟存储器是一个大容量存储器的逻辑模型，不是任何实际的物理存储器，它借助于磁盘等辅助存储器来扩大主存容量，使之为更大或更多的程序所使用。虚拟存储器是解决存储容量和存取速度矛盾的一种方法，也是管理存储设备的有效方法。有了虚拟存储器，用户无需考虑所编程序在主存中是否放得下或者放在什么位置。当 CPU 用到的数据或指令不在主存时，出现缺页，此时要求从外存调进包含有这条指令或数据的页面，假如主存已被全部用完，就需要采取替换策略把主存中的一些页面置换到磁盘中。由于访问主存较访问磁盘要快 1 千倍左右，如果缺页次数很多，那么会导致任务执行时间增长。因此，需要确保内核能够有效管理虚拟存储器，并采取相应措施避免某些任务的缺页，比如采用内存预留策略，确保任务所使用的内存页面不被置换到磁盘，从而减少该任务执行时间。

4. 硬件

- 流水线与分支预测

现代处理器大多采用流水线设计以实现指令的并行执行以增加单位时钟周期内的指令执行数从而提高处理器的运行速度。如果流水线中存在条件分支指令，那么条件分支指令后的周期将会损失，因此会引入分支预测技术弥补损失的周期，削弱控制相关，防止流水线断流。但是分支预测的准确度不可能达到100%，如果分支预测失败，那么预取的数据都必须作废，重新开始取指令执行，

从而带来很大性能损失，如果流水线级数较长，这种损失将更大。因此，无论是任务本身还是内核，在实现时都必须通过一定的措施减少分支预测失败。

- Cache 命中

Cache 作为一种高速缓冲存储器，是为解决 CPU 和主存间速度不匹配而采用的技术。Cache 介于 CPU 和主存之间，是一个小容量存储器，但是存取速度比内存要快 5-10 倍，Cache 能高速地给 CPU 提供指令和数据，从而加速程序执行，当 CPU 从主存取数据时，同时给 Cache 和内存发出数据地址，如果 Cache 存有该地址的一份数据，直接从 Cache 读取，如果不存在就需要访问内存。从 Cache 读取称为 Cache 命中，而一个程序执行期间通过 Cache 进行数据存取占据整个数据存取操作（访问 Cache 和访问内存操作的和）的比例称为命中率。由于访问 Cache 更快，如果 Cache 命中率越高，那么程序执行时间越短，反之，执行时间越长。Cache 命中率不仅与硬件 Cache 的设计（Cache 与主存的地址映射关系、替换策略、写策略等）关联，而且跟应用程序的设计有关。有些处理器的 Cache 由内核来管理，因此，内核也可以提供类似内存预留的策略来预留 Cache，即确保某些任务的 Cache 不被置换到内存从而有效减少任务执行时间。

- TLB 命中

类似于 Cache，TLB 是虚拟内存管理中的页表缓存，用于加速程序地址和物理地址间的转换，当内存管理单元(MMU)试图访问 TLB 进行程序地址到物理地址转换时，如果 TLB 命中，那么将直接通过 TLB 中的页表缓存进行转换，否则需要进行页表异常处理，采用页表置换策略从内存中加载对应页表并进行转换。如果 TLB 命中，这种地址转换过程将很快，反之则会因为额外的页表异常处理和访存操作而导致页表转换变慢，程序执行时间也会加长。类似于内存预留和 Cache 预留，为确保某些程序的页表一直驻留在内存中，可以考虑采用采用 TLB 预留策略。另外，由于硬件提供的 TLB 缓存有限，如果能够减少需要缓存的页表表项，那么程序所属页表被替换到内存的概率就会降低，而页表表项数由内存大小和页面大小决定，因此，对于一块固定大小的内存，可以通过扩大页面大小来减少页表数，从而提升 TLB 命中率。当然，增加页面大小将增加内存碎片减少内存利用率。

- 地址空间和指令长度

随着内存价格越来越便宜，内存大小不断增加，但是受限于指令长度，地址无法直接在短跳转指令中编码，因此对于跨越地址空间的函数调用，处理器会提供长跳转（通过寄存器跳转），但是长跳转执行时间比短跳转要长，因此任务执行时应减少跨越地址空间的函数调用。比如，在目前的 MIPS 内核中，内核和模块的地址空间不一致，从模块空间访问内核空间时，会进行长跳转，因此为了减少长跳转开销，可以考虑把模块编译到内核中。

上面从应用程序本身、系统库、内核、硬件等几个层面考虑了程序的执行时间，实际上，由于程序启动过程的相关操作也涉及到可能的条件分支、Cache 访问、TLB 访问等，因此，程序启动时间也跟它们紧密相关。

2.2.2 时间限制

“指定范围”意味着系统必须在指定时间范围内提供服务，即系统服务响应时

间必须介于一个指定时间范围内，不能超前也不能延后。

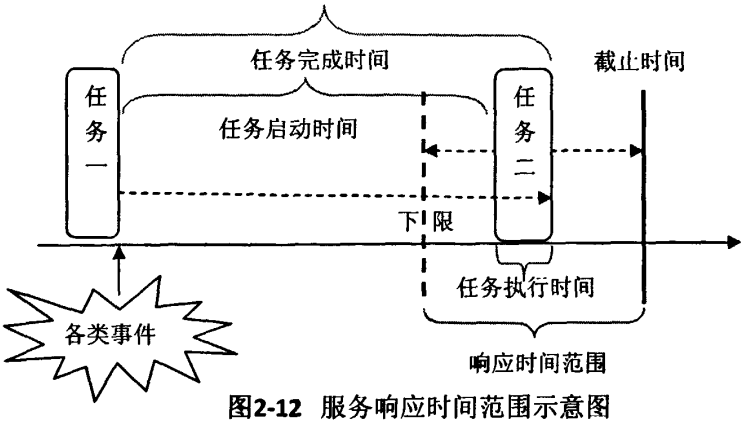

图2-12 服务响应时间范围示意图

如图 2-12所示，任务响应时间即任务完成时间，包括任务启动时间和任务执行时间。截止时间即任务最迟必须在该时间点前完成，而虚线所示为任务完成时间的下限，即任务最早必须在该时间点后完成，对于有些实时任务，下限的理想值是 0，即在事件发生后就可以响应，但实际上，由于中断延迟、中断处理时间、调度器延迟、任务调度时间、任务执行时间都大于 0，因此下限不可能为 0。

下面以两个例子来对上图进行解释：

资料[13]中提到的汽车安全保障系统（airbag），如果汽车发生碰撞，安全气囊中的传感器监测碰撞强度，当其超过预设值就产生中断，使控制器发出点火信号以触发气体发生器，气体发生器接收到点火信号后，迅速点火并产生大量气体给气囊充气以防驾驶员碰撞到坚固装置避免人员伤亡。考虑到碰撞时汽车运行速度可能很快，因此不仅要点火顺利、气囊完全膨胀，而且要确保控制器在一定时限内发出点火信号以触发气体发生器，否则驾驶员可能会在空气囊完全膨胀前碰撞到坚固装置。因此，为有效保护驾驶员的人身安全，需要保障实时任务尽可能快地启动（getting started as quickly as possible），并确保该实时任务尽可能快的完成（getting done quickly once started）[16]。

假设汽车以 360 公里的时速[4]运行，驾驶员离车前挡风玻璃距离为 50 厘米，如果汽车突然发生碰撞，那么驾驶员将在 5ms内撞上车前挡风玻璃。假设气囊离驾驶员只有 25 厘米，那么气囊响应时间只有 2.5ms。假设传感器采集信息、点火以及气囊完全膨胀所花时间是 2ms，那么控制器任务完成时间必须在 0.5ms 即 500us以内，这 500us即上图中的截止时间，这里没有下限，理想情况下是 0us。假设该任务执行时间是 1us（发出点火信号一般是简单的I/O操作），那么任务启动时间必须在 0us到 499us。

而另外一种应用情况则是数控机床控制系统，假设数控机床通过步进电机控制，步进电机转速是 12000 转/min，而步进电机以每转四步转动，即可以通过驱动输入四个控制码来控制步进电机转动一圈，那么每两步间间隔是 1.25ms，假设步进电机的轴（包括齿轮）周长是 2ms，假设加工仪器的精度要求是$_{+}$0.4ms，那么每步精度必须控制在 1.25*0.4/2=250us，因此任务必须在 1.25ms$\pm$250us 时

[4] 我国《道路交通安全法实施条例》规定高速公路最高车速不得超过 120 公里每小时.

限内完成，所以该任务截止时间是 1.5ms，而下限是 1ms，假设任务执行时间是 1us（一个简单的 I/O 操作），那么任务启动时间在 $1.25ms_{\pm}245us$。

比较上述两种情况，前者对于下限没有要求，而后者则要求有下限。对于前者，为满足截止时间，必须尽可能地减少中断延迟、中断处理时间、调度器延迟、任务调度时间以及任务本身的处理时间。对于后者，需要在确保定时器精度的基础上，尽量减少系统调用延迟、中断延迟、中断处理时间、调度器延迟、任务调度时间以及任务本身执行时间。

为确保系统响应时间满足指定时限，不仅要求在绝大部分条件下（平均性能）满足，而且要求在最坏情况下满足，否则系统响应时间可能错过截止时间从而产生系统故障。比如说，上面的汽车安全保障系统，500us 是指最坏情况下的截止时间，因为平均情况下，系统也许能够在 250us 内提供服务，但是必须确保在最坏情况下能在 500us 内响应。最坏情况对操作系统而言，无非是系统有很高负载，其资源被充分消耗，这类负载包括大量的处理器负载、内存消耗和 I/O 操作。

实时操作系统的响应时间能否满足指定时限，其如何波动，可以人为地增加系统负载尽可能地模拟一个最坏情况并通过实时示波器进行测试与分析，因为实时示波器能够以很高分辨率对每个数据（比如某个时钟脉冲）进行采样、存储和分析进而确定特定数据和信号是否对系统稳定性造成影响。

通常，为了测试系统响应时间，除了要求实时示波器以外，还需要高频率的方波产生器，以便于模拟产生大量外部事件用于采样足够样本。

如果没有方波产生器，则可以通过其他方式模拟外部中断，比如，通过配置系统的 RTC 时钟产生周期性中断。如果没有实时示波器，数据采集可以利用处理器平台相关的高精度定时器（比如 X86 上的 tsc 时钟计数器，MIPS 平台上的 R4K C0 计数器）来实现。

对于采样到的数据，可以通过Worst-case Latency[13]，Jitter[1][20]和标准偏差[23]进行分析。

- Worst-case Latency

Worst-case Latency 指最坏情况下系统响应时间，即采样数据的最大值：

$$\text{Worst-case Latency} = \text{Max}(X_I)$$

- Jitter

Jitter即抖动，“抖动就是信号相对于其理想时间位置的偏离”[21][22]，可以用采样结果的最大值减去最小值来表示：

$$\text{Jitter} = \text{Max}(X_I) - \text{Min}(X_I)$$

最小值相当于一个理想值，也是后文将介绍的实时操作系统优化的目标。

- 标准偏差(Std.Dev.，Standard Deviation)和平均响应时间

平均响应时间即采样数据的算术平均值：

$$\overline{X} = \frac{1}{n}\sum_{i-1}^{n} X_i$$

Std.Dev.基于样本估算标准偏差，用于反映样本相对于平均值的离散程度。标准偏差越小，这些值偏离平均值就越少，反之亦然。标准偏差的大小可通过标准偏差与平均值的倍率关系来衡量。标准偏差公式：

$$\text{Std. Dev.} = \sqrt{\frac{1}{n-1}\sum_{i-1}^{n}(X_i - \overline{X})^2}$$

- Jitter 和标准偏差的比较

标准偏差除了反应系统的波动性以外，通常和算术平均值一起用于反应整个系统的 Qos（服务质量）。对于有些实时应用，除了满足必要的实时性能以外，还需要满足一定的 Qos，例如，如果两个实时系统的 Jitter 和标准偏差如下：

$$\text{Jitter} = \text{Max}(X_i) - \text{Min}(X_i) = 60 - 1$$

其中一个的平均值和标准偏差分别为 7 和 1，而另外一个的平均值和标准偏差分别为 32 和 20，这说明，第一个实时系统的大部分结果都集中在 7 周围，而第二个系统的结果围绕在 32 附近，而且很分散。可见，虽然 Jitter 一样，但是第一个系统的平均性能更好，而且更集中，有更好的 Qos，更适合那些即要求实时性，有要求一定的 Qos 的应用，比如 VoIP。

通常，实时操作系统需要的平均响应时间是 10~20us，在最坏情况下，响应时间也必须在几百us以内[13]。

对于软实时操作系统，由于容忍错过截止时间，因此这种偏差的容忍范围较宽松；而硬实时操作系统要求更苛刻的系统响应时间。当然，特定的应用可能有特定的需求，比如对于精密仪器的加工（如上面的数控机床），高效的燃气利用系统（如发动机）等可能不属于硬实时操作系统，错过截止时间并不会造成灾难性后果，但是如果服务响应时间更快，那么加工出来的仪器将更加精密，燃气的利用将更充分，产生更大的经济价值和科研价值等。

2.2.3 性能指标

通过对实时操作系统基本概念的分析，不难理解，实时操作系统要求提供“指定范围”内的“服务响应时间”，其性能指标包括以下几个部分：

1. 可确定的任务启动时间

对于任务启动时间，在综合考虑非周期性任务、周期性任务和多任务情况下，其包括如下几个部分。

- 中断延迟

- 中断处理时间
- 调度器延迟
- 任务调度时间
- 时钟精确性
- 同步、互斥和通信延迟

2. 可确定的任务执行时间

对于任务执行时间，有如下几部分：

- 任务本身的执行时间
- 系统库函数的执行时间
- 系统调用开销
- 处理器调度策略（禁止低优先级任务抢占高优先级任务）
- 内存申请开销
- 数据访问开销（访问内存和 Cache）
- 地址转换开销（虚拟地址到物理地址的相互转换）

对于这些性能指标，可以从以下几个方面进行评价：

- Worst case Latency（最根本的评价指标）
- Jitter（用于反映系统的稳定性/波动性）
- Best case Latency（用于反映系统潜在的优化目标）
- 平均性能和标准偏差（用于反映系统的 Qos）

关于上述术语的详细解释，请参考上一节。

为了达到较高的性能指标，需要实时操作系统从中断管理，任务调度，时钟管理，同步、互斥与通信机制，内存管理等各个方面满足一定需求，以确保整个系统响应时间的可确定性，从而尽量避免因为错过截止时间而产生系统故障，以免造成不必要的性能损失，更不能造成灾难性后果。

下面介绍实时操作系统的基本需求及为满足这些需求而制定的 POSIX 标准。

2.2.4 基本需求

为达到较高性能指标，综合上一节的分析以及资料[30][31]，实时操作系统应满足以下基本需求：

1. 中断管理

为了能够及时响应外部事件，实时系统必须提供底层中断事件管理机制。这种机制要求按照事件的紧迫程度响应中断，即实现中断优先级。

大部分实时操作系统（如实时抢占补丁）都试图把中断处理例程线程化（或进程化），让它们变成可调度的实体并赋予不同优先级从而可以根据外部事件的

紧迫程序进行响应。线程化中断例程前，它们具有相同优先级，并且由于它们不可调度，在任何中断发生后都能抢占其他任务，所以具有最高优先级，由于这些中断处理例程的执行时间不可预测，因此会严重影响实时任务的响应延迟。线程化中断例程后则避免了这类问题，由于此时有些通过关闭中断保护数据的操作不再需要，因此不仅会减少紧迫中断的响应延迟，而且会增加启动调度器的机会，减少调度器延迟。除此之外，由于线程化的中断处理例程可以被调度，在执行时可以主动释放处理器资源（睡眠），进行调度，也可以减少调度延迟，并且可以减少中断处理时间。

有部分实时操作系统（如 RTAI）采用双核技术，引入了实时硬件抽象层，所有的中断管理由该硬件抽象层接管，紧迫的外部事件会在该抽象层优先处理，而其他的事件则延迟到最后并交由 Linux 处理。这也变相地实现了中断优先级。

2. 任务调度

对于复杂的实时应用，要求支持多任务调度与抢占。不仅要求清楚地区分可调度与不可调度实体。调度实体具有上下文（控制块），能够明确地请求各类资源（包括 CPU、内存和 I/O），通过调度器调度，而调度器本身与非线程化的中断处理例程等都是不可调度的实体。

多任务调度要求不同调度实体（任务）间快速切换，并且要求支持基于优先级的抢占调度策略，比如更紧迫任务被赋予较高优先级，能够抢占较低优先级任务并保证不被更低优先级任务抢占。这类调度策略有先进先出(FIFO)，时间片轮转（RR）等。

为保障可以调度有截止时间要求的实时任务，还要求动态确定任务截止时间，并根据截止时间赋予不同优先级进而用于任务调度和各种资源的分配。这类调度策略有，最早截止时间优先（EDF）和最低松弛度优先（LLF）等。

实际应用还需要有足够数量的优先级以反映各种任务的不同紧迫程度。资料[30]提到，对于典型的实时应用来说，32 个优先级足够了，不过有的实时操作系统提供了更多，比如实时抢占补丁提供了 100 个，Vxworks提供了 256 个，而 RTAI则提供了 2^30 个，也有一些实时操作系统，比如Windows CE只提供了 8 个，无法满足有些实时应用的需求。

3. 时钟管理

对于通过自身驱动的周期性任务来说，需要准确地获取时间并有足够精度的定时器。通常，如果底层硬件支持（有足够高频率的时钟计数器），一般的实时操作系统都能够提供纳秒(ns)级的时钟精度。

4. 同步、互斥和通信机制

如果实时任务间存在对排他性资源（设备、内存区域、缓冲区等）的共享，那么实时操作系统必须提供互斥机制；如果实时任务的执行存在先后关系，那么还需要同步机制；而如果不同任务间存在数据交互，那么还需要提供通信机制。

对于一般的实时操作系统，常见的同步机制有实时信号，互斥机制有互斥锁、信号量等，而通信机制有共享内存、消息队列、管道、FIFO 和套接字等（有名管道）。

对于实时信号，为了能够优先处理一些紧急任务，要求支持信号优先级。

对于信号量，当高优先级与低优先级任务共享资源时，应提供优先级继承协议防止中间优先级任务抢占，从而避免不可预测的延迟。

对于任务间通信，要求支持非阻塞式通信、可确定的延迟时间以及同步通信。

5. 内存（包括 Cache、TLB）管理

内存管理包括两个方面，一个是内存分配和释放，另一个是内存访问。实时操作系统要保障内存分配（释放）时间和内存访问时间的可确定性。

对于内存分配（释放），除了静态内存分配（释放）算法外，还应该提供可确定性的动态内存分配（释放）策略，以满足一些复杂应用的需要。

对于内存访问，如果硬件支持，应该提供一些 API 以允许任务预留一定的 Cache、TLB 和内存，以避免 Cache 失效、TLB 失效和缺页带来的不确定性。

2.2.5 POSIX兼容性

为方便实时应用程序员编写可移植的程序，相关组织基于实时操作系统的基本需求，制定了相应的实时操作系统标准，规范了各种接口。

这类标准最典型的有IEEE POSIX 1003.1b[12]，大部分实时操作系统都遵循该标准，Linux实时抢占补丁也不例外。

第3章实时抢占补丁研究

在分析传统 Linux 实时性的不足后，该章比较了几种不同类型的 Linux 实时改造技术，调研了实时抢占补丁项目的研究现状与开发趋势，并总结了其优缺点。最后，本章就实时抢占补丁中采用的一些实时改造技术和实时调试与优化技术进行了深入分析。

3.1 实时抢占补丁概述

Linux自 1991 年由Linus创建以来，受益于其开放源码，市集式的开发模式[45]和GPL协议，得到了来自世界各地的自由软件爱好者、相关公司与学术团体的大力支持，其功能不断完善，应用范围不断拓展。

由于其良好的性能、多处理器支持、网络文件系统支持以及之后的包括xen、kvm、lguest、 qemu等在内的各类虚拟技术支持，使得它在服务器领域占据了一定市场；而由于其支持大量处理器平台与外部设备，也得到了个人用户的喜爱；并且由于其良好的可裁减性与移植性，在嵌入式领域也得到广泛应用。但随着其在音频 / 视频、VoIP Bluetooth、802.11、GSM、CDMA等领域的应用，它在实时方面的不足逐渐被人们发现并迫切需要得到改善，通过Motovisa的实时抢占补丁和Ingo的低延迟补丁[28]，Linux逐渐满足了软实时需求。与此同时，其他团体准备拓展其在雷达、医学议表、机器人、工业控制与飞行模拟等领域的应用，这些领域要求更高的实时性能。为满足这些领域的需求，FSLab的RTLinux试图分开实时和非实时部分（双核技术）来实现硬实时，其后的RTAI、Xenomai、 XtratuM等都采用了类似原理，而RED-Linux、Linux/RK、Kurt-Linux、UTIME、Timesys等则试图在调度策略、资源保留、高精度时钟、中断线程化等方面对原有Linux进行实时改造，后来的实时补丁项目继承了它们的优点，并提供一定的硬实时支持。

下面先分析传统 Linux 在实时方面的不足，对比几类流行的实时改造方法，进而介绍实时抢占补丁在实时改造方面的特点，以及其研发进展。

3.1.1 Linux在实时方面的不足

Linux 在实时方面的不足，体现在以下几个方面：

1. 早期 Linux 在任何时候，硬中断、软中断、tasklet、bh 具有最高优先级，它们能够中断任何其他实时任务，并且在中断处理时，中断可能被长时间地关闭，其他任务得不到调度，会造成很大延迟。
2. 早期 Linux 不支持抢占，其他任务只可能在任务本身因为等待事件被阻塞或者是时间片用完才让出处理器。不支持抢占意味着任务得到执行的机会少，因此响应延迟会不可预期。
3. 早期 Linux 在操作某些核心数据结构时，为保证数据结构的完整性，采取了关闭中断的措施。关闭中断时，系统外部事件无法得到响应，将极

大地增加任务响应延迟。同样地，为保护多处理器系统上的某些共享数据的完整性，采用了自旋锁，由于这些锁不能被抢占，其他进程可能会无限期地等待它们，使得实时任务的执行时间不可预测。

4. 早期时钟管理系统时钟分辨率很低，基于硬编码的HZ，即使HZ被设置为1000，时间分辨率只有1ms，对于一些任务周期少于1ms的实时任务来说，完全无法满足需求。而实际上，一些周期性实时任务要求在10us到几百us内得到调度[13][46]。如果把HZ设置得更大，比如100000，可以达到10us左右的时钟分辨率，但会使得时钟中断过于频繁，导致系统绝大部分时间浪费在正文切换上，严重影响系统性能（吞吐量）。
5. 早期调度器的时间复杂度是O(n)，调度器选择任务的开销与系统运行的任务数有关，对于一个任务数不确定的系统，任务调度时间将不可预期。
6. 早期Linux没有提供解决优先级反转的处理措施，当高优先级任务被共享了资源的低优先级任务阻塞时，中间优先级任务可能抢占低优先级任务，从而使得高优先级任务的运行时间变得不可预期。
7. 动态内存分配算法的时间复杂度是O(n)，具有不确定性。
8. Linux虽然提供了SCHED_FIFO和SCHED_RR两种实时调度策略，但是还没有提供基于截止时间优先的实时调度策略，比如EDF和LLF。

3.1.2 各种Linux实时改造方法

资料[32][46][47][48]等对各种Linux实时改造方法进行了整理与分析，从基本改造方法上可分为[13]：

1. 修改Linux内核本身，让其原生提供实时能力。这些实时改造方法有RED-Linux、Linux/RK、Kurt-Linux、UTIME、Timesys、Preempt-RT等。
2. 引入新的硬件抽象层，在该抽象层上实现实时操作系统的核心功能，而让Linux作为较低优先级任务运行在该抽象层上，以便利用Linux提供的各种丰富的子系统和设备驱动。这类实现包括L4Linux、RTLinux、RTAI、Xenomai、XtratuM、I-pipe等。

另外一个可行的分类方法是根据系统结构来划分：

整体式内核（monolithic kernel）	RED-Linux，Linux/RK，Kurt-Linux，UTIME，Timesys，实时抢占补丁等
双内核（dure-kernel）	RTLinux，RTAI，Xenomai，I-pipe
微内核（micro-kernel）	L4 + L4Linux + Partikle
超微内核（nano-kernel）	XtratuM + Linux + Partikle

绪论中已分析和比较了RTAI、Xenomai、XtratuM与实时抢占补丁各自的特点，更详细的分类和比较请参考资料[46]。

3.1.3 实时抢占补丁

实时抢占补丁继承与改进了所有整体式内核实时改造方法的优点，其核心思想是"Improve the kernel preemption itself; that is,make the parts of the code as preemptible as possible."[13]，它在原有的低延迟补丁和抢占补丁的基础上，引入了中断线程化、高精度时钟、临界区可抢占以及优先级继承等。

由于实时操作系统要求任务得到及时响应需要频繁地进行任务调度和正文切换会浪费大量处理器资源，而通用操作系统要求很高的系统资源利用率，因此两者间的矛盾不可能得到根本上统一[16]，所以在实现时，实时抢占补丁引入了不同内核配置选项以允许用户根据需求选择实时特性[19]，这些选项如下表所示：

抢占级别	应用场合	配置选项	简介
No Forced Preemption	服务器	PREEMPT_NONE	等同于没有使能抢占选项，主要适用于科学计算等服务器环境。
Voluntary Kernel Preemption	桌面（软实时）	PREEMPT_VOLUNTARY	使能了自愿抢占，但仍然失效抢占，它通过增加抢占点减少了抢占延迟，适用于一些需要较好的响应性的环境，如桌面系统，当然这种好的响应性是以牺牲一些吞吐率为代价的。
Preemptible Kernel	低延迟桌面（软实时）	PREEMPT_DESKTOP	既包含了自愿抢占，又使能了可抢占，因此有很好的响应延迟，实际上在一定程度上已经达到了软实时性。适用于桌面和一些嵌入式系统，但是吞吐率更低。
Complete Preemption	实时（硬实时）	PREEMPT_RT	使能了所有实时功能，因此完全满足软实时需求，适用于延迟要求为100微秒或稍低的系统。

如上表所示，随着抢占级别增加，系统实时能力不断提升，但资源利用率（吞吐量）明显下降，反之，随着抢占级别降低，实时能力不断下降，而资源利用率却不断提升。因此，吞吐量和系统响应时间几乎站在跷跷板两端，资料[16]更详细地对实时操作系统与通用操作系统在这两方面的不同进行了分析和比较。

目前，前三种配置都已经进入Linux官方，而PREEMPRT_RT还没有完全进入，不过包括CFS调度算法、高精度时钟、优先级继承的Mutex(PI-Mutex)在内的大部分工作都已经进入，资料[49]详细描述了实时抢占补丁当前的研发进展，而资料[15][50][51]则介绍了更早期的Linux实时改造历史。

在总结上述资料和分析 Linux 内核 git 仓库的日志后，得到了一份更详细的列表（仅以 X86 平台为例）：

内核版本	实时改造技术	作者
2.x	SMP Critical sections	

2.2	Low-Latency Patches	Ingo Molnar/ Audio Community
2.2 / 2.4	Preemption Points / Kernel Tuning	
2.4	Preemptible Kernel Patches	MontaVista : Nigel Gamble、Robert Love
2.5	Deterministic Scheduler: O(1) Scheduler	MontaVista -> Ingo Molnar
2.6.12?	Voluntary Preemption	Ingo Molnar
2.6.16	High-Resolution Timer	Thomas Gleixner
2.6.17	Lockdep	Ingo Molnar
2.6.17	PI-mutexes	Steven Rostedt
2.6.23	Deterministic Scheduler: CFS Scheduler	Ingo Molnar
2.6.25	Preemptive Read-Copy Update	Paul E. McKenney
2.6.26	Real time debugging framework: Ftrace	Ingo Molnar， Steven Rostedt
2.6.30	IRQ threads	Ingo Molnar
2.6.30	Real time tuning framework: Perf	Ingo Molnar
2.6.33	Raw spinlock Annotation	Thomas Gleixner
future release	Real Time Preemption	MontaVista->Ingo Molnar， Thomas Gleixner...
	SCHED_DEADLINE scheduling class[52][53]	Dario Faggioli and Michael Trimarchi
	Deterministic dynamically memory malloc/free: TLSF[54]	Miguel Masmano

如上表所示，随着时间推移，Linux 的实时改造工作一直在进行中，各种实时抢占补丁不断地被官方 Linux 接收（白色底纹），而在实时抢占补丁之外，也有其他团体在开展相关工作（灰色底纹），可以预计不久以后这些工作都可能进入到官方 Linux，进而不断改善 Linux 的实时能力。

实时抢占补丁中还没有进入官方的部分（绿色底纹）目前由 Thomas Gleixner 维护：

下载地址	http://www.kernel.org/pub/linux/kernel/projects/rt/
Git 开发分支	git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git rt/head
Git 发布分支	git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git

	rt/2.6.33

下面根据资料[46]及实时抢占补丁最新研发进展总结实时抢占补丁优缺点：

- 优点

实时抢占补丁能提供出色的调度延迟和中断响应延迟（几十 us），在某些配置下能够潜在地应用于一些硬实时服务中。内核中现有的各种外部设备驱动无需进行特别的修改就能应用于实时服务。支持 POSIX 的 API。该方法的大量实时改造技术不断地被官方内核接收，将得到长期的社区支持。实时任务跟非实时任务的开发区别不大，而且能够同时运行于同一个系统中，交互起来非常容易。

- 缺点

由于Linux系统本身庞大而复杂，目前做的测试都非常有限，更无法直接通过形式化手段认证其安全级别[27]，因此Linux的鲁棒性问题依然存在，不能使用在一些Safety Critical的环境中。由于它对整个 Linux进行实时改造，因此需要改动的代码较多，包括底层的中断处理，进程调度，所有关中断、禁止抢占，持有锁以及锁的实现，底层硬件的配置等各个方面，因此工作量较大。在引入实时支持后，无论是实时任务还是非实时任务，对系统的吞吐量都会造成一定影响。另外，由于采用整体式内核结构，实时任务和非实时任务没有隔离，因此，其中某个任务的出错可能导致整个系统崩溃，影响其他任务的执行，不过随着Xen、KVM、lguest等虚拟技术的出现，这方面的情况可能会得到改善。

下面对实时抢占补丁的一些实时改造技术进行分析。

3.2 实时抢占补丁分析

该节结合最新的稳定版(stable)实时抢占补丁 rt/2.6.33，对其部分关键实时改造技术进行分析。早期进入内核的低延迟补丁、自愿抢占补丁和抢占补丁是实时抢占补丁的基础，会先进行介绍，随后再介绍实时抢占补丁本身。

3.2.1 自愿抢占补丁/低延迟补丁

自愿抢占补丁（Voluntary Preemption）由早期的低延迟补丁（low-latency patches）演进而来[19]。

低延迟补丁的核心思想是自愿抢占，即在一些执行时间较长的代码块中明确地引入调度点，主动放弃处理器资源。实现方法是先找出内核中耗时较大的操作，并在操作超出某一时限时，检查是否有其他更紧迫任务需要执行（need_resched 是否为 1），进而安全地放弃处理器资源，这里的安全意味着资源的放弃应该避免在持有锁的情况下发生，因此如果持有锁，那么需要先释放锁，回来之后再持有锁，即所谓的锁分解[28]。

低延迟抢占提供的接口如下：

include/linux/sched.h:
#define cond_resched() ({ \

```
    __might_sleep(__FILE__,  __LINE__,  0);   \
    _cond_resched();                          \
})
```

kernel/sched.c:

```
int __sched _cond_resched(void)
{
  if (should_resched()) {
     __cond_resched();
     return 1;
  }
  return 0;
}
```

一个很简单的例子是:

drivers/char/tty_io.c:

```
do_tty_write {
    ...
   for (;;) {
     ...
      cond_resched();
   }
}
```

上述例子中，在一个大循环里头，通过 cond_resched()明确地引入了调度点。而关于锁分解的例子有:

drivers/md/raid5.c:

```
raid5d () {
    ...
    spin_lock_irq(&conf->device_lock);
     while (1) {
              ...
               spin_unlock_irq(&conf->device_lock);
                ...
               cond_resched();
```

```
            spin_lock_irq(&conf->device_lock);
        }
        ...
        spin_unlock_irq(&conf->device_lock);
}
```

如果不考虑调度点的引入，上述整个循环体可以直接用前后两个锁操作保证数据的完整性，但是引入调度点后，需要避免在持有锁时发生调度，避免锁被无限期地持有，避免无限期地阻塞其他共享了资源的任务。

在内核中潜在地存在着大量需要明确引入调度点的地方，但是如何找出这些执行时间较长的热点区域呢，早前是通过Andrew Morton's rtc-debug patch来实现的[28]，这个patch通过修改Real Time Clock驱动找出调度延迟大于某个阈值的区域。找到这样的区域后把当前执行路径打印到系统日志文件中。通过检查这些日志文件就可以找出哪些函数导致了较长延迟并分析原因，进而插入调度点以减少延迟。不过，对于最新内核，由于引入了Ftrace内核调试框架，可以直接通过Ftrace来跟踪。

经过统计，在内核 rt/2.6.33 中，引入了 cond_resched()的位置多达 420 处。

需要注意的是，明确地引入抢占点可能会影响系统的性能（吞吐量），比如在网络数据传输过程中，如果刻意在传输一定字节后引入调度点，那么整个网络传输性能会明显下降，所以在引入抢占点时需要对系统响应延迟和吞吐量有一个折衷地考虑。另外，在编写新驱动时也需要在权衡吞吐量和延迟之后通过引入调度点以避免执行时间过长的代码块。

3.2.2 抢占补丁

与低延迟补丁的自愿抢占思想不同，抢占补丁的思想是强制抢占，即让调度器有更多机会启动，从而减少调度器延迟[28]。它通过修改自旋锁和中断返回代码以便安全地抢占当前进程，即当有更紧迫任务发生时(通过need_resched判断是否需要重新调度)，调度器有机会启动。

早期 Linux 内核假设在通过系统调用或中断进入内核后，在内核能够安全地进行重新调度前，当前的进程不能被修改。这种假设允许内核在不请求互斥锁（如自旋锁）保护的情况下修改内核的一些数据结构，随着时间推移，这类无须保护就可修改内核数据结构的代码不断减少，正是基于这一点，抢占内核假设：如果执行的代码不在中断处理例程中或者没有自旋锁保护，那么当前进程（或线程）发生正文切换是安全的。抢占补丁给每个数据结构增加了一个变量 preempt_count 用于维护每个线程状态，该变量可以通过如下几个宏来修改：

preempt_disable()， preempt_enable()和 preempt_enable_no_resched()。

宏 preempt_disable()增加该变量而 preempt_enable*()减少该变量，宏 preempt_enable()通过检查 need_resched 确认是否要重新调度，如果是的话，就把 preempt_count 修改为 0 并调用 preempt_schedule()。该函数把 preempt_count 加上 0x4000000 以标记当前发生了抢占调度，随后执行 schedule()进行调度，调

度完后，把 preempt_count 减去 0x4000000 以表示退出了抢占调度。相应地，也需要修改调度器，检查 0x4000000 以判断当前是否正在发生抢占调度，如果是则避免再次调度（对于同时采用低延迟补丁和抢占补丁的情况，在_cond_resched()中调用的 should_resched()也会判断该变量）。相应地，宏 spin_lock()和 spin_unlock()调用 preempt_disable()和 preempt_enable()以避免临界区被抢占，而宏 spin_trylock()则先通过 preempt_disable()禁止抢占，如果得不到锁则调用 preempt_enable()以允许抢占。

另外，抢占补丁也修改了中断返回代码，当不在临界区时，会跟 preempt_enable()一样，先检查 need_resched 确认是否要重新调度，如果是则调用 preempt_schedule()。

从中断返回的强制抢占，一般都在特定架构下的 entry*.S 中实现。例如，32 位 X86 平台的实现如下：

arch/x86/kernel/entry_32.S：

```
#ifdef CONFIG_PREEMPT
#define preempt_stop(clobbers) DISABLE_INTERRUPTS(clobbers); TRACE_IRQS_OFF
#else
#define preempt_stop(clobbers)
#define resume_kernel                restore_all
#endif
...
ret_from_exception:
        preempt_stop(CLBR_ANY)
ret_from_intr:
        GET_THREAD_INFO(%ebp)
check_userspace:
        movl PT_EFLAGS(%esp), %eax      # mix EFLAGS and CS
        movb PT_CS(%esp), %al
        andl $(X86_EFLAGS_VM | SEGMENT_RPL_MASK), %eax
        cmpl $USER_RPL, %eax
        jb resume_kernel                # not returning to v8086 or userspace.
...
#ifdef CONFIG_PREEMPT
ENTRY(resume_kernel)
        DISABLE_INTERRUPTS(CLBR_ANY)
```

```
        cmpl $0,TI_preempt_count(%ebp)    # non-zero preempt_count ?
        jnz restore_all
need_resched:
        movl TI_flags(%ebp), %ecx         # need_resched set ?
        testb $_TIF_NEED_RESCHED, %cl
        jz restore_all
        testl $X86_EFLAGS_IF,PT_EFLAGS(%esp) # interrupts off (exception path) ?
        jz restore_all
        call preempt_schedule_irq
        jmp need_resched
END(resume_kernel)
#endif
```

从中断返回时会明确地检查是否有更紧迫任务（即上面提到的通过检查need_resched以确定是否需要重新调度），如果有则在通过restore_all恢复中断正文前调用preempt_schedule_irq()进行抢占。下面是在配置了PREEMPT（强制抢占）后，X86 平台上的中断软件响应过程[15]，图中彩色部分标记了强制抢占（PREEMPT）与图 2-4描述的非强制抢占（PREEMPT_NONE）的不同。

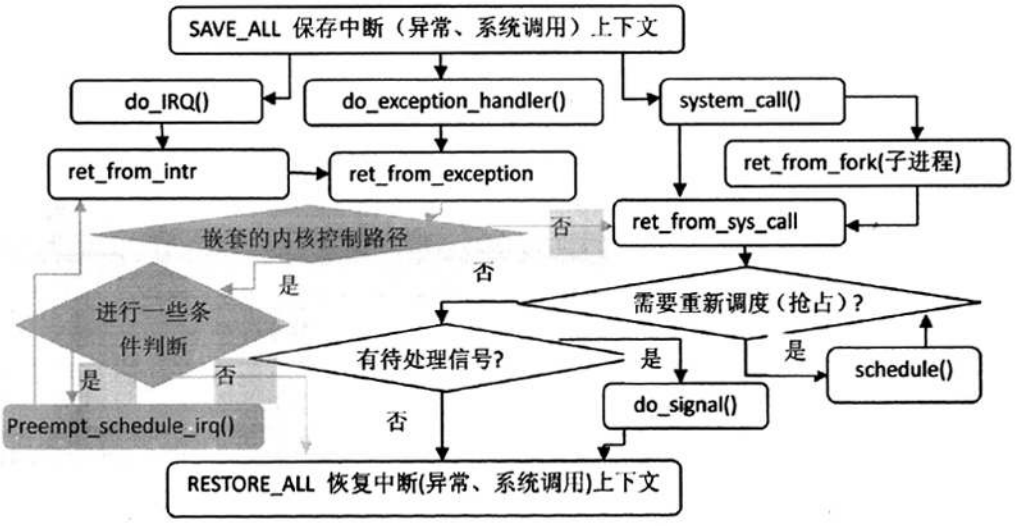

图3-1 允许抢占时 X86 平台下中断的软件响应过程

而从 spin_unlock()返回时，会通过 preempt_enable()明确地允许抢占，同样检查是否有更紧迫任务发生，如果是就发生抢占。

include/linux/preempt.h:

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

抢占补丁减少了从任务被唤醒到检查 need_resched 标记直到调度器被启动的这段时间。因此，在释放自旋锁和中断发生时，调度器就有机会启动。

3.2.3 实时抢占补丁

上面两种实时改造方法都极大地提高了响应能力，但单独无法保障最大响应延迟，资料[28]试图把两种补丁都追加到内核中，获得了较好的实时性能，"The next night I ran the combined preempt+low-latency kernel for 15.5 hours and got a maximum latency of 1.5ms."该结果满足一些软实时需求，但对于要求更高的实时任务，上述两个补丁还是无法满足需求。于是，包括Timesys、UTIME、Kurt-Linux与RED-Linux等试图在其他方面取得突破，相应的改进包括把中断处理例程线程化，提高时钟系统的分辨率、改善调度策略、让临界区可抢占等，下面将深入分析这些实时改造技术。

3.2.3.1 中断线程化

抢占补丁允许在中断返回时发生调度，但从中断发生到中断返回这段时间，任务无法得到调度。从中断发生到中断返回期间，除了硬件层面的寄存器保护工作外，还包括中断处理例程。硬件层面的工作对于具体硬件平台基本上是确定的，但是中断处理例程的执行时间不可预测。只要有中断发生，任何其他任务都会被中断，因此中断具有最高优先级，如果外部 I/O 事件频繁发生，那么其他实时任务就很难得到调度，另外，在执行中断处理例程时，由于中断处理例程不能被调度，而用于保存中断上下文的栈空间有限，为了防止级联中断，期间会关闭中断，这意味着即使有更紧迫外部事件发生，也无法得到处理。

为了解决上述问题，实时抢占补丁引入了中断处理例程的线程化，让中断例程变成可调度实体，不仅可以根据任务的实时要求给中断线程安排优先级，而且减少了大量关闭中断的区域，让调度器能够更早地启动。

中断线程化包括两部分，即硬中断和软中断的线程化。

软中断的线程化在 spawn_ksoftirqd()中完成，该函数在内核初始化时执行，它通过 kthread_create()调用 run_ksoftirqd()创建相应的软中断线程，设置各个软中断线程的优先级，软中断线程都采用实时调度策略(SCHED_FIFO)，其优先级是用户线程优先级的一半减去 1，比硬中断线程优先级低。线程化软中断后，所有软中断的处理都转到软中断线程中去。处理完硬中断后，在 irq_exit()中调用 do_softirqd()通过 wakeup_softirqd()唤醒相应的中断处理线程。

硬中断的线程化在__setup_irq()中完成，通过 kthread_create()调用 irq_thread()创建相应的硬中断处理线程，硬中断线程的调度策略也是 SCHED_FIFO，优先级为用户线程优先级一半，比软中断线程的优先级要高。__setup_irq()被 setup_irq()，request_irq()或者 request_threaded_irq()调用，分别用于静态地和动态地注册中断。中断发生后，硬中断的处理在 do_IRQ()中完成，最后通过 handle_IRQ_event()唤醒相应的中断线程来处理。

需要注意的是，有些中断不能被线程化，比如时钟中断用于驱动整个系统，需要有最高优先级，不能被延迟处理。而一些执行时间短的中断处理例程，例如所有处理例程为no_action()的中断，由于处理例程执行时间短，对系统响应延迟造成的影响可以忽略，线程化后反而会带来较大调度开销。除此之外，对于诸如Bus Error等中断，表示系统出错不能正常工作，应该禁止其他任务运行，因此需要及时处理，也不能线程化。对于不能线程化的中断，需要在中断的irqaction结构体的flags成员中加上IRQF_NODELAY标记，特别地，对于时钟中断，因为IRQF_TIMER标记包含了IRQF_NODELAY标记，只需要加IRQF_TIMER标记[5]。

3.2.3.2 高精度时钟

时钟管理系统主要有两个功能，即获取时间与提供软时钟以便在指定时限内启动任务。

早期 Linux 系统基于 Jiffies 通过周期性时钟滴答计时，当系统时钟频率 HZ 被设置为 1000 时，Jiffies 能够分辨到 1ms。而系统软时钟也基于 Jiffies，因此，只能提供 1ms 粒度的软时钟，例如可以设置任务在下 1ms 唤醒。在这种时钟系统下，为获取高精度时间，需要访问硬件平台的特定时钟计数器，比如 X86 平台的 TSC 寄存器，而为了调度周期性更短的任务，则需要把时钟频率设置得更短。因此，传统时钟管理系统存在两个明显的问题：

- 在获取时间方面，为了获取高精度时间，依赖特定的平台。
- 为调度短周期任务，需要设置时钟频率为更小值，意味着时钟中断更加频繁，可能造成大量时钟中断处理和资源浪费。

为解决上述问题，Thomas Gleixner等人设计了全新的时钟管理系统[55][56]，并抽象出了两个子系统，一个是clocksource，用于获取系统时间，另外一个是clockevent，用于按需设置下一个定时事件。前者为用户提供统一的时钟获取接口，并返回以ns为单位的时间，而后者不再单纯地采用周期性(periodic)时钟中断，而可以根据任务需要动态地设置下一次时钟中断发出的时间(oneshot)，这个时间也是以ns为单位。

对于特定硬件平台，只需要实现底层相关接口，注册相应的 clocksource 和 clockevent 结构体。而对于用户，只要硬件提供高分辨率(ns 级)的时钟计数器，就可以获取到 ns 级的时间并指定 ns 级的睡眠时间。

最新的时钟管理系统大体结构如下[55]：

[5] 2.6.30 之前的 Linux 并没有引入 IRQF_TIMER 标记，因此，还是需要使用 IRQF_NODELAY 标记

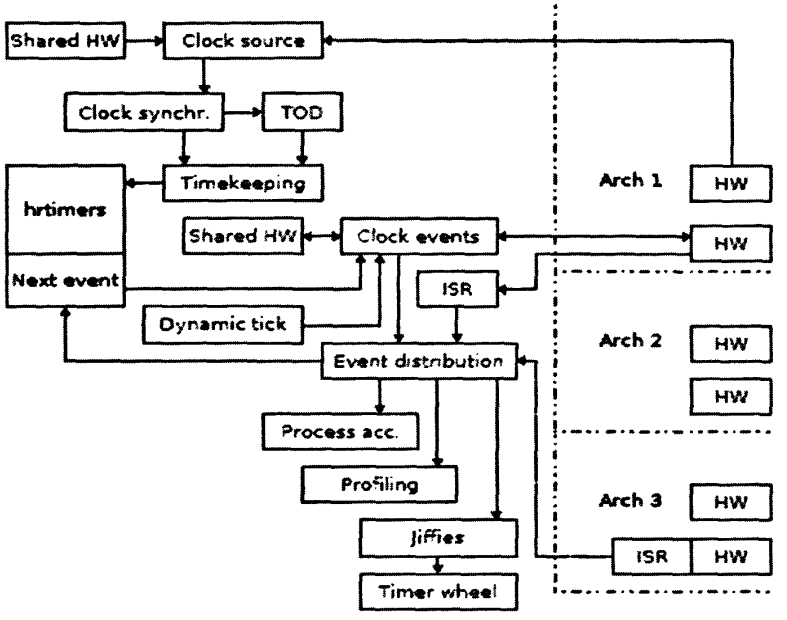

图3-2　高精度时钟管理系统结构

如上图所示，新时钟管理系统不仅能够提供旧时钟管理系统的功能，为上层应用提供必要的接口，而且能够减少平台相关代码的冗余，并能够基于它实现Tickless节能[56]。

对于特定平台的实现，以 MIPS 的 r4k 时钟为例进行简要介绍。

- 注册 clocksource，实现底层时钟获取函数

arch/mips/kernel/csrc-r4k.c:

```
static cycle_t c0_hpt_read(struct clocksource *cs)
{
        return read_c0_count();
}
static struct clocksource clocksource_mips = {
        .name           = "MIPS",
        .read           = c0_hpt_read,
        ...
};
int __init init_r4k_clocksource(void)
{
        ...
```

```
        clocksource_register(&clocksource_mips);
        return 0;
}
```

- 注册 clockevent，实现硬件定时器的设置

```
arch/mips/kernel/cevt-r4k.c:
static int mips_next_event(unsigned long delta,  struct clock_event_device *evt)
{
    /* 硬件计数器的设置 */
}
int __cpuinit r4k_clockevent_init(void)
{
    ...
    struct clock_event_device *cd;
    cd->name                    = "MIPS";
    cd->features                = CLOCK_EVT_FEAT_ONESHOT;
    cd->set_next_event          = mips_next_event;
    clockevents_register_device(cd);
    ...
}
```

对于用户空间的接口，则主要有 clock_gettime()用于获取以 ns 为单位的时间，而 clock_nanosleep()用于设置 ns 级的软时钟。如果仅需要获取 us 级的时间，可以调用 gettimeofday()，相应地，可以通过 usleep()实现 us 级的睡眠，但是在实时应用中建议采用 clock_gettime()和 clock_nanoslep()。

3.2.3.3 实时调度算法

对于实时调度，不仅要求调度的时间复杂度（选择任务的时间）可确定，而且要求实时任务得到优先处理。

早期Linux系统采用了O(n)的调度算法，这意味着选择任务的开销与系统中运行的任务数有关，是不确定的，后来，Ingo Molnar引入了O(1)调度算法，使得任务的选择时间与系统的任务数无关，从而改善了调度器的实时性能。O(1)调度器[57]具有很好的伸缩性，它结合交互性的指标和大量的启发式方法来决定一个任务是I/O紧密型（I/O-bound）还是处理器紧密型（Processor-bound）。但是，由于O(1)调度算法需要管理大量用于计算启发式的代码，维护起来比较困难，所以后来Ingo Molnar引入了新的CFS调度器，该算法时间复杂度是O(log n)，随着任务数增加，任务选择开销变化较小，当任务数较少时，依然可以认为是确定的。

为了优先处理实时任务，Linux系统引入了不同的调度策略[58]。前面的CFS（SCHED_NORMAL或早期的SCHED_OTHER）用于调度普通任务，而另外两种调度策略SCHED_FIFO和SCHED_RR用于调度实时任务。所有采用实时调度策略的任务会被优先调度。

主要的实时调度策略是 SCHED_FIFO，它实现了先进先出的调度算法，当一个 SCHED_FIFO 的任务开始运行，它会一直运行到该任务放弃处理器或者被更高优先级的实时任务阻塞或抢占。它没有时间片的概念，所有其他低优先级任务只能等到它释放处理器资源时才有机会得到调度，两个相同优先级的 SCHED_FIFO 任务不会抢占对方。SCHED_RR 与 SCHED_FIFO 类似，这些任务除了基于优先级分配时间片以外，会一直运行到用光所分配的时间片。

对于默认配置的内核，实时任务的优先级从 0 到 (MAX_RT_PRIO-1)，其中 MAX_RT_PRIO 为 100。非实时任务优先级从 MAX_RT_PRIO 到(MAX_RT_PRIO + 40)，这个优先级被映射到用户空间的 nice，范围为-20 到 19。需要注意的是，从数字上看，非实时优先级比实时优先级要高，但是由于调度器先根据调度策略选择要调度的任务，采用实时调度策略的任务优先被调度，所以会优先执行实时任务，在实时任务组中再根据优先级进行调度。

实时调度器必须确保系统范围内的任务严格按照优先级顺序执行，这种系统范围内意味着它不仅要考虑单一处理器上的任务运行队列，还要确保所有给定的处理器上的运行队列中的实时任务都遵循优先级的调度策略。关于这一部分，请参考资料[58]，至于优先级的反转问题，会在随后讨论。

现有的实时调度策略虽然满足一般要求，但是无法调度那些有明确截止时间要求的任务。这个需求在本文第二章以及资料[59]已经明确地提出，而Dario Faggioli和Michael Trimarchi实现了最早截止时间优先的调度策略，叫SCHED_DEADLINE，该调度策略相关论文如资料[52]，其源码维护在资料[53]。

由于有截止时间要求的任务较复杂，需要指定任务周期、运行时间和截止时间，而原有系统调用只能指定调度策略和优先级，因此无法满足要求，所以需要内核为用户提供新的系统调用。

该调度策略跟其他实时调度策略一样，采用该调度策略的任务会比普通任务优先执行。早期版本的 SCHED_DEADLINE 的优先级比 SCHED_FIFO 和 SCHED_RR 要低，但是最新的版本则比 SCHED_FIFO 和 SCHED_RR 要高，这意味着，如果任务采用 SCHED_DEADLINE 调度策略，它会比其他任何任务都优先执行。

论文[52]和资料[53]显示，该调度策略能够调度具有特定运行时间、周期和截止时间的任务。不过由于相关API还没有稳定下来，支持的平台仅有X86 和ARM，底层实现还存在一些问题，所以该调度策略还在不断改进和完善，但可以预见，不久后该调度策略可能被官方Linux接收。

下一章将讨论该调度策略在MIPS平台的移植，并分析当前SCHED_DEADLINE调度策略[6]存在的bug和解决办法。

[6] 从事该研究时的最新 SCHED_DEADLINE 基于 rt/2.6.31，而最新版本基于 rt/2.6.33，本文将仅介绍前者

3.2.3.4 临界区可抢占

抢占补丁中，退出临界区或从中断返回时可抢占，但临界区被禁止抢占。为保障数据结构的完整性，系统中存在大量的临界区，这些临界区包括自旋锁、大内核锁和RCU保护的区域，很大程度上影响了系统的响应能力。因此，实时抢占补丁采取了措施以确保这些锁可以抢占。对于自旋锁和大内核锁，基本解决办法是把它们全部转化为优先级继承的Mutex，转换后，这些Mutex保护的临界区可以被没有共享资源的其他高优先级任务抢占，对于高优先级任务，如果发现低优先级任务持有了锁，会把自己挂到该锁的等待队列中，等到该锁被释放后就会得到执行（存在优先级反转，详细讨论见后文）。该等待队列把锁的请求者按照优先级排序，所以对于竞争资源的高优先级任务在资源可用时会优先获得。

需要注意的是，对于有些内核数据结构、寄存器、非线程化的中断处理例程中存在的临界区，不能被抢占，因此相关锁不能转换为Mutex。比如非线程化的中断处理例程由于不能被调度，其临界区不能睡眠，如果转换成Mutex，那么整个系统将由于得不到执行而僵死；而对于某些硬件寄存器，如果把保护它们的锁转换为Mutex，相关的操作可能被中断（或抢占操作）破坏[50]，而某些关键的内核数据结构，比如die_lock，因为内核出现了严重错误而使用该锁，应该禁止其他任务执行，所以不能抢占。

除此之外，为检查自锁与非顺序锁引起的死锁问题，实时抢占补丁引入了死锁检测机制[19]，即lockdep，在遇到这类死锁时，会打印当前执行路径并panic，从而方便内核开发人员调试并解决这类死锁问题。

3.2.3.5 优先级继承

为阻止不可预期的优先级反转问题，实时抢占补丁实现了优先级继承协议。关于该协议及其在内核中的实现，请参考资料[17][60][61]。

3.2.3.6 其他

- 用户空间的实时支持

关于用户空间的实时支持请参考POSIX的标准[12]及相应的开发手册[25]。关于Glibc中的优先级继承futex(PI-futexes)以及相应的内核支持，请参考资料[80][81][82][83]。

- 实时虚拟化技术、容错技术

资料[2]介绍了XtratuM作为一款实时Hypervisor（虚拟机），能够有效隔离实时系统和非实时系统，防止错误在不同系统上扩散，除此之外，该资料在文末还提到了一种通过实时操作系统冗余实现容错的技术。对于实时抢占补丁，虽然Linux的现有相关虚拟化技术没有XtratuM这么轻量级，但也在研究当中，资料[79]提到，使用实时抢占补丁后，Qemu/KVM虚拟机已满足一定的软实时要求。

- 内存分配算法

目前内核采用了非实时的内存分配算法[42][43][44]，因此，在没有可用的实时内存分配算法时，实时开发中应禁用动态内存分配：

"Developers of real-time systems avoid the use of dynamic memory management because they fear that the worst-case execution time of dynamic memory allocation routines is not bounded or is bounded with a too important bound" (I. Puaut， 2002)

不过，目前有实时内存分配算法可用，该算法是TLSF[54]，基于TLSF，有人改进它并实现了xvMalloc[62]。不过，相关算法并没有进入官方Linux，因此，在最新内核中可能不能直接使用。

3.3 实时抢占补丁的调试与优化支持

实时系统开发过程的一个重要问题是找出影响实时性能瓶颈的位置并找到合适的解决方案，进而提高实时性能，这就是实时性能的调试和优化，它涉及到两个方面，首先是可确定性的调试，即消除系统中的不确定性（引起各种延迟的原因），其次是在可确定性的基础上，减少响应延迟。

3.3.1 实时调试和优化工具 1：Ftrace

至于可确定性的调试，Ingo Molnar和William Lee Irwin III在实时抢占补丁中创建了早期的latency tracer工具[65]用于找出内核中各种引起延迟（关闭中断、关闭抢占）的区域，辅助分析进而减少各类延迟。该工具的一个重要特性是添加了函数追踪，它在内核中基于gcc的-pg参数实现了一个mcount函数[64]用于记录内核中所有被调用到的函数，并显示在中断关闭和抢占禁止时，哪些函数被执行到，以及这些函数的时间开销。

Steven Rostedt早期维护了自己的LOGDEV工具[63]，用于跟踪终端或网络流量抑或是那些可能在 1s内（或更少时间内）执行成百上千次的代码，后面他发现，该工具可以充分利用latency tracer中的Function tracer，而且发现Function Tracer对普通的调试也很有作用，于是改造了早期的latency tracer形成了现在的Ftrace[67][68]，成为内核追踪的框架，这意味着用户可以在这个基础上添加更多的跟踪器（Tracer）。

Ftrace后来集成了event tracer[66]，用于跟踪系统事件，比如 timer，系统调用，中断等。

关于Ftrace的用法、原理和实现细节请参考资料[67][68]，下一章将详细讨论Ftrace在特定平台的移植。

3.3.2 实时调试和优化工具 2：Perf

早期 Linux 提供了一个叫 Oprofile 的子系统，通过该系统以及相应的驱动就可以利用硬件提供的硬件性能计数器辅助内核与应用程序的性能分析与优化。该子系统需要一个程序 Oprofile 做为前端来跟底层驱动进行交互，收集并分析数据。

虽然该工具可以用于分析硬件性能计数器提供的 CPU 周期数、分支预测失败、Cache 失效等各种硬件信息，但是不能用于辅助一些软件计数器的性能分析，比如缺页、正文切换次数、地址对齐错误等。

为了把各种硬件性能计数器和软件计数器统一起来，更好地分析整个系统性能的瓶颈，Ingo Molnar 等人试图抽象出一个性能分析框架，进而对硬件性能、软件性能甚至是 Ftrace 中的各种 event tracer 统计到的其他事件进行分析。

Perf 的具体实现包括两个部分，一部分是内核支持，它实现了底层各种计数器的开启、关闭与数据采样并为用户空间提供系统调用 sys_perf_event_open()；另一部分则是用户空间工具 Perf，该工具通过系统调用 sys_perf_event_open()设置各种计数器，然后通过 prctl()控制计数器的开启和关闭，之后通过 read()访问性能计数器的统计数据。

对于硬件性能计数器，需要硬件平台实现平台特定的底层驱动，这些驱动与早期的 Oprofile 驱动接口类似，具体移植时可以予以参考。关于该工具的实现请参考内核源代码下 tools/perf/design.txt，tools/perf/Documentationtools/perf 以及 arch/*/kernel/perf_event*.c。tools/perf 是 Perf 提供给用户空间的工具。

第4章实时抢占补丁移植

刚进行该研究时的稳定版(stable)实时抢占补丁(2.6.26.8-rt16)支持包括X86、PowerPC、ARM与MIPS[69]在内的诸多平台，但还有很多平台没有提供完整支持，例如，虽然龙芯跟MIPS基本兼容，但是该补丁不能直接在龙芯[70]平台上使用。而当时最新的开发（develop）版本：2.6.29-rc6-rtX仅支持X86 与PowerPC等几个平台，因此，最初的工作包括把 2.6.26.8-rt16 和 2.6.29-rc6-rtX移植到MIPS（龙芯）上，作者的论文[72]已经对这部分工作进行了介绍，而该章主要以 64 位龙芯平台为例介绍最新的实时抢占补丁 2.6.33-rt11 的移植情况，由于相关移植成果已被该项目官方社区接收[73]，所以把移植的过程记录下来希望能为其他平台的内核开发人员在移植和开发实时抢占补丁时提供一定的参考。

Ftrace作为源自实时抢占补丁项目的一款实时调试工具，能够有效辅助内核开发人员定位实时系统的瓶颈[74]。因此，在移植实时抢占补丁前，最好先移植Ftrace，因为移植完Ftrace后，就能利用它辅助实时抢占补丁的调试和优化。

下面先介绍 Ftrace 的移植，接着再介绍实时抢占补丁的移植。

4.1 Ftrace的移植

移植Ftrace前需要了解Ftrace的基本工作原理、用法及平台相关性。资料[67][75][76]介绍了它的基本原理和用法，而资料[68]以及kernel/trace/*，arch/*/kernel/{ftrace.c，entry*.S}等源码为具体移植工作提供了第一手资料。

4.1.1 确定待移植内容

在内核 2.6.33 源码目录下，通过"make menuconfig ARCH=x86"进入菜单：

```
Kernel hacking   --->
    [*] Tracers   --->
```

可以看到多达 16 个以上的Tracer，如果目标平台都不支持，那么移植工作量会很大。但是，可以优先移植那些跟实时调试与优化关系较大的Tracers。从资料[74]可以获得这样一份列表：

1. Lantecy Tracers

- Irqsoff Latency Tracer

关闭中断时，系统无法响应来自外部设备的事件，甚至进程之间都无法通信，这意味着系统响应延迟。Irqsoff Tracer 能跟踪并记录内核中关闭了中断的函数，对于其中关闭中断时间最长的，Irqsoff 将在日志文件的第一行标示出来，从而方便开发人员迅速定位造成响应延迟的原因。

该 Tracer 的大体实现是，如果配置了 CONFIG_IRQSOFF_TRACER，会在所有关

闭和开启中断的位置插入跟踪点，例如，local_irq_disable()和 local_irq_enable()：

```
#define local_irq_enable() \
        do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
#define local_irq_disable() \
        do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
```

同样地，对于那些明确地关闭中断的 Irq 操作，例如 local_irq_save()和 local_irq_restore()等也会插入跟踪点。

通过 trace_hardirqs_on()和 trace_hardirqs_off()记录中断被确切地关闭的时间，如果此时开启了 Function Tracer，期间被调用的内核函数也会记录。系统最终反应的是关闭中断事件最长的部分，从而辅助开发人员对这部分进行优化。

如果没有配置 CONFIG_IRQSOFF_TRACER，那些跟踪点就不会插入到相关函数中，从而避免给系统带来额外开销。这个通过 C 语言的宏开关实现：

```
#ifdef CONFIG_IRQSOFF_TRACER
    /* 同上 */
#else
#define local_irq_enable() raw_local_irq_enable()
#define local_irq_disable() raw_local_irq_disable()
#endif
```

关于关闭中断的时间统计，在 trace_hardirqs_on()和 trace_hardirqs_off()中最终会分别调用到 stop_critical_timing()和 start_critical_timing()。

- Preemptoff Lantecy Tracer

开启中断但禁止抢占会允许外部设备通知 CPU 有事件发生，但是任务必须等到内核开启抢占后才能调度，因此会造成调度延迟，增加系统响应时间。该 Tracer 跟踪并记录禁止抢占的函数，从而方便开发人员迅速定位造成调度延迟的原因。

类似地，Preemptoff Tracer 是在配置了 CONFIG_PREEMPT_TRACER 时，在禁止和允许抢占的函数中插入跟踪点，例如：

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

```
        barrier(); \
        dec_preempt_count(); \
} while (0)
void __kprobes sub_preempt_count(int val) {
    ...
    if (preempt_count() == val)
        trace_preempt_on(CALLER_ADDR0,  get_parent_ip(CALLER_ADDR1));
    ...
}
void __kprobes add_preempt_count(int val) {
    ...
    if (preempt_count() == val)
        trace_preempt_off(CALLER_ADDR0, get_parent_ip(CALLER_ADDR1));
    ...
}
```

类似地，trace_preempt_on()和 trace_preempt_off()通过 stop_critical_timing()和 start_critical_timing()来统计抢占被禁止的时间。

- Preemptirqoff Latency Tracer

当中断和抢占同时或者有一个被禁止时，任务将不能被调度。该 Tracer 跟踪并记录中断或抢占中至少有一个被禁止的函数，以及禁止时间最长的，从而确定任务无法调度的原因。

这里会记录上述两种 Tracer：Irqoff tracer 和 Preemptoff tracer 的结果。

- Wakeup Latency Tracer

Wakeup 和 Wakeup_rt Tracer 跟踪进程调度延迟，即高优先级任务从进入 ready 状态到获得 CPU 的延迟。前者考虑所有任务，而后者只考虑实时任务，两个跟踪器都只跟踪当前系统中最高优先级任务。

Wakeup Tracer 在任务被 Wakeup 时插入跟踪点。在任务被 Wakeup 为 ready 状态即 TASK_RUNNING 时，wakeup 的时间被记录下来：

```
static int try_to_wake_up(struct task_struct *p,  unsigned int state,
                          int wake_flags,  int mutex) {
    ...
    trace_sched_wakeup(rq,  p,  success);
    ...
```

```
}
```

任务获得 CPU 时，也插入相应的跟踪点，实现时该跟踪点被插入在发生正文切换时（即被调度器选为下一个即将执行的任务），而不是任务执行时：

```
static inline void context_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next)
{
    ...
    trace_sched_switch(rq, prev, next);
    ...
}
```

所以这里跟踪到的 Wakeup 时间并不包括正文切换开销，只包括从任务被唤醒到任务被选中后即将发生正文切换时的时间。该时间通过如下计算得到：

```
static void probe_wakeup(struct rq *rq, struct task_struct *p, int success) {
    ...
    data = wakeup_trace->data[wakeup_cpu];
    data->preempt_timestamp = ftrace_now(cpu);
    tracing_sched_wakeup_trace(wakeup_trace, p, current, flags, pc);
    ...
}
static void notrace probe_wakeup_sched_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next) {
    ...
        tracing_sched_switch_trace(wakeup_trace, prev, next, flags, pc);
        T0 = data->preempt_timestamp;
        T1 = ftrace_now(cpu);
    ...
}
```

上述两个函数通过注册，分别在 trace_sched_wakeup()和 trace_sched_switch()函数中调用。

2. 其他 Tracers

- Function Tracer

Function Tracer 用于追踪内核中的所有函数并记录各个函数的起止执行时间，能够有效辅助分析内核执行过程并进行性能优化。Function Tracer 能为 Latency Tracer 提供更多信息，更好地辅助调试。Function Tracer 可以跟上述各种

Latency Tracer 结合，实现时，各个 Latency Tracer 会注册一个自己的 Tracing function，在 mcount 插入点中调用用于收集特定信息。例如，在 wakup latency tracer 中，注册了 wakeup_tracer_call()函数。

- Event Tracer[66]

Event Tracer 用于跟踪系统事件，比如时钟、系统调用和中断等。Event Tracer 不是 Ftrace 的一个插件，当它使能后，相关结果会记录到所有插件结果中，甚至包括 nop Tracer。因为 Function Tracer 在使能后会产生一定负载，所以在没有 Function Tracer 时可以考虑开启负载较少的 Event Tracer。

- Schedule switch Tracer

Schedule switch Tracer用于跟踪任务被调度即正文切换的情况，从而辅助跟踪任务切换时间以及任务是否被正确调度，例如可以用于测试基于优先级的调度策略是否被正确实现。第 6 章会介绍其用法。

4.1.2 确定平台相关性

初步了解这些 Tracers 后，现在来分析它们的平台相关性。首先通过前面提到的 Tracers 的内核配置菜单查看这些 Tracers 是否存在。在 MIPS 平台下，只有 Function Tracer 没有实现。接着查看所有 Tracers 所在的内核配置文件：kernel/trace/Kconfig，找到 Function tracer 对应的内核配置选项，分析它们依赖的内核配置选项，确定它们对平台是否存在依赖：

kernel/trace/Kconfig：

```
config FUNCTION_TRACER
        bool "Kernel Function Tracer"
        depends on HAVE_FUNCTION_TRACER
    ...
config DYNAMIC_FTRACE
        bool "enable/disable ftrace tracepoints dynamically"
        depends on FUNCTION_TRACER
        depends on HAVE_DYNAMIC_FTRACE
    ...
config FUNCTION_GRAPH_TRACER
        bool "Kernel Function Graph Tracer"
        depends on HAVE_FUNCTION_GRAPH_TRACER
    ...
```

可见，要让 MIPS 支持 Function Tracer，必须实现 HAVE_FUNCTION_TRACER，HAVE_FUNCTION_GRAPH_TRACER 和 HAVE_DYNAMIC_FTRACE。

另外，在所有Tracers跟踪结果中，有一个重要信息，那便是时间戳。对于实时调试，时间戳的分辨率非常重要，要求分辨到us级[67]。默认情况下，MIPS平台上的Irqsoff， Preemptoff的跟踪结果中的时间戳的后三位都是 0，这说明MIPS平台上的时间戳分辨率只有 1ms，所以必须设法找出相关函数，分析其获取其精度不够的原因。

内核文件 kernel/trace/trace_clock.c 中定义了三个获取系统时间戳的函数 trace_clock_local()，trace_clock()，trace_clock_global()，三个函数最终都会调用 sched_clock()。因此，必须确保 sched_clock()获取高分辨率的时间戳。

在内核的 kernel/sched_clock.c 中，有一个平台无关通用的 sched_clock()实现：

kernel/sched_clock.c:

```
unsigned long long __attribute__((weak)) sched_clock(void)
{
        return (unsigned long long)(jiffies - INITIAL_JIFFIES) * (NSEC_PER_SEC / HZ);
}
```

从该实现可以看出，它基于系统的 jiffies，而 jiffies 的分辨率和 HZ 相关，即使 HZ 被设置为 1000，分辨率也只有 1s/1000=1ms。可见该通用的 sched_clock()远远无法满足实时调试的要求。

由于没有平台相关的特定实现，MIPS 只能使用该通用的 sched_clock()，这就是时间戳精度不够的原因。因此，必须实现 MIPS 平台特定的 sched_clock()。

下面以 MIPS 平台为例讨论高分辨率 sched_clock()和 Function Tracer 的实现。

4.1.3 高分辨率sched_clock()

4.1.3.1 硬件支持

为实现高分辨率的 sched_clock()，必须有相应的硬件支持。

比如在 X86 平台上，有一个 64 位的 TSC 寄存器，用于计数 CPU 的时钟周期，并提供专门的指令读取该寄存器。通过读取该寄存器可以快速获取时间戳。由于 TSC 的计数频率跟 CPU 频率相同，如果处理器的主频为 400M，那么它能提供高达 1/(400*10^6)=2.5ns 的计数分辨率。通过阅读 arch/x86/kernel/tsc.c 源码发现，在 TSC 没关闭时，X86 正是通过访问该寄存器实现其 sched_clock()的。

而在 MIPS 平台上，不同的变体有所不同，例如，同 X86 平台一样，Cavium Octeon 提供了一个 64 位的 MIPS C0 计数器，但是龙芯等其他 MIPS 变体仅仅提供了一个 32 位的 MIPS C0 计数器。在龙芯上，该计数器的计数频率是处理器主频的一半，龙芯的主频为 800M，因此计数分辨率可以达到 2.5ns。

4.1.3.2 实现要求

实现前讨论几个 sched_clock()的需求。

1. 关闭变频支持

无论是 X86 还是 MIPS，相关计数器的计数频率和主频关联，如果系统主频变化，那么计数将会混乱。因此，如果相关计数器计数频率跟主频关联并且想开启 sched_clock()，那么必须关闭处理器的变频支持。其实，关闭变频也是实时系统正常工作的要求，否则系统时间将会混乱，所有任务调度将变得不可预测。该部分还会在后面讨论到。

2. 开销小，快速高效

使用 Ftrace 调试会给系统带来额外负载，而 sched_clock()作为获取时间戳的函数，在 Ftrace 中被广泛调用，因此应该尽量减少 sched_clock()的开销，让其快速高效，这也是 Ftrace 为什么不直接使用系统原有高精度 getnstimeofday()来获取时间戳的原因（见 trace_clock_global()函数的注释）。

3. 必须考虑计数器潜在的溢出

对于 64 位的计数器，如果计数频率是 400M，将在 1/400M * (2^64-1) = 87741 天左右溢出，对于实时调试来说，已经足够。但对于一个 32 位的计数器和同样的计数频率，将在 10 多秒后溢出。因此，必须采取办法避免溢出[7]。

4. 本身不能被 Trace

使用 Function Tracer 后会让所有函数调用一个特定的 mcount()函数，对于 sched_clock()也不例外，而 sched_clock()本身在 mcount 中被 Function Tracer 间接或者直接调用以便获取时间戳，从而存在如下的调用关系：

```
mcount() --> sched_clock() --> mcount()
```

上述调用是个死循环，必须避免。因此，sched_clock()本身不能被 Trace，如何做到呢？Linux 内核通过宏 notrace 或在 Makefile 中配置 CFLAGS_REMOVE_file.o = -pg 实现。notrace 用于指定某个函数不被 trace，而 CFLAGS_REMOVE_file.o = -pg 则用于指定在编译该文件时不使用-pg 选项，因而该文件中的所有函数都不会被 trace。例如，notrace 用法如下：

```
notrace unsigned long long sched_clock(void)
{
}
```

而 CFLAGS_REMOVE_file.o = -pg 的用法如下：

```
arch/x86/kernel/Makefile:
ifdef CONFIG_FUNCTION_TRACER
```

[7] 对于一个固定位数的计数器，溢出是无法从根本上避免的，只能设法延迟溢出发生的时间。

```
# Do not profile debug and lowlevel utilities
CFLAGS_REMOVE_tsc.o = -pg
...
Endif
```

如上所示，相应的 Makefile 通过 CFLAGS_REMOVE_tsc.o = -pg 避免对定义了 sched_clock()函数的文件 arch/x86/kernel/tsc.c 采用-pg 编译选项，因而不会在 sched_clock()函数中插入 mcount()函数，从而避免死循环。实现时可根据需要选择这两种方法中的一种。

5. 返回时间单位为 ns

Ftrace 要求 sched_clock()返回的时间单位为 ns，跟通用的 sched_clock（）一致，类型被定义为 unsigned long long，即 64 位的无符号整形，因此，从计数器获取到时钟周期数以后，需要转换为 ns，换算过程既要保障精度，又要防止潜在的溢出并降低计算的复杂度以减少负载。

4.1.3.3 具体实现

对于关闭变频，可通过禁用 Linux 的内核配置选项 CONFIG_CPU_FREQ 来实现，下面来考虑其他几个设计原则。

首先根据要求设计 sched_clock()的原型：

```
notrace unsigned long long sched_clock(void)
{
    unsigned long long time;
    unsinged long long clock;

    clock = read_c0_clock(); /* 读取 MIPS C0 计数器中的时钟数 */
    time = cyc2ns(clock);  /* 把时钟数转换为 ns 数 */
    return time;
}
```

Sched_clock()返回类型为 *unsigned long long*，需要通过 notrace 禁止被 Trace，其主要工作是读取计数器的时钟：read_c0_clock()，并转换为 ns 数：cyc2ns()。

下面考虑 read_c0_clock()和 cyc2ns()的实现。

1. 读取 MIPS C0 计数器的时钟数: read_c0_clock()

读取时钟时，对于拥有 64 位计数器的 Cavium octeon，可以直接调用 read_c0_cvmcount()函数，但是对于 32 位的计数器，必须考虑潜在的溢出。下面讨论两种避免溢出的措施。

方法一：时钟差累加

资料[72]提出了对时钟差进行累加的方法，大概原理如下：

记录当前时钟数跟前一次时钟数的差，并把这些差值累加起来，得到的总时钟数就是当前的实际时钟数。计算过程如下：

1. 初始化实际时钟数为 clock 为 0
2. 把旧的时钟数 old_clock 初始化为 0
3. 读取当前时钟数 current_clock
4. 求差值：如果 current_clock 比 old_clock 大，直接求差，如果 old_clock 比 current_clock 大，求差值后加上该计数器能够计数的最大值，即：delta = (curernt_clock - old_clock) & mask, 其中 mask = (2^计数器位数) - 1
5. 把差值累加 clock = clock + delta
6. 把当前时钟数赋值为旧的时钟数 old_clock = current_clock

实现原型如下：

```
#define mask ((1ULL << 32) - 1)
DEFINE_RAW_SPINLOCK(clock_lock);
static inline unsigned long long read_c0_clock(void)
{
        static u64 clock;
        static u32 old_clock;
        u32 current_clock;
        raw_spin_lock(&clock_lock);
        current_clock = read_c0_count();
        clock += ((current_clock - old_clock) & MASK);
        old_clock = current_clock;
        raw_spin_unlock(&clock_lock);
        return (unsigned long long)clock;
}
```

上述方法需要考虑一个问题：假设计数器在计满时间T（溢出周期）后溢出，如果sched_clock()在该时间段内没有被调用，那么最终记录的时钟数将错过对这个时间段的累加。因此，必须想办法进行补偿。资料[72]通过sched_clock()中记录的jiffies来补偿，如果系统的jiffies更新了，说明sched_clock()在该jiffies时间内没有被调用，需要通过jiffies更新sched_clock()记录的时钟数(clock)，但是由于jiffies的分辨率不高，将会影响sched_clock的分辨率。为避免带来分辨率损失，可以考虑通过Linux下的定时器来实现sched_clock()的定时调用，并且定时器的更新周期必须小于计数溢出周期。

该方法还需禁止抢占（见粗体部分），以避免抢占时错过对 clock 的累加。

方法二：把 32 位的硬件计数器抽象为 63 位的虚拟计数器

该方法把一个 32 位的计数器扩展为一个虚拟的 63 位计数器，并被抽象为 include/linux/cnt32_to_63.h 里头的一个宏 cnt32_to_63()。

它把虚拟的 63 位计数器的高 31 位放在内存，低 32 位保留在硬件计数器，而内存中的最高位用于同步硬件计数器中的时钟，要求至少每半个计数溢出周期同步一次，当经过一个溢出周期后，将进位以完成内存中的高 31 位的计数。

基于 cnt32_to_63()的 read_c0_clock()可以实现为：

```
static inline unsigned long long read_c0_clock()
{
    unsigned long long clock;
    clock = cnt32_to_63(read_c0_count());
    if (clock & 0x8000000000000000)
        clock &= 0x7fffffffffffffff;
    return clock;
}
```

类似地，该方法也要求定时调用，以免错过对内存中高 31 位的进位，不过定时调用周期必须少于计数溢出周期的一半。另外，因为内存中的最高位用于同步硬件时钟，所以返回时需要清理掉。cnt32_to_63()的源码如下：

include/linux/cnt32_to_63.h:

```
#define cnt32_to_63(cnt_lo) \
({ \
        static u32 __m_cnt_hi; \
        union cnt32_to_63 __x; \
        __x.hi = __m_cnt_hi; \
        smp_rmb(); \
        __x.lo = (cnt_lo); \
        if (unlikely((s32)(__x.hi ^ __x.lo) < 0)) \
                __m_cnt_hi = __x.hi = (__x.hi ^ 0x80000000) + (__x.hi >> 31); \
        __x.val; \
})
```

以上两种方法都能有效处理 32 位计数器的溢出问题，通过比较看出，前者的计算复杂度比后者要高，考虑到sched_clock()的开销问题，在应用中采用后者。第 6 章会对两者的开销进行评测。

2. 把时钟数转换为 ns 数: cyc2ns()

假设时钟频率为 freq，每个时钟为 1/freq (s)，即 1/freq * NSEC_PER_SEC (ns)，cycles 个时钟的 ns 数可以通过如下公式求得（这里的 NSEC_PER_SEC 为 10^9）：

$$ns = cycles * (NSEC\_PER\_SEC / freq)$$

实际运算时，由于内核不支持浮点运算，只支持整数的除法运算，会带来很大的精度损失，因此上述公式会进行如下转换（参考 arch/x86/kernel/tsc.c）：

$$ns = cycles * (NSEC\_PER\_SEC * SC / freq / SC)$$

如果 SC 取值为 2^N，即假设 SC = 2^N，上述公式可以转换为：

$$ns = (cycles * (NSEC\_PER\_SEC << N) / freq) >> N$$

$$= (cycles * ((NSEC\_PER\_SEC << N) / freq)) >> N$$

设 cyc2ns_scale = (NSEC_PER_SEC * SC / freq)，公式可写作：

$$ns = (cycles * cyc2ns\_scale) >> N$$

为保证足够精度，N 值在 cyc2ns_scale 符合 32 位无符号整数时越大越好。因为 Linux 内核的 clocksource 结构体的两个成员：mult 与 shift 分别对应 cyc2ns_scale 与 N，而 mult 和 shift 都被限制为 32 位整数，因此，上述公式可记为：

$$ns = (cycles * mult) >> shift$$

在 MIPS 下，mult 与 shift 在 clocksource_set_clock()函数中计算。这里需要考虑另外一个问题，那就是 sched_clock()的返回值是 64 位的无符号整形，要防止溢出。因为 cycles 也是 64 位的无符号整数，cycles * mult 的结果可能超过 64 位，因此，需要在计算中间结果时转换为 128 位，具体处理办法见 2.6.33 内核文件 arch/mips/cavium-octeon/csrc-octeon.c 中的 sched_clock()实现。

4.1.3.4 实现效果

以 Irqsoff Tracer 为例，在采用高分辨率 sched_clock()前后的效果如下（只保留了时间戳信息）：

基于 jiffies 的 sched_clock()
tracer: irqsoff # # irqsoff latency trace v1.1.5 on 2.6.33.1-rt11 # -- # latency: 1000 us, #31/31, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0) # ----------------- # | task: -4 (uid:0 nice:0 policy:1 rt_prio:49) # ----------------- # => started at: handle_int

```
#   => ended at:    schedule
...
    <...>-1355      0d....      0us : trace_hardirqs_on <-restore_partial
    <...>-1355      0d....      0us!: handle_int
sirq-tim-4          0d.... 1000us : schedule
sirq-tim-4          0d.... 1000us : trace_hardirqs_on <-schedule
sirq-tim-4          0d.... 1000us : <stack trace>
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper
```

高分辨率的 **sched_clock()**:

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 2.6.33.1-rt11
# --------------------------------------------------------------------
# latency: 90 us,  #4/4,  CPU#0 | (M:preempt VP:0,  KP:0,  SP:0 HP:0)
#      -----------------
#      | task: -4 (uid:0 nice:0 policy:1 rt_prio:49)
#      -----------------
#   => started at: handle_int
#   => ended at:    schedule
...
    <...>-1492      0d....      3us+: handle_int
sirq-tim-4          0d....     89us+: schedule
sirq-tim-4          0d....     92us+: trace_hardirqs_on <-schedule
sirq-tim-4          0d....     96us : <stack trace>
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper
```

通过上述比较（粗体部分）发现，在实现高分辨率的 sched_clock()后，Ftrace 追踪结果能够提供 us 级的时间戳，但是之前的结果只能提供 ms 级。

4.1.4 Function Tracer

4.1.4.1 工作原理和编译器支持

Function Tracer 依赖于编译器支持，使用不同内核编译选项时，产生的代码可能有较大差异，实现时不可能兼顾所有情况但应尽可能满足内核默认编译选项及其组合，对于那些内核中目前没有使用的选项，需要综合实际需求予以考虑。

下面分别从静态 Function Tracer、Function Graph Tracer 和动态 Function Tracer 三部分讨论其工作原理及相应的编译器支持。

1. 静态 Function Tracer：CONFIG_FUNCTION_TRACER

从内核主目录下的 Makefile 可以找到:

```
ifdef CONFIG_FUNCTION_TRACER
KBUILD_CFLAGS    += -pg
Endif
```

上述代码表示，如果配置了 CONFIG_FUNCTION_TRACER（即开启 Function Tracer），则使用 gcc 的-pg 选项编译内核，而 gcc 使用手册显示该选项在函数入口插入额外代码以提供程序分析信息给 gprof 工具。这些额外代码即对 mcount() 函数的调用，在 X86 平台上执行如下命令可以看到确实插入了"call mcount"：

```
$ echo 'main(){}' | gcc -x c -S -o - - -pg | grep mcount
    call mcount
```

GCC会给mcount()传递两个参数[8]，以下表中的foo()函数中的call mcount为例，ip是mcount()的返回地址，parent_ip为foo()函数自身的返回地址，通过这两个参数并结合可执行文件中的符号表，就可以方便地给出函数的调用关系，gprof正是通过mcount()获取函数调用关系。

```
bar () {
    foo ()
        <-- parent_ip
}
foo () {
    call mcount
        <-- ip
}
mcount (unsigned long ip,    unsigned long parent_ip) {
```

[8] mcount 函数与普通函数的参数传递时的寄存器用法不一样

```
    ...
}
```

对于普通应用程序，该 mcount()函数在 libc 中实现，可以通过 nm 查看到：

```
$ nm /lib/libc.so.6 | grep " mcount"
000d1770 W mcount
```

但由于 Linux 内核不能链接外部的 libc 库，因此 Ftrace 必须在内核中重新实现 mcount()，并且，诸如寄存器的保护，参数的传递等方面可以参考 libc 中的实现。对于静态的 Function Tracer（只配置了 CONFIG_FUNCTION_TRACER)，mcount()会直接调用 ftrace_trace_function(ip， parent_ip)。由于 mcount()需要保存和恢复寄存器，通过寄存器传递参数，需要用汇编实现，因此和平台相关。

如果计划移植 Function Tracer 到特定平台，必须确保编译器为该平台提供了-pg 选项。为了更好地支持 Function Tracer，可能还需要对目标平台的-pg 选项进行适当的优化（具体讨论见后文)。

2. 图形化 Function Tracer：CONFIG_FUNCTION_GRAPH_TRACER

在 Ftrace 中的 Function Tracer 出现前，有其他人开发了另外一个 Kernel Function Tracing补丁，即KFT: http://elinux.org/Kernel_Function_Trace，也用于内核函数的跟踪和调试。

与 Function Tracer 不同的地方在于，它采用的是 gcc 的-finstrument-functions 选项。开启该选项后，GCC 在函数入口和出口各插入一个函数：

```
$ echo 'main(){}' | gcc -x c -S -o - - -finstrument-functions | grep profile | grep call
    call __cyg_profile_func_enter
    call __cyg_profile_func_exit
```

而同样地，在 libc 中也实现了这两个函数：

```
$ nm /lib/libc.so.6 | grep __cyg_profile_func_
000e1960 T __cyg_profile_func_enter
000e1960 T __cyg_profile_func_exit
```

对于普通应用程序，gprof 利用这两个函数记录被跟踪函数的起止执行时间，进而计算它们的时间开销。KFT 作为一个内核补丁，实现了这两个函数，能够统计内核的函数调用及时间开销，从而辅助开发人员进行系统性能分析。

虽然基于-pg 的 Function Tracer 仅在内核函数入口插入了 mcount()，但是它通过一定技巧模拟了类似-finstrument-functions 的功能，因此也能获取到函数本身的时间开销，该技巧原理如下：

```
bar () {
    foo ()
        <-- parent_ip
```

```
}
foo () {
    /* 在调用 mcount 之前会保存 parent_ip 到栈中 */
    call mcount
        <-- ip
    /* 从栈中取出 parent_ip */
    /* foo()执行完以后会返回到 bar()中的 parent_ip 处 */
}
mcount (unsigned long ip,  unsigned long parent_ip) {
    prepare_function_return(...) /* 保存 parent_ip，把栈内 parent_ip 改为 Ftrace
的 return_to_handler 函数，另外，该函数还调用其他函数记录当前时间戳等信息
*/
}
return_to_handler() {
    parent_ip = ftrace_return_to_handler(...) /* 记录当前时间戳等信息并取出之
前保存的 parent_ip */
    /* 跳到 parent_ip 处返回 bar() */
}
```

通过上述技巧，调用 mcount()时，prepare_function_return()起到类似__cyg_profile_func_enter()的作用，记录函数开始执行的时间戳 calltime，并劫持（hijack）保存在栈内的 foo()的返回地址(parent_ip)为 return_to_handler()，因此当 foo()函数从 mcount()返回并试图取出栈内的 parent_ip 时，取出了 return_to_handler()的地址，并跳转到 return_to_handler()执行，进而调用 ftrace_return_to_handler()获取此时的时间戳（退出函数），记为 rettime，之后，该函数取出保存好的 foo()的真实返回地址，返回到 bar()中。这样，ftrace_return_to_handler()起到了类似__cyg_profile_func_exit()的功能。

实际上，这就是开启配置选项 CONFIG_FUNCTION_GRAPH_TRACER 后需要实现的功能，这部分功能因为涉及到栈的保护与访问等，因此也和平台相关。

由此可见，Ftrace 能够实现 KFT 的所有功能，并且在开启 CONFIG_FUNCTION_GRAPH_TRACER 后，还直接提供了可视化的树状输出结果，内核开发人员只需 cat 命令就可以直接阅读跟踪结果，而 KFT 则不然，它需要额外的复杂脚本来辅助输出结果的分析，因此 Ftrace 的 Function Tracer 最终被官方内核接收，而 KFT 至今还是以补丁的方式维护。

3. 动态 Function Tracer

除以上功能外，在 CONFIG_DYNAMIC_TRACER 的帮助下，可以实现动态 Function Tracer，可以动态地指定需要跟踪的函数。

配置 CONFIG_DYNAMIC_TRACER 后，编译内核时会调用一个 Perf 脚本：scripts/recordmcount.pl。该脚本把所有 mcount()调用位置（实现时不一定就是调用位置，跟平台相关）记录下来，并保存到内核映像文件的__mcount_loc 段中。

内核调用 ftrace_init()初始化 Ftrace 时通过调用 ftrace_convert_nops()函数进而通过平台相关的 ftrace_make_nop()函数把___mcount_loc 段中所有 mcount()的调用点替换为 nops（或者说，取消对 mcount 的调用），因此，系统启动后，Ftrace 的开销很小。进入系统并挂载 debugfs 文件系统后，用户可以通过 current_tracer 接口设定当前的 Tracer 为 function，并通过 set_ftrace_filter 接口指定需要跟踪的函数，此时 Ftrace 会通过 ftrace_replace_code()调用 ftrace_make_call()开启指定函数中的 mcount()调用点。

因此，为实现动态 Function Tracer，需要修改 scripts/recordmcount.pl 并实现 ftrace_make_nop()和 ftrace_make_call()等平台相关函数。另外，对于有些平台，由于模块地址空间和内核地址空间不一致，在内核和模块中调用 mcount()的方式不一样，所以，在平台相关实现中需要区别对待内核与模块。

4.1.4.2 实现要求

在理解 Ftrace 的工作原理后，来分析实现 Ftrace 时需要注意的一些事项。

1. 开销小，快速高效

虽然 Ftrace 比 KFT 引入的开销更小，但还是有一定开销，为让 Trace 的结果与真实结果更接近，在实现时应尽可能减少开销，例如，尽量用汇编语言实现，尽量优化相关算法，甚至在必要情况下改进 gcc 对-pg 的实现。

2. 不需要 Trace 的函数

有些函数无法被调试，例如那些加有__init 标记的函数，在系统启动后将从内存中移除，同样地对于内核压缩映像中的函数，解压完后就失效了。

有些函数，在对它们调试时会产生副作用，例如 early_printk，它们自身用于调试，如果插入额外代码，会产生较大开销，甚至对它们的功能造成影响。

而另外一些函数，比如前面提到的 sched_clock()，以及 Ftrace 本身的所有函数都不能被 Trace，否则会进入死循环。

为了阻止对某些函数进行追踪，可参考4.1.3。用notrace标记函数或者用CFLAGS_REMOVE_file.o = -pg禁止对某个文件用-pg编译。

3. 需要考虑模块支持

对模块的支持非常重要，因为大多数设备驱动都以模块的形式开发。

前面提到，有些平台上，在模块和内核里头，调用 mcount()的方式可能不一样，在实现 Function Tracer 时，需要同时考虑对内核与模块的支持。

4. 要考虑不同的平台 ABI

例如，MIPS 处理器支持 32 位和 64 位，当内核配置分别为 32 位和 64 位时，指令的 ABI 不一致，因为 Function Tracer 涉及到汇编语言的实现，因此必须参考

处理器平台的指令手册，研究不同的 ABI，包括栈的操作、参数的传递等。

5. 良好的容错处理

对于动态 Function Tracer，涉及到对内存指令空间的访问和修改，容易出现缺页，需要一定的容错措施降低调试工具对原系统的影响，提高工具本身和原系统的健壮性。

4.1.4.3 具体实现

下面讨论 MIPS 平台上的 Function Tracer 实现。

考虑到时间关系，本文仅以 64 位 MIPS 为例。官方 Linux 2.6.33 以及后续版本都有相关支持，源码部分请参考 arch/mips/kernel/{mcount.S， ftrace.c}，scripts/recordmcount.pl 以及 arch/mips/include/asm/ftrace.h。

下文涉及到很多MIPS平台的特性，比如MIPS汇编语言，请参考[72]。另外，对于文中涉及到的一些Linux内核开发细节，请参考[77]，而关于Perl程序设计部分请参考[78]。

1. 静态 Function Tracer

MIPS 平台上内核对 mcount()的函数调用如下：

```
$ echo 'main(){}' | gcc -mabi=64 -x c -S -o - - -pg | grep -A1 -B 2 mcount
    .set noat
    move    $1，$31          # save current return address
    jal   _mcount
    .set at
```

上面的$31 为ra寄存器，$1 为临时寄存器AT。$31 保存着当前的函数返回地址，即前文提到的parent_ip，执行完move $1， $31 后，parent_ip被保存到了AT寄存器中，之后执行jal _mcount[9]跳转到_mcount()函数中，并把ra寄存器中的内容修改为jal _mcount的下一条指令的地址，即前文提到的ip，因此在mcount()中可以很方便地获取到ip和parent_ip的值。

而模块对mcount()的调用，因为涉及到从模块空间（0xc0000000）到内核空间（0x80000000）的调用[10]，地址跨越超过 28 位，而jal指令只保留了 26 位的地址编码，因此无法通过jal指令直接跳转，所以需要使用长跳转，即先把mcount()的地址装载到一个寄存器中，然后通过寄存器跳转：

```
lui     v1，  hi_16bit_of_mcount
addiu   v1，v1，low_16bit_of_mcount
.set noat
```

[9] 有些平台上的 mcount()函数名为_mcount()，本文在大部分情况下直接使用 mcount()

[10] 见 arch/mips/kernel/vmlinux.lds.S 和 arch/mips/kernel/module.c

```
move   $1，$31
jalr v1
  nop
.set at
```

对于内核与模块，除了调用 mcount()的方式不同以外，其他都一样，因此，对于静态 Function Tracer，因为只需要实现 mcount()，而无论是内核还是模块，在调用 mcount()后都传递了 mcount()需要的 ip 和 parent_ip 参数，因此，对于模块和内核来说，静态 Function Tracer 的实现是一样的。

在正式讨论 mcount()前，先讨论跟 mcount()相关的其他几个内容。

- 函数指针 ftrace_trace_function 与空函数 ftrace_stub()

Function Tracer 定义了一个全局的 ftrace_func_t 类型函数指针 ftrace_trace_function，该指针指向的函数被 mcount()调用，完成具体 trace 动作。

ftrace_stub()是一个空函数，Ftrace 把 ftrace_trace_function 等 ftrace_func_t 类型的函数指针初始化为它。

当用户使用 Tracer 时，会把 ftrace_trace_function 通过 register_ftrace_function() 指向对应 Tracer 自身的跟踪函数，例如，静态 Function Tracer 注册了一个 function_trace_call()。而当退出某个 Tracer 时，相应的跟踪函数会被注销掉，ftace_trace_function 被重新指向 ftrace_stub()。

其中，ftrace_stub 的原型如下：

```
void ftrace_stub(void)
{
    return;
}
```

在 MIPS 中，可以实现为：

```
    .globl ftrace_stub
ftrace_stub:
    jr ra     /* ra 寄存器存放了返回地址 ip，直接跳转到 jal _mcount 中 */
     move ra，  AT  /* 跳转时从 AT 临时寄存器恢复旧的返回地址 parent_ip */
```

在 MIPS 的跳转指令后面有一个时钟周期的延迟槽，可以插入空指令或者放入一条执行时间为一个周期的指令。在编写汇编代码时，习惯上要求在延迟槽中的指令前添加一个空格作为缩进，以便程序员对程序的阅读和维护。

- 开启和关闭 Function Tracer

内核定义了一个 function_trace_stop 变量。当用户往 debugfs 文件系统中的 tracing_enabled 文件中写入 1 或 0 时，分别设置 function_trace_stop 为 0 或 1，相应地开启和关闭 Function Tracer。

因此在 mcount()调用 ftrace_trace_function 指向的函数时，需要先判断 function_trace_stop，如果 function_trace_stop 为真，直接调用 ftrace_stub()返回。

关于 function_trace_stop 的判断，如果在 mcount()不进行判断，那么 ftrace_trace_function 会首先调用 ftrace_test_stop_func 对其进行判断，接着调用 __frace_trace_function 指向的函数，即在切换 Tracer 时，真正注册的会是 __ftrace_trace_function，而不是 ftrace_trace_function。反之，如果在 mcount() 中对 function_trace_stop 进行了判断，则可以定义 HAVE_FUNCTION_TRACE_MCOUNT_TEST 以便关闭通用的 C 语言实现。

如果 ftrace_trace_stop 为假，要判断 ftrace_trace_function 指向的函数是否为 ftrace_stub()，如果是则直接调用 ftrace_stub()返回，否则调用 ftrace_trace_function 指向的函数。但在调用 ftrace_trace_function 指向的函数前后，需要遵循处理器 ABI 规定，通过栈进行寄存器的保护和恢复，并进行相应的参数传递。

因此，mcount()的原型如下：

```
void mcount(unsigned long ip,  unsigned long parent_ip) {
    if (!function_trace_stop) {
        if (function_trace_stop != ftrace_stub) {
            /* 保护寄存器 */
            /* 传递参数 */
            ftrace_trace_function(ip,  parent_ip);
            /* 恢复寄存器 */
        }
    }
    return;
}
```

上述注释部分跟处理器的 ABI 有关，对于特定平台，可以参考 libc 的实现，例如，在 MIPS 平台上，可以通过 objdump 命令查看到其实现：

```
$ objdump -d /lib/libc.so.6 | grep -A10 "mcount>:"
000f7e30 <_mcount>:
    f7e30:   3c1c0009    lui  gp，0x9
    f7e34:   279ceb30    addiu   gp，gp，-5328
    f7e38:   0399e021    addu    gp，gp，t9
    f7e3c:   27bdffd0    addiu   sp，sp，-48
    f7e40:   afbc002c    sw  gp，44(sp)
    f7e44:   afa40018    sw  a0，24(sp)
```

```
f7e48:  afa5001c    sw  a1，28(sp)
f7e4c:  afa60020    sw  a2，32(sp)
f7e50:  afa70024    sw  a3，36(sp)
f7e54:  afa20028    sw  v0，40(sp)
f7e58:  afa10010    sw  at，16(sp)
f7e5c:  afbf0014    sw  ra，20(sp)
f7e60:  03e02821    move    a1，ra
f7e64:  00202021    move    a0，at
f7e68:  8f998c18    lw  t9，-29672(gp)
f7e6c:  00000000    nop
f7e70:  27397eac    addiu   t9，t9，32428
f7e74:  0411000d    bal f7eac <_mcount+0x7c>
f7e78:  00000000    nop
```

内核默认没有使用 gp 寄存器（因为使用了-G0 编译），所以不用考虑 gp 的保护和恢复。又因为调用 mcount()前没有使用 v0 寄存器，因此也无须保护 v0，另外，由于 ftrace_trace_function 指向的函数参数中，第一个为 ip，第二个为 parent_ip，而 ra 存放的是 ip，at 存放的是 parent_ip，因此，内核 mcount()的参数传递顺序跟 libc 的不一致。除此之外，因为这里的 libc 遵循 o32 ABI，而内核遵循 64 位的 ABI（仅考虑 64 位），所以参数寄存器个数会更多，还有 a4 到 a7。

因此，MIPS 中需要保护的寄存器，主要有参数寄存器 a0 到 a7，返回地址寄存器 ra 以及临时变量寄存器 AT(at)，而参数传递是通过 a0 和 a1 来实现，分别用于传递 ip 和 parent_ip，因为 mcount()的 ip 和 parent_ip 存放在 ra 和 AT 寄存器中。因此内核中的参数传递为：

```
move    a0，    ra
move    a1，    AT
```

对于寄存器的保护和恢复，以及之前的函数返回操作因为都比较常用，都可以抽象为宏来实现。另外，考虑到兼容 32 位和 64 位 ABI 的需要，一些指令，诸如 32 位上的 sw 和 64 位上的 sd 不一样，但是在 MIPS 中，可以通过宏 PTR_S 来自动根据内核配置选择合适的指令，又比如，由于 64 位和 32 位 MIPS 需要保存的寄存器个数不一样，为了兼容两种不同的 ABI，对于栈的深度以及寄存器的存放位置等都可以通过前缀为 PT_的宏来处理。综上所述，上述原型可以实现为：

```
void mcount(unsigned long ip，  unsigned long parent_ip) {
    if (!function_trace_stop) {
        if (function_trace_stop != ftrace_stub) {
            MCOUNT_SAVE_REGS    /* 保护寄存器 */
```

```
            MCOUNT_SET_ARGS      /* 传递参数 */
            ftrace_trace_function(ip,  parent_ip);
            MCOUNT_RESTORE_REGS    /* 恢复寄存器 */
        }
    }
    RETURN_BACK;
}
```

关于宏的实现请参考内核源码，下面给出 mcount()和 ftrace_stub()的实现：

```
NESTED(_mcount,  PT_SIZE,  ra)
        lw      t1,  function_trace_stop
        bnez    t1,  ftrace_stub
         nop
        PTR_LA  t1,  ftrace_stub
        PTR_L   t2,  ftrace_trace_function /* Prepare t2 for (1) */
        beq     t1,  t2,  ftrace_stub
         nop
        MCOUNT_SAVE_REGS   /* 保护寄存器 */
        MCOUNT_SET_ARGS     /* 设置参数 */
        jalr    t2                  /* (1) call *ftrace_trace_function */
        MCOUNT_RESTORE_REGS   /* 恢复寄存器 */
        .globl ftrace_stub
ftrace_stub:
        RETURN_BACK
        END(_mcount)
```

上述函数可以类似 X86，PowerPC 平台一样存放到 entry*.S 中，但考虑到日后维护的方便，作者在实现时，把它们放到了一个新的文件 arch/mips/kernel/mcount.S 中，并在 arch/mips/kernel/Makefile 中添加了如下支持：

```
obj-$(CONFIG_FUNCTION_TRACER)    += mcount.o
```

另外，前面介绍到有些文件不能跟踪，例如 early_printk，因此还添加了：

```
ifdef CONFIG_FUNCTION_TRACER
CFLAGS_REMOVE_early_printk.o = -pg
Endif
```

实现 mcount()后，需要在模块中访问到它：

arch/mips/kernel/mips_ksyms.c:

```
#ifdef CONFIG_FUNCTION_TRACER
/* _mcount is defined in arch/mips/kernel/mcount.S */
EXPORT_SYMBOL(_mcount);
#endif
```

还需要定义宏 MCOUNT_ADDR 和 MCOUNT_INSN_SIZE，前者为_mcount()的地址，而后者为 jal _mcount 指令占用的字节数，MIPS 的指令是定长的，长度是 4 个字节，因此，可以直接定义为 4。MCOUNT_INSN_SIZE 在 kernel/trace/ftrace.c 的 print_ip_ins()函数中调用到，用于打印调用指令。

arch/mips/include/asm/ftrace.h:

```
#ifdef CONFIG_FUNCTION_TRACER
#define MCOUNT_ADDR ((unsigned long)(_mcount))
#define MCOUNT_INSN_SIZE 4              /* sizeof mcount call */
#ifndef __ASSEMBLY__
extern void _mcount(void);
#define mcount _mcount
#endif /* __ASSEMBLY__ */
#endif /* CONFIG_FUNCTION_TRACER */
```

至此，静态 Function Tracer 就已经实现，可以在目标平台上选上 HAVE_FUNCTION_TRACER。另外，因为在汇编中实现了对 ftrace_trace_stop 的判断，因此，还需要选上 HAVE_FUNCTION_TRACE_MCOUNT_TEST：

arch/mips/Kconfig:

```
config MIPS
    ...
    select HAVE_FUNCTION_TRACER
    select HAVE_FUNCTION_TRACE_MCOUNT_TEST
    ...
```

2. 动态 Function Tracer

动态 Function Tracer 的关键在于如何取消和开启对 mcount()函数的调用。对于全局的控制，可以通过 ftrace_trace_stop 来实现，而更细粒度的控制通过开启和取消指定的调用点来实现，下面来讨论具体实现。

- 确定所有 mcount 的调用位置

Ftrace 通过 scripts/recordmcount.pl 记录各个调用点的位置：先通过 objdump

加上-hdr 选项找出调用点的偏移。该工作在编译内核的链接过程前完成。

在 X86 平台上，可以看到：

```
$ echo 'main(){}' | gcc -x c -o a.out -c - -pg
$ objdump -hdr a.out | grep -A7 -B6 mcount
Disassembly of section .text:
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp，%ebp
   3:   e8 fc ff ff ff          call   4 <main+0x4>
                4: R_386_PC32   mcount
   8:   5d                      pop    %ebp
   9:   c3                      ret
$ objdump -hdr a.out | grep mcount | tr -d '[[:space:]]' | cut -d: -f1
4
```

通过以上结果，可以找到 mcount()相对.text 段的偏移是 4（注意：相对于.text 段，而不是 main 函数），由于段在最终的 Linux 内核映像文件中的地址不确定，因此，不能参考.text 段，但可以参考附近潜在的全局变量，例如：

```
#   .section ".sched.text",   "ax"
#           [...]
#   func1:
#           [...]
#           call mcount    (offset: 0x10)
#           [...]
#           ret
#   .globl fun2
#   func2:                  (offset: 0x20)
#           [...]
#           [...]
#           ret
#   func3:
#           [...]
#           call mcount (offset: 0x30)
```

```
#          [...]
```

上面有一个全局变量 func2，那么在__mcount_loc 里头可以这么记录：

```
#  .section __mcount_loc
#  .dword func2 - 0x10
#  .dword func2 + 0x10
```

链接时，func2 - 0x10 和 func2 + 0x10 都会转换成 mcount()调用点的地址，但还可能存在另外一个情况，那就是在某个段中不存在全局函数，因此在外部文件中无法引用这些函数。解决办法是创建一个临时目标文件，把其中某个函数转换为全局的，然后跟__mcount_loc 进行链接，这样就不会出错，链接后，把相应函数改回局部的，并把链接结果保存为原目标文件，之后就可以完成所有的链接。

通过上述描述发现，其平台相关部分就是如何找到 mcount()调用点的偏移。scripts/recordmcount.pl 提供了基本框架，最关键的部分是 mcount()函数所在行的字符串匹配正则表达式：$mcount_regex，下面讨论 MIPS 平台上该表达式的实现。

首先，开启静态 Function Tracer 时，通过 objdump -hdr 查看 vmlinux 文件的输出，并找到 mcount()相关的指令，编译内核时，得到：

```
#      10:   03e0082d          move    at，ra
#      14:   0c000000          jal     0 <loongson_halt>
#                    14: R_MIPS_26    _mcount
#                    14: R_MIPS_NONE *ABS*
#                    14: R_MIPS_NONE *ABS*
#    18:   00020021          nop
```

因此，参照 X86 等其他平台的实现，该表达式可以简单地写为：

```
$mcount_regex = "^\\s*([0-9a-fA-F]+):.*\\s_mcount\$";
```

而对于模块，编译时得到：

```
#       c:   3c030000          lui     v1，0x0
#                    c: R_MIPS_HI16   _mcount
#                    c: R_MIPS_NONE   *ABS*
#                    c: R_MIPS_NONE   *ABS*
#      10:   64630000          daddiu  v1，v1，0
#                    10: R_MIPS_LO16 _mcount
#                    10: R_MIPS_NONE *ABS*
#                    10: R_MIPS_NONE *ABS*
#      14:   03e0082d          move    at，ra
```

```
#        18:   0060f809          jalr      v1
```

如果使用同样的$mcount_regs，将找到两个 mcount()的位置，并且发现即使找到两个，这两个都不是调用点而是装载 mcount()地址的位置。因此，如果必须找到调用点，需要去匹配 jalr v1，但是 jalr v1 可能是其他函数的调用点。因此，需要找到一个合适的解决办法。

动态 Function Ftracer 还要求平台实现一个 ftrace_call_adjust()函数用于调整调用点的偏移，对于上面这个调用点，如果记录第一个 mcount()的位置，那么 ftrace_call_adjust()可以实现为：

```
static inline unsigned long ftrace_call_adjust(unsigned long addr)
{
        return addr + 12;
}
```

但是，由于内核的 ftrace_call_adjust()不需要调整，而 ftrace_call_adjust()没有提供参数以判断地址来自内核还是模块中，所以该方法不可行。该函数只能简单地实现为：

```
arch/mips/include/asm/ftrace.h:
static inline unsigned long ftrace_call_adjust(unsigned long addr)
{
        return addr;
}
```

即 ftrace_call_adjust()直接返回__mcount_loc 中记录的位置。因为 ftrace_call_adjust()的方法不可行，那么得寻求其他办法。

既然记录调用点是为了用于 ftrace_make_nop()和 ftrace_make_call()取消和开启该调用点，因此只要后面有办法开启和取消该调用点，那么这里就不一定要记录调用点。对于上面那个调用点，可以通过把 0xc 处的 lui 指令替换为一条跳转指令来取消对 mcount()的调用，反之，如果要开启，则可以替换回去。例如：

lui v1, 0x0	b 1f
daddiu v1, v1, 0	
move at, ra	
jalr v1	
nop	
	1: (0xc + 12)

因此，可以只记录第一个匹配到的 mcount()位置。由于内核记录的是真实调用点，可以直接通过替换调用点为 nop 来取消和开启对 mcount()的调用，因此，在实现开启和取消对 mcount()的函数时需要对内核和模块区别对待。综上所述，对于模块，mcount_regs 可以实现为：

```
$mcount_regex = "^\\s*([0-9a-fA-F]+): R_MIPS_HI16\\s+_mcount\$";
```

由于 scripts/recordmcount.pl 提供了 is_module 来区分内核与模块，因此，$mcount_regex 最终可实现为：

```
if ($is_module eq "0") {
   $mcount_regex = "^\\s*([0-9a-fA-F]+):.*\\s_mcount\$";
} else {
   $mcount_regex = "^\\s*([0-9a-fA-F]+): R_MIPS_HI16\\s+_mcount\$";
}
```

剩下的工作就是根据 scripts/recordmcount.pl 的输入来获取特定平台的 objdump， cc， ld 等工具和它们的选项，以及用于匹配函数的$function_regex 正则表达式和存放调用点地址的数据类型 type。需要注意的是，对于那些工具，其参数可能需要考虑不同的 ABI，比如 MIPS 需要考虑 32 位和 64 位，还要考虑大端和小端。关于详细实现，请参考源码。

- 开启和取消对 mcount 的调用

通过上述介绍，开启和取消对 mcount()的调用，对于内核和模块来说是不一样的，并且基本算法在上面已经提到。因此，这里直接给出实现原型。

关于取消对 mcount()的调用，通过 ftrace_make_nop()实现：

```
int ftrace_make_nop(struct module *mod,
                    struct dyn_ftrace *rec,  unsigned long addr)
{
   /* 获取 mcount 函数调用点的位置 */
   unsingned long ip = rec->ip;
   /* 判断 mcount 来自内核还是模块， in_module()在后面讨论 */
   if (in_module(ip)) {
       /* 把调用点替换为一条跳转指令 insn1，即"b 1f" */
    } else {
       /* 把调用点置为空指令 insn2，即 nop */
    }
}
```

对于开启对 mcount()的调用，通过 ftrace_make_call()实现：

```
int ftrace_make_call(struct dyn_ftrace *rec,  unsigned long addr)
{
   /* 获取 mcount 函数调用点的位置 */
```

```
    unsingned long ip = rec->ip;
    /* 判断 mcount 来自内核还是模块， in_module()在后面讨论 */
    if (in_module(ip))
        /* 替换为原指令 insn3 */
    } else {
        /* 替换为原指令 insn4，即"jal ftrace_caller" */
    }
}
```

对于 mcount()来自内核还是模块，可以通过 mcount()调用点的地址来判断，如果该地址在内核空间(0xc0000000)，则调用来自内核，如果该地址在模块空间(0x80000000)，则调用来自模块，因此 in_module()实现如下：

```
arch/mips/include/asm/ftrace.h:
#define in_module(ip) ((ip) & 0x40000000)
```

接下来的工作就是对insn1，insn2，insn3，insn4 进行编码，对于指令insn1，insn2 可以静态编码，而对于指令insn3 和insn4，其编码跟mcount()和ftrace_caller()的地址有关，而mcount()和ftrace_caller()的地址在编译内核时才确定，所以不能静态编码，需要在内核初始化时动态地编码。对于insn1 和insn2 的编码，请参考资料[69]，对于insn3 和insn4 的编码会在下文介绍到。

需要注意的是，ftrace_make_nop()和 ftrace_make_call()以及之后将要介绍到的图形化 Function Tracer 涉及到对内存中指令的修改，在修改某条指令后，必须立即刷新 cache，因为在修改这条指令之前，原来的指令可能已装载到 cache，如果不刷新 cache，这条指令可能就不会执行。

- mcount 与 ftrace_caller

动态 Function Tracer 也需要实现一个 mcount()，但是该 mcount()只完成空操作，其原型同静态 Function Tracer 的 ftrace_stub()。实际的功能由 ftrace_caller()完成，因此该函数跟静态 Function Tracer 的 mcount()类似，但是有一点不同，该 ftrace_caller()没有直接调用 ftrace_trace_function 指向的函数，而是通过 ftrace_update_ftrace_func()动态更新该位置的调用函数。ftrace_caller()原型如下：

```
void ftrace_caller(unsigned long ip,  unsigned long parent_ip) {
    if (!function_trace_stop) {
        MCOUNT_SAVE_REGS    /* 保护寄存器 */
        MCOUNT_SET_ARGS     /* 传递参数 */
ftrace_call:  ftrace_stub()    /* 默认调用 ftrace_stub 直接返回 */
        MCOUNT_RESTORE_REGS    /* 恢复寄存器 */
    }
```

```
    RETURN_BACK;
}
```

MIPS 上的实现如下：

```
NESTED(ftrace_caller,  PT_SIZE,  ra)
        lw      t1,  function_trace_stop
        bnez    t1,  ftrace_stub
         nop
        MCOUNT_SAVE_REGS
         MCOUNT_SET_ARGS
    .globl ftrace_call
ftrace_call:
        nop     /* a placeholder for the call to a real tracing function */
        MCOUNT_RESTORE_REGS
        .globl ftrace_stub
ftrace_stub:
        RETURN_BACK
        END(ftrace_caller)
```

实现时把_mcount()函数放在 ftrace_caller()的开头，即确保_mcount()的入口地址和 ftrace_caller()的入口地址一致，_mcount()本身只完成空操作，直接返回，此时 ftrace_caller()实现如下：

```
NESTED(ftrace_caller,  PT_SIZE,  ra)
        .globl _mcount
_mcount:
        b       ftrace_stub
          nop
real_ftrace_caller:
         ...
        .globl ftrace_stub
ftrace_stub:
        RETURN_BACK
        END(ftrace_caller)
```

因为_mcount()在内核初始化后不再使用，因此，进入系统后，_mcount()可以被修改掉。如果内核初始化时把 ftrace_caller()中的第一条指令"b ftrace_stub"

修改为 nop 指令，因为只多了两条空指令，那么 ftrace_caller()将和原来一样工作。这样做的好处是，由于此时 ftrace_caller()和 mcount()的入口一致，对于模块，仅需要替换一条指令就可以实现调用点的开启和取消。

对于内核还可以进行优化，在 ftrace_caller()中再添加一个标记：real_ftrace_caller，可以把 mcount()调用点替换为"jal real_ftrace_caller"，从而可以跳过头两条 nop 指令：

```
NESTED(ftrace_caller,  PT_SIZE,  ra)
        .globl _mcount
_mcount:
        b       ftrace_stub
         nop
real_ftrace_caller:
        ...
        .globl ftrace_stub
ftrace_stub:
        RETURN_BACK
        END(ftrace_caller)
```

因此，这个实现不仅兼容了对内核和模块的支持，而且通过一定技巧，做到了尽可能的优化。

- 如何更新跟踪函数

ftrace_update_ftrace_func()动态地把 ftrace_call 位置的函数调用替换为对 ftrace_trace_function 指向的函数。实现时只需要往 ftrace_call 位置写入"jal func"指令即可。至于"jal func"指令，需要在 ftrace_update_trace_func()中动态编码，因此，其原型如下：

```
int ftrace_update_ftrace_func(ftrace_func_t func)
{
    /* 对"jal func"指令进行编码，假设为 insn5 */
    /* 在 ftrace_call 位置，写入指令 insn5 */
}
```

- 动态 Function Tracer 的初始化

对于动态 Function Tracer，有些平台可能需要初始化一些平台相关的变量，可以通过 ftrace_dyn_arch_init()实现。

对于 MIPS 平台，根据之前的讨论，需要动态编码 insn3 和 insn4，并且需要把 ftrace_caller()开头的"b ftrace_stub"替换为 nop 指令。

对于 insn3 和 insn4 的编码，可以通过内核自带的微型汇编器

(arch/mips/include/asm/usam.h， arch/mips/mm/uasm.c)实现，该汇编器提供了一些用于汇编简单指令的函数。比如对于 insn3，可以通过 USAM_i_LA_mostly()编码，而对于 insn4，可通过 usam_i_jal()编码。关于实现细节，请参考内核源码。

- 容错处理

因为动态 Function Tracer 引入了对指令内存的访问，可能出现缺页（比如内存中的内容正好被置换到了磁盘上，或者还没有加载到内存中）。出现缺页时，MMU 将抛出异常，内核此时会执行 do_page_fault()异常处理函数，并通过 fixup_exception()函数调用 search_exception_table()去查找用户自定义的异常处理操作，如果用户定义了异常处理操作，那么执行用户定义的异常处理，然后立即返回，否则内核会报告 Oops，并 die()掉。

作为调试用的 Function Tracer，不应对原有系统造成太大影响，因此应该进行缺页处理，处理后停止 Function Tracer 的工作，但并不影响原有内核的执行。

动态 Function Tracer 的函数全部定义在 arch/mips/kernel/ftrace.c 中，由于 Ftrace 自身不能被 Trace，因此需要修改 arch/mips/kernel/Makefile，加入对 ftrace.c 文件的过滤，而且需要把它编译到内核中：

```
arch/mips/kernel/Makefile:
ifdef CONFIG_FUNCTION_TRACER
CFLAGS_REMOVE_ftrace.o = -pg
endif
obj-$(CONFIG_FUNCTION_TRACER)     += mcount.o ftrace.o
```

实现完动态 Function Tracer 后，就可以添加 HAVE_DYNAMIC_FTRACE 了，当然，还有 HAVE_FTRACE_MCOUNT_RECORD。

```
arch/mips/Kconfig:
config MIPS
    ...
    select HAVE_DYNAMIC_FTRACE
    select HAVE_FTRACE_MCOUNT_RECORD
    ...
```

- 图形化 Function Tracer

图形化 Function Tracer 的关键是在 mcount()中劫持调用它的函数存放在栈内的函数返回地址。需要对函数栈的布局、寄存器的保护和恢复有清晰的了解。

下面根据其工作原理分析 mcount()， prepare_function_return()，return_to_handler() 和 ftrace_return_to_handler()的函数原型：

```
void mcount(unsigned long ip,  unsigned long parent_ip,
    unsigned long parent_ip_addr,  unsinged long framepointer) {
```

```
    ...
    ip = mcount 本身的返回弟子和
    parent_ip = 调用 mcount 的函数的返回地址
    parent_ip_addr = 栈内保存了 parent_ip 的地址
    framepointer = 当前 frame 的指针
    prepare_function_return(parent_ip_addr,  ip,  framepointer);
    ...
}
void prepare_ftrace_return(unsigned long *parent_ip,  unsigned long ip,
                                    unsigned long framepointer)
{
    ...
    *parent_ip =  (unsigned long)&return_to_handler;
    ...
}
typedef void (*return_to_parent) (void);
void return_to_handler(void)
{
    parent_ip = ftrace_return_to_handler();
    (return_to_parent)parent_ip();
}
```

假设有如下函数调用，

```
foo() {
}
main() {
    foo();
}
```

在 X86 平台上，其汇编实现如下（仅保留相关部分）：

```
$ echo 'foo(){} main(){ foo(); }' | gcc -x c -S -o - - -pg
foo:
        pushl   %ebp
        movl    %esp,  %ebp
```

```
        call    mcount
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp,  %ebp
        call    mcount
        call    foo
        popl    %ebp   <-- old_eip
        ret
```

可见，无论对于 foo(除了调用 mcount 外，没有调用其他函数)还是 main()函数，其栈的访问都一致。因此，对于 X86 来说，在 mcount 中对栈的操作一致。另外，由于 foo()要返回的地址 old_eip 总是被压入 ebp+4 的位置，因此，在 mcount 中很容易被访问到。如果要劫持该地址，只需要先通过 prepare_function_return()把 ebp+4 处的地址先保存，之后修改为 return_to_handler()，然后在 foo()返回时会先执行到 return_to_handler()，进而调用 ftrace_return_to_handler()，最后把 parent_ip 恢复到原栈中返回。

对于 MIPS（o32 ABI），其汇编实现如下（只保留相关部分）：

```
$ echo 'foo(){} main(){ foo(); }' | gcc -x c -S -o - - -G0 -pg
foo:
        addiu   $sp，$sp，-24
        sw      $fp，20($sp)
        move    $fp，$sp
        .set    noat
        move    $1，$31             # save current return address
        subu    $sp，$sp，8         # _mcount pops 2 words from   stack
        jal     _mcount
        .set    at
        move    $sp，$fp
        lw      $fp，20($sp)
        addiu   $sp，$sp，24
        j       $31
main:
```

```
addiu   $sp，$sp，-40
sw      $31，36($sp)
sw      $fp，32($sp)
move    $fp，$sp
.set    noat
move    $1，$31          # save current return address
subu    $sp，$sp，8      # _mcount pops 2 words from   stack
jal     _mcount
.set    at
jal     foo
move    $sp，$fp
lw      $31，36($sp)
lw      $fp，32($sp)
addiu   $sp，$sp，40
j       $31
```

可见，就 MIPS 而言，main()函数和 foo()函数的栈操作不一致，在 foo()函数中没有保存$31 寄存器，而 main()函数保存了。如果不使用-pg 编译，foo()是最后一个被调用的函数，而 main()不是，因此可以认为 foo()为叶子函数。所以，可以说，MIPS 上-pg 对于叶子函数和非叶子函数操作不一致。

对于 main()等非叶子函数，可以类似 X86 那样劫持栈内的函数返回地址，而对于叶子函数，则不然，因为叶子函数的返回地址根本没有保存在栈内，而是直接存放在$31（即 AT 寄存器中），因此可以通过劫持$31 寄存器来实现。但是在 mcount()中没有合适的办法来快速区分到底哪个函数是叶子函数，哪个函数是非叶子函数。

因为在 mcount()中通过栈保存了 31 号寄存器，因此可以考虑通过劫持该栈内的 31 号寄存器来实现。这样的话，就可以统一两者的 prepare_function_return()函数接口。因此，最关键的部分就是如何获取到叶子函数和非叶子函数存放返回地址（即 31 号寄存器）的栈内地址。对于叶子函数，因为相关栈的操作由 mcount()实现，因此很容易获取，而对于非叶子函数，有如下操作：

```
sw      $31，36($sp)
```

但实际上，它并不一定把 31 号寄存器保存在 sp+36 处，36 只是一个随机数，由编译器在编译过程中根据函数输入进行计算获得。因此，无法直接确定非叶子函数的栈内地址。不过，通过分析 main()和 foo()的汇编代码，对于它们两者来说，从 jal _mcount 到 move $fp，%sp 的位移是确定的，从 move $fp，$sp 指令开始往上查找，在 main()函数中找到 sw $31 寄存器之前都是 store 指令，而在 foo()中所有的 store 指令中都没有 sw $31 的操作。因此，可以通过一个简单的算法来区分叶子函数和非叶子函数：

通过 jal _mcount 调用点的位置(ip)，往上偏移一定的位置到 move $fp，%sp，然后往上查找 sw $31 指令，如果找到一条不是 store 的指令，那么直接返回，并认为该函数是叶子函数，否则一直查找，直到找到 sw $31 指令为止。

对于老的编译器，上述方法是唯一可行的，但是，可想而知，如果 main() 的栈很大，那么这种查找会产生较大开销。因此，最后考虑通过修改编译器中对 -pg 的实现。改进办法是设法让-pg 在插入 mcount 时额外传递一个参数，以告知非叶子函数的返回地址在栈内的偏移，对于非叶子函数，该偏移不可能为 0，因此对于叶子函数，可以把该值设置为 0 用于区分。改进之后，将很快就获取到了叶子函数和非叶子函数的返回地址存放在栈内的位置。

该改进最终由 David Daney 添加到了 gcc 4.5，它通过给 GCC 添加 -mmcount-ra-address 参数实现。为兼容早期 GCC 版本，本文作者对两种劫持函数返回地址的算法都予以了实现。

至于 mcount()的其他部分，跟静态 Function Tracer 和动态 Function Tracer 类似，因此不再详述，请查看内核源码。

实现完图形化 Function Tracer 就可添加 HAVE_FUNCTION_GRAPH_TRACER。

arch/mips/Kconfig:

```
config MIPS
    ...
    select HAVE_FUNCTION_GRAPH_TRACER
    ...
```

至此， Ftrace 的实现部分讨论完毕，下面通过实现效果验证实现是否成功。

4.1.4.4 实现效果

实现后需要通过使用效果检测实现是否成功。下面是 MIPS 平台上开启动态 Function Tracer 和图形化 Function Tracer 后的内核函数跟踪效果(只摘录一部分)。

```
# mkdir /debug
# mount –t debugfs nodev /debug
# echo function > /debug/tracing/current_tracer
# echo *timer* > /debug/tracing/set_ftrace_filter
# echo 1 > /debug/tracing/tracing_enabled
# sleep 1
# echo 0 > /debug/tracing/tracing_enabled
# cat /debug/tracing/trace | head -10
    # tracer: function
    #
```

```
    #              TASK-PID     CPU#      TIMESTAMP   FUNCTION
    #                 | |        |           |           |
                    bash-27175     [000]          9378.186588:      del_timer
<-flush_delayed_work
                    <idle>-0       [000]      9378.187244:   hrtimer_interrupt
<-c0_compare_interrupt
                    <idle>-0       [000]      9378.187247:   __run_hrtimer
<-hrtimer_interrupt
                    <idle>-0       [000]      9378.187251:   tick_sched_timer
<-__run_hrtimer
                    <idle>-0       [000]      9378.187253:   do_timer
<-tick_do_update_jiffies64
                    <idle>-0       [000]      9378.187261:   hrtimer_run_queues
<-update_process_times
# echo function_graph > /debug/tracing/current_tracer
# echo 1 > /debug/tracing/tracing_enabled
# sleep 1
# echo 0 > /debug/tracing/tracing_enabled
# cat /debug/tracing/trace | head -20
    # tracer: function_graph
    #
    # CPU    DURATION                  FUNCTION CALLS
    # |      |    |                    |   |   |   |
     0)               |  hrtimer_interrupt() {
     0)               |    __run_hrtimer() {
     0)               |      tick_sched_timer() {
     0)    5.723 us   |        do_timer();
     0)    1.518 us   |        hrtimer_run_queues();
     0)    1.552 us   |        run_posix_cpu_timers();
     0)    1.914 us   |        hrtimer_forward();
     0)  + 28.661 us  |      }
     0)    2.290 us   |      enqueue_hrtimer();
     0)  + 38.969 us  |    }
```

```
0)  + 67.588 us   |  }
0)                |  run_timer_softirq() {
0)   1.520 us     |    hrtimer_run_pending();
0)   5.186 us     |  }
0)   2.946 us     |  del_timer();
```

如上所示，动态 Function Tracer 能指定需要跟踪的函数，例如上面跟踪了所有函数名包含 timer 的内核函数，而图形化 Function Tracer 不仅能以树状形式显示出内核函数的调用关系，而且把各个函数的时间开销反应出来，并且对于开销较大的函数，比如 enqueue_hrtimer()，能以+38.969us 的形式突出显示出来，从而方便内核开发人员进行针对性的优化。

除单独使用外，Function Tracer 还能结合 Irqsoff， Preemptoff 等 Tracer 使用，例如结合 Irqsoff 使用的效果如下：

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 2.6.34-rc3
# --------------------------------------------------------------------
# latency: 166 us,  #92/92,  CPU#0 | (M:preempt VP:0,  KP:0,  SP:0 HP:0)
#      -----------------
#      | task: swapper-0 (uid:0 nice:0 policy:0 rt_prio:0)
#      -----------------
#   => started at: serio_interrupt
#   => ended at:   serio_interrupt
...
  <idle>-0        0d.h1.    0us+: serio_interrupt
  <idle>-0        0d.h2.    3us+: atkbd_interrupt <-serio_interrupt
  <idle>-0        0d.h2.    4us+: dev_get_drvdata <-atkbd_interrupt
  <idle>-0        0d.h2.    7us+: input_event <-atkbd_interrupt
  <idle>-0        0d.h3.    9us+: add_input_randomness <-input_event
  <idle>-0          0d.h3.      11us+:  add_timer_randomness
<-add_input_randomness
  <idle>-0          0d.h4.      13us+:  mix_pool_bytes_extract
<-add_timer_randomness
  <idle>-0        0d.h4.  22us+: credit_entropy_bits <-add_timer_randomness
  <idle>-0        0d.h5.  25us+: __wake_up <-credit_entropy_bits
```

<idle>-0	0d.h5.	27us+: kill_fasync <-credit_entropy_bits
<idle>-0	0d.h3.	30us+: input_handle_event <-input_event
<idle>-0	0d.h2.	38us : input_event <-atkbd_interrupt
<idle>-0	0d.h3.	39us : add_input_randomness <-input_event
<idle>-0	0d.h3.	40us : input_handle_event <-input_event
<idle>-0	0d.h3.	41us : atkbd_event <-input_handle_event
<idle>-0	0d.h3.	42us+: dev_get_drvdata <-atkbd_event
<idle>-0	0d.h3.	44us+: input_pass_event <-input_handle_event
<idle>-0	0d.h4.	48us+: kbd_event <-input_pass_event
<idle>-0	0d.h4.	50us+: __tasklet_schedule <-kbd_event

通过比较上述结果与4.1.3.4的Irqsoff跟踪结果（不使用Function Tracer）对比发现，结合Function Tracer后，Irqsoff Tracer能定位到引起相关延迟的热点函数(hotspot)，从而更好地辅助内核开发人员对实时内核进行优化。

4.2 实时抢占补丁的移植

这一节将介绍本章最重要的工作，即实时抢占补丁在特定平台上的移植。移植前先对其平台相关性进行分析。

4.2.1 平台相关性分析

从上一章可知，实时抢占补丁的实时改造技术包括自愿抢占、强制抢占、中断线程化、自旋锁转换为 Mutex(PI-Mutex)、高精度时钟、实时调度策略等几个方面。下面分析这几部分的平台相关性。

自愿抢占与平台不相关，在编写设备驱动时可以合理引入。

强制抢占与平台相关的部分在 arch/*/kernel/entry*.S 中实现，在中断返回恢复寄存器前，如果当前没有禁止抢占并且有其他任务需要抢占，那么调用 preempt_schedule_irq()抢占当前任务。具体实现可以参考 X86 和 MIPS。

中断线程化与平台相关的部分主要是硬中断的线程化。默认情况下，所有中断都会线程化，但是前面提到，有些中断不能被线程化，例如时钟中断、级联中断，以及其他处理例程很短的中断，如中断处理例程为 no_action()的中断，当然，还有一些系统出错中断，如 Bus Error。因此，需要了解目标平台相关中断的配置。

自旋锁转换为 Mutex 与平台相关的部分包括所有不能线程化的中断中持有的锁必须使用原有锁，当然，其他一些锁，比如用于保护硬件寄存器和出错信息的锁也不能被转换为 Mutex。

高精度时钟不仅要求目标平台有高精度的硬件时钟支持，也需要注册相应的 clocksource 和 clockevent 以提供底层的时钟获取和下一次时钟事件设置。

实时调度策略与平台相关的部分主要包括相关的系统调用。对于内核已经提供的调度策略 SCHED_FIFO 和 SCHED_RR，需要实现包括 sched_setscheduler()，sched_setparam()以及 sched_getparam()在内的系统调用。而对于还没有进入内核的 SCHED_DEADLINE，则主要包括 sched_setscheduler_ex()，sched_setparam_ex()和 sched_getparam_ex()。

如果要提供完整的实时支持，还需要提供相关的系统库，这些库也可能跟平台相关，请参考相关资料[80][81][82][83]。

4.2.2 龙芯平台上的实时抢占补丁移植实例

4.2.2.1 目标平台概述

福珑 2F 是一款由江苏龙芯梦兰科科技股份有限公司（Lemote）采用国产处理器龙芯 2F 生产的机器，该机器体型小巧，处理器具有中等性能（最高可稳定运行于 800MHz），并且频率可通过软件调节，功耗较低（10W 左右），提供外接接口丰富（包括硬盘、音频、视频、串口、USB、红外等），其中，有些型号采用无风扇设计，没有噪音。下面是该机器其中一个型号的照片：

图片4-1福珑 2F 迷你计算机

综合考虑福珑 2F 的这些硬件特性，它能潜在地应用于一些实时领域，比如它的体型小巧，占用空间少，功耗较低、节能，而性能中等，且主频可按需降低进而降低功耗，至于丰富的外部接口则可满足多种不用应用的需求，但目前还没有一款合适的实时系统，这正是本文移植实时抢占补丁到龙芯平台的原因之一。

到目前为止，福珑 2F 以及几乎所有龙芯产品都采用 Linux 操作系统，福珑 2F 的板极支持也已经在本文作者的推动下进入 Linux 官方 2.6.33，而最新的龙芯 Linux 由作者和另外一名自由软件工作者在以下站点共同维护：

http://dev.lemote.com/code/linux-loongson-community

开发实时抢占补丁时需要对福珑 2F的板极支持有充分的了解，可以参考内核源码以及Lemote公司提供的龙芯芯片手册和内核移植开发手册[70][71]。

下面结合福珑 2F 的资料讨论实时抢占补丁在其上的移植。

4.2.2.2 具体实现

1. 中断线程化

研究中断线程化前，需要了解福珑 2F的中断处理情况，可参考资料[69][70][71]的中断相关部分。

MIPS 处理器通过它的 Cause 寄存器识别中断，在该寄存器中有一组 8 个独立的中断位，可以识别 8 个中断，头两个是纯粹的软件中断，最后一个一般用于时钟中断，剩下的 5 个用来接外部中断。

MIPS 提供了一个高精度的内部时钟，通过 0 号协处理器的 count 和 compare 寄存器实现，当这两个寄存器中的值相等时触发时钟中断。龙芯跟 MIPS 完全类似，而外接中断与特定主版相关，比如在福珑 2F 上，有（IP 为 MIPS 为 Cause 寄存器中的中断位）：

IP7(内部时钟中断)
IP6(bonito 北桥，性能计数器中断)
IP5()
IP4()
IP3(处理器的 uart 串口)
IP2(8259A 南桥)
IP1(软件使用)
IP0(软件使用)

对于串口中断，完全可以线程化，而对于时钟中断，因为它本身用来驱动整个系统，因此应该赋予最高优先级，不能线程化，至于 IP6 和 IP2，用于级联来自北桥和南桥的中断，会被很快处理，因此可以线程化，但需要注意的是，IP6 被性能计数器中断和北桥中断共享，由于性能计数器的中断会很频繁带来很大开销，因此也应该被线程化，并且在实时操作系统中禁用（默认会禁用）。

另外，在南桥上，8259A 中断控制器还有一个级联中断，用于级联两片 8259A 以提供 15 个中断位，该中断应该禁止线程化。对于其他中断，都可以线程化。

下面对需要禁止线程化的中断通过 IRQF_NODELAY 进行标记。

- 时钟中断

MIPS 的内部时钟中断操作如下：

```
arch/mips/kernel/cevt-r4k.c:
struct irqaction c0_compare_irqaction = {
        .handler = c0_compare_interrupt,
        .flags = IRQF_DISABLED | IRQF_PERCPU | IRQF_TIMER | IRQF_NODELAY,
```

```
        .name = "timer",
};
```

不过，最新的实时抢占补丁有如下的宏定义：

```
#define IRQF_TIMER                (__IRQF_TIMER | IRQF_NODELAY)
```

而__IRQF_TIMER 同 IRQF_TIMER，因此，可以不外加 IRQF_NODELAY 标记。

不过，由于 IRQF_TIMER 在 Linux 2.6.30 才引入，内核中有些时钟中断可能还没有加上该标记，因此，移植实时抢占补丁时，需要确保添加了该标记。

在福珑 2F 上，还存在另外一个时钟，即 CS5536 MFGPT 提供的时钟，由于该时钟的时钟精度不高，因此在实时抢占补丁中将禁用，在讨论高精度时钟时会进一步讨论它，这里不再详述。

2. 级联中断

对于龙芯平台上的级联中断，通过如下方式非线程化：

arch/mips/loongson/common/irq.c：

```
struct irqaction ip6_irqaction = {
        .handler = ip6_action,
        .name = "cascade",
        .flags = IRQF_SHARED | IRQF_NODELAY,
};
struct irqaction cascade_irqaction = {
        .handler = no_action,
        .name = "cascade",
        .flags = IRQF_NODELAY,
};
```

对于 8259A 的级联中断：

arch/mips/kernel/i8259.c:

```
static struct irqaction irq2 = {
        .handler = no_action,
        .flags = IRQF_NODELAY,
        .name = "cascade",
};
```

对于 8259A，很多操作都需要自旋锁保护，由于在非线程化的中断中自旋锁必须失效抢占，防止死锁，因此所有 8259A 用到的自旋锁必须正确定义并使用对应的操作函数。最新内核中的这些锁可以通过 DEFINE_RAW_SPINLOCK 定义，而相应的操作都有 raw_前缀，例如：

```
~~DEFINE_SPINLOCK(i8259A_lock);~~
DEFINE_RAW_SPINLOCK(i8259A_lock);


~~spin_lock_irqsave(&i8259A_lock, flags);~~
raw_spin_lock_irqsave(&i8259A_lock, flags);
```

3. 高精度时钟

上一节提到，MIPS 上的 0 号协处理器提供了 count 和 compare 寄存器，用于实现内部时钟。通过访问 count 计数器，可以获取当前的系统时间。

如果要实现时钟中断，其计时原理如下：

假如想设置一个周期性中断，中断时间间隔为 delta，那么先读出 count 计数器的值，加上 delta 后写入 compare 计数器，count 计数器会自增，当 count 计数器的值增加到 compare 时会触发一个时钟中断。

龙芯上的该计数器的计数频率是处理器主频的一半，通过4.1.3已知，它的分辨率可以达到 2.5ns，满足一般实时应用的需求。而上一章已介绍了MIPS平台上的底层clocksource和clockevent支持，为整个系统提供了一款高精度时钟。

需要注意的是，由于该计数器的计数频率跟处理器主频关联，而龙芯支持动态变频，由于在内核中启用动态变频后，处理器主频可能随着系统负载自动调节，因此会使得该计数器的计数紊乱，严重影响整个系统的实时能力。所以，在实时抢占补丁中，需要禁用动态变频。

arch/mips/Kconfig:

```
if !PREEMPT_RT
...
source "drivers/cpufreq/Kconfig"
...
Endmenu
```

禁用内核中的动态变频后，内核开发人员还是可以在内核初始化时为龙芯设置一个固定频率以满足特定应用的需求。只要没有启用内核中的动态变频模块，之后的处理器主频不变，计数器计数频率就会稳定，但是频率不能设置得太低，否则会降低计数器的计数精度。

在福珑 2F 上，还有另外一款时钟可以使用，那就是南桥 CS5536 上的多功能时钟(MFGPT)，但是那款时钟的频率最高为 14318000，即 1/14318000 s=69.8 ns，精度差了几十倍，并且，该时钟的 oneshot 模式工作不正常，因而不适合作为高精度时钟源。为了防止用户错误选择时钟源，该时钟在实时抢占补丁中被禁用。

arch/mips/loongson/Kconfig:

```
config CS5536_MFGPT
        bool "Using cs5536's MFGPT as system clock"
```

```
depends on CS5536 && !PREEMPT_RT
```

4. 增加抢占点

信号处理函数 do_signal()默认关闭中断并禁止了抢占。实际上，这里可以明确引入调度点，允许抢占，从而减少调度器延迟：

arch/mips/kernel/signal.c：

```
static void do_signal(struct pt_regs *regs)
{
    ...
#ifdef CONFIG_PREEMPT_RT
        /*
         * Fully-preemptible kernel does not need interrupts disabled:
         */
        local_irq_enable();
        preempt_check_resched();
#endif
    ...
}
```

另外，编写驱动时，也应该合理地引入自愿抢占，以提高系统响应性能。

5. 添加 SCHED_DEADLINE 实时调度策略

前面提到，当前内核和实时抢占补丁中都没有一款调度策略完全满足有截止时间要求的任务。不过，其他一些团队已经做了相关工作，即 SCHED_DEADLINE，下面讨论如何把该调度策略移植到 MIPS 平台。由于作者在进行该研究时，SCHED_DEADLINE 只支持 2.6.31，因此，这里仅以 rt/2.6.31 为例来介绍。

- 添加 MIPS 平台上的系统调用接口

由于原调度策略中用到的系统调用无法满足 SCHED_DEADLINE 调度策略的要求，于是 SCHED_DEADLINE 作者添加了四个新的系统调度：

sched_setscheduler_ex	改变一个任务的调度策略和优先级
sched_setparam_ex	设置 SCHED_DEADLINE 任务的参数
sched_getparam_ex	获取任务的参数
sched_wait_interval	根据调度类型睡眠

这四个系统调用都在 kernel/sched.c 中实现，但是要求添加平台相关的系统调用表中，并在内核提供给用户空间的 arch/mips/include/asm/unistd.h 头文件中提供各个系统调用在对应的系统调用表中的偏移。

由于 MIPS 支持多种不同 ABI，内核为了兼容来自用户空间不同 ABI 的系统调

用请求，需要提供不同版本系统调用。这里仅考虑内核 ABI 为 64，用户态 ABI 为 o32：

arch/mips/kernel/scall64-o32.S:

```
    PTR     sys_accept4
+       PTR     sys_sched_setscheduler_ex       /* 4335 */
+       PTR     sys_sched_setparam_ex
+       PTR     sys_sched_getparam_ex
+       PTR     sys_sched_wait_interval
```

arch/mips/include/asm/unistd.h:

```
#if _MIPS_SIM == _MIPS_SIM_ABI32
#define __NR_accept4                    (__NR_Linux + 334)
+#define __NR_sched_setscheduler_ex     (__NR_Linux + 335)
+#define __NR_sched_setparam_ex         (__NR_Linux + 336)
+#define __NR_sched_getparam_ex         (__NR_Linux + 337)
+#define __NR_sched_wait_interval       (__NR_Linux + 338)

 /*
  * Offset of the last Linux o32 flavoured syscall
  */
-#define __NR_Linux_syscalls             334
+#define __NR_Linux_syscalls             338
#endif /* _MIPS_SIM == _MIPS_SIM_ABI32 */
```

添加系统调用后就可在用户空间使用 SCHED_DEADLINE 调度任务。

- 修改 schedtool 以便可以调整任务的调度策略为 SCHED_DEADLINE

Ingo Molar 编写了 schedtool，该工具可以改变任务的调度策略，但是默认不支持 SCHED_DEADLINE，所以 SCHED_DEADLINE 的作者对其进行了修改。但是，即使是修改过后的 schedtool 也无法直接在 MIPS 上使用，因为还需要加入平台相关的系统调用偏移地址的定义并考虑不同 ABI 数据传输的兼容性。

通过下面的修改后，schedtool 才可以在 MIPS 正常工作：

首先，把上面 unistd.h 中的内容添加到 edf_syscall.h 中（仅考虑 o32 ABI）：

edf_syscall.h:

```
#define __NR_sched_setscheduler_ex      365
 #endif
```

```
+#ifdef __mips__
+/* XXX use the proper syscall number */
+#if _MIPS_SIM == _MIPS_SIM_ABI32
+#define __NR_Linux                    4000
+#define __NR_sched_setscheduler_ex    (__NR_Linux + 335)
+#endif
+#endif
+
+#endif /* __mips__ */
```

接着考虑 ABI 兼容性问题，由于内核采用的是 64 位的 ABI（仅考虑 64 位），而用户空间是 o32 的 ABI（仅以 o32 为例），因此数据类型不一致，所以在通过 sys_sched_setscheduler_ex() 给内核传递参数时会出错，其中发现 sys_sched_setscheduler_ex()从用户态传递了一个结构体 timespec 给内核空间，在内核空间，其数据类型如下：

```
include/linux/coda.h:
struct timespec {
        long        ts_sec;
        long        ts_nsec;
};
```

对于 64 位内核，上面两个成员都是 64 位，但是 o32 ABI 的系统库中的数据类型定义如下：

```
struct timespec
 {
   __time_t tv_sec;
   long int tv_nsec;
 }
```

在o32 ABI下，tv_nsec的数据类型也是 64 位，但是__time_t的定义却为long，只有 32 位[11]，因此，内核和用户态的数据类型不一致，需要调整该定义。解决办法是在sched_tool.c的开头加入如下内容：

```
#ifdef __mips__
#define __timespec_defined
```

[11] 对于最新的 libc，可能有更新

```
struct timespec
  {
    long long tv_sec;          /* Seconds.   */
    long long tv_nsec;         /* Nanoseconds.   */
  };
#endif
#endif
```

其中#define __timespec_defined 确保 gcc 在编译时不出现重定义错误，定义该宏后将自动取消 libc 中的该定义。解决以上两个问题后就可以通过 schedtool 设置任务的调度策略为 SCHED_DEADLINE，大概用法如下：

```
$ ./schedtool -E -d 30000 -b 10000 -g 50000 -p0 -e yes
-E 使用 SCHED_DEADLINE 调度策略
-d 指定任务的截止时间
-b 指定任务的运行时间
-g 指定任务的周期
-p 指定任务的优先级（SCHED_DEADLINE 任务的优先级必须指定为 0，实际优先级通过最早截止时间确定）
-e 指定要运行的任务程序
```

采用该调度策略的任务在单处理器系统上必须满足如下关系：

```
period >= deadline > runtime
```

- 修订该调度策略实现中的一个 bug

虽然通过上述工作，SCHED_DEADLINE 可以在 MIPS 工作，但是发现，通过 schedtool 设置任务的调度为 SCHED_DEADLINE 后，再次使用 schedtool 时，schedtool 无法运行也无法通过外部程序结束，而此时整个系统还正常工作，因此系统可能出现某种死循环导致 schedtool 任务无法退出。最后，通过分析 dmesg 命令输出的内核日志发现 sirq-hrtimer 这个软件中断出现异常，经过调试分析发现问题出现的场景：当用户试图通过 kill 或者 CTRL+C 终止 schedtool 时，系统就会出现异常。通过后续分析发现确实是系统出现了某种死锁。

在 kernel/sched_deadline.c 的 deadline_timer()中使用了 atomic_spin_lock 和 atomic_spin_unlock，在 atomic_spin_unlock()中允许了抢占，并进入了 schedule()，如果此时用户试图终止 schedtool，那么 schedtool 将进入 DEAD 状态，并会切换到其他任务，由于该 SCHED_DEADLINE 的实现把 hrtimer 放在各个任务自身的结构体中，正文切换时需要释放前一个 DEAD 任务的资源，包括 hrtimer，因此试图调用 hrtimer_cancel()取消该任务的 hrtimer，但是此时并没有从 deadline_timer()函数中返回，所以无法释放 hrtimer，造成死循环：

```
kernel/hrtimer.c:
```

```
int hrtimer_cancel(struct hrtimer *timer)
{
        for (;;) {
                int ret = hrtimer_try_to_cancel(timer);
                if (ret >= 0)
                        return ret;
                hrtimer_wait_for_timer(timer);
        }
}
```

可行的解决办法是在 deadline_timer()中使用 atomic_spin_lock_bh()和 atomic_spin_unlock_bh()禁止抢占：

kernel/sched_deadline.c:

```
static enum hrtimer_restart deadline_timer(struct hrtimer *timer)
        __acquires(rq->lock)
{
    ...
-   atomic_spin_lock(&rq->lock);
+   atomic_spin_lock_bh(&rq->lock);
    ...
-   atomic_spin_unlock(&rq->lock);
+   atomic_spin_unlock_bh(&rq->lock);
    return HRTIMER_NORESTART;
}
```

或者类似其他调度策略，把 hrtimer 成员从单个任务的结构体中移到整个调度任务的 dl_bandwidth 结构体中，并在撤消调度任务组的资源时释放该资源，而无须在正文切换时撤销，从而避免上述问题。

include/linux/sched.h:

```
struct sched_dl_entity {
        ...
-   struct hrtimer    dl_timer;
};
struct dl_bandwidth {
```

```
    ...
+   struct hrtimer    dl_timer;
};
+static void destroy_dl_bandwidth(struct dl_bandwidth *dl_b)
+{
+           hrtimer_cancel(&dl_b->dl_timer);
+}
static void free_deadline_sched_group(struct task_group *tg)
{
+   destroy_dl_bandwidth(tg);
          kfree(tg->dl_rq);
}
```

当然，相应的 dl_timer 的操作也得调整。

目前采用了第一个解决办法，见源码中的该补丁：SCHED_DEADLINE: temp fixup of "BUG: scheduling while atomic"。不过第二个解决办法更好，因为第一个办法在 deadline_timer()中禁止了抢占，并且 hrtimer_cancel()在正文切换中完成，会增加正文切换开销。

上述相关源码在 http://dev.lemote.com/rt4ls 的 rt/2.6.31/loongson 分支中，而相应的 schedtool 则在 rt/schedtool-deadline 分支中。

6. 其他

- 使用原始自旋锁

默认情况下，实时抢占补丁把所有自旋锁都转换为 Mutex，但是跟非线程化中断中的 i8259A_lock 锁用法类似，有些锁不能转换为 Mutex，例如，die()函数中用到了一个 die_lock，因为 die()函数比较特殊，此时系统可能出现了严重的错误，因此，die_lock 保护的操作不能够被抢占。

- 有些 schedule()需要替换为__schedule()

由于早期的 schedule()在实时抢占补丁中被抽象成了__schedule()，如果有些地方本身关闭了中断，并且进行了 need_sched 检查，可以直接调用__schedule()，例如 arch/mips/kernel/entry.S 中的 work_resched()和 arch/mips/kernel/process.c 中的 cpu_idle()。

至此，实时抢占补丁的移植工作讨论完了，关于相关成果是否达到了要求，将在进一步优化后，进行性能评测和分析。

第5章实时抢占补丁优化

本章讨论了实时抢占补丁的优化方法，首先分析了实时操作系统优化的一般方法与原理，之后以龙芯处理器平台为例介绍了实时抢占补丁的优化过程。

5.1 实时操作系统优化概述

实时操作系统优化的主要目标是消除系统中各种不确定性因素，让系统服务响应时间更加确定。

对于特定的硬件平台，最好情况下的延迟（即采样结果中的最小值）基本上是确定的（主要决定于各项硬件性能指标），因此优化的目标是尽可能地减少最坏情况下的延迟，让它们趋向最好情况下的延迟，从而让整个系统服务响应时间更加确定。例如，在优化前的最坏情况下中断延迟为 200us，最好情况下中断延迟为 50us，如果能够把延迟时间优化到 100us，那么中断延迟时间的 Jitter 将减少 100us，整个系统服务响应时间将变得更加确定。

因此，确切地说，实时操作系统优化的目标是提升各种实时操作系统的性能指标，减少中断延迟、调度器延迟、任务调度时间、提高时钟系统的精确性，减少各种同步、互斥和通信机制的延迟等，从而让整个系统获得更好的响应能力。那么如何进行相应的优化呢？

5.1.1 通用优化方法

有些通用优化方法，不仅能提升处理器和内存利用率，增加系统吞吐量，也能减少任务执行时间，提升系统服务响应能力。

例如，基于时间复杂度的算法优化可以减少算法的时间开销，基于编译器的指令优化能够利用某些处理器特有的优化指令减少某些指令的执行时间，而基于某些硬件特性（比如性能计数器、Cache、TLB 管理、跳转指令）的性能优化，能够消除一些执行时间较长的路径，减少不必要的 Cache 和 TLB 失效等，例如有些平台通过合并内核、模块和进程的地址空间，减少长跳转指令的使用，或者利用 VDSO 方法，消除系统调用减少系统调用开销。还有一些方法试图优化系统中调用较多的函数，诸如锁、原子操作、基本数学函数库等。

关于更多通用的优化方法请参考资料[34][35]。

5.1.2 实时操作系统特定的优化方法

不过，通用优化方法更多地关注处理器和内存利用率，而实时系统特定的优化方法则是为了减少各种系统响应延迟，减少引起延迟的的中断关闭、抢占禁用等区域。这方面的优化方法，可以参考资料[36][37][38][39]。

需要注意的是，资料[38]说明，优化过程需要兼顾各个性能指标，有些优化

动作虽然能够提升其中一项性能指标，但会增加整个系统响应延迟。另外，资料[36][37] [39]说明，为了提升整个系统的实时性能，单纯优化操作系统本身的性能指标是不够的，包括模块，应用程序架构，各种协议栈等都需要统筹兼顾。

5.2 实时抢占补丁优化

下面在总结实时抢占补丁各种优化方法后，以龙芯平台为例进行实例分析。

5.2.1 实时抢占补丁优化方法

资料[36][37][38]介绍了优化实时抢占补丁的各种方法，总结如下：

1. 选择合适的编译器和编译器选项

如果有专门针对目标平台优化过的编译器可用，那么可以考虑选择这样的编译器。而有些编译器支持特定处理器的优化选项，如 gcc 的-march 选项，另外，在不影响内核正常工作的情况下，可以开启 gcc 的最大优化选项，比如-O3。

2. 调整内核配置选项

关闭所有无关的内核配置选项，例如，如果不用 USB 设备，就可以关掉相关内核支持，消除相关驱动中潜在的延迟影响；关掉一些 SMI（系统管理中断）驱动，例如，有些平台利用 SMI 监控 CPU 温度，这些中断可能比较频繁或处理比较耗时，抑或是根本不受内核控制（在 BIOS 中处理），会带来不可预测的延迟，但是停用相关驱动就不会使能这些中断；开启所有实时抢占补丁必须的内核选项，比如硬中断和软中断的线程化、高精度时钟、实时抢占 RCU、设置较高的 HZ 等；如果系统内存够用，可以考虑完全关闭 SWAP 支持；关闭动态变频等功耗管理支持，动态变频会根据 CPU 利用率动态调节 CPU 主频，让实时任务的执行时间变得不可预测；如果内核支持 VDSO，开启相关支持可减少系统调用开销；评测各种其他选项对系统实时性能的影响，找出最好的组合。

3. 合理配置硬件

PCI 等外部总线控制器提供了专门的寄存器用于配置一些参数，可以调节和测试这些参数的组合对整个系统性能的影响，从而找出最佳的组合。

4. hotspot 优化

利用 Ftrace，Perf 等工具找出系统中引起延迟并且执行时间较长的热点区域，进行专门的优化。比如减少或者消除关闭中断与禁止抢占的区域，在长代码块中明确引入自愿抢占，改进算法以减少时间复杂度，消除分支语句以减少分支预测失败等，从而有效提升相关路径执行效率，提高系统响应能力。

5. 具体应用时的优化

除了在实时应用开发过程中遵循相应实时标准（比如通过mlockall预留内存）[25][26]外，还可以调节那些存在潜在竞争的中断处理线程和某些实时任务的cpu affinity，让它们都有足够的处理器资源可用。同样地，如果硬件和内核支持，也可以采用Cache预留和TLB预留。

6. 数据结构与算法的改进

比如，在有些互斥机制中采用合适的锁，例如在读较多的数据结构中，可以用高效的 RCU 锁，而不是 reader-writer 锁，甚至在有些地方都可以用更高效的 lock-free 机制。另外，对于那些非 O(1)的算法，可以尽量优化成 O(1)，从而使得它们的执行时间可确定。

5.2.2 龙芯平台上的实时抢占补丁优化实例

下面以龙芯平台为例，讨论实时抢占补丁的优化情况。

1. 减少条件分支预测失败：添加 cpu-feature-overrides.h 文件

由于 MIPS 变体繁多，不同变体具有不同特性。为支持各种不同变体，Linux 根据它们的特性提供相应的代码支持，例如原子操作：

arch/mips/include/asm/atomic.h:

```
static __inline__ void atomic_add(int i,  atomic_t * v)
{
        ...
            } else if (kernel_uses_llsc) {
                int temp;
                __asm__ __volatile__(
                "       .set    mips3                                   \n"
                "1:     ll      %0,  %1         # atomic_add            \n"
                "       addu    %0,  %2                                 \n"
                "       sc      %0,  %1                                 \n"
                "       beqz    %0,  2f                                 \n"
                "       .subsection 2                                   \n"
                "2:     b       1b                                      \n"
                "       .previous                                       \n"
                "       .set    mips0                                   \n"
                : "=&r" (temp),  "=m" (v->counter)
                : "Ir" (i),  "m" (v->counter));
        } else {
                unsigned long flags;
                raw_local_irq_save(flags);
                v->counter += i;
```

```
                raw_local_irq_restore(flags);
        }
}
```

arch/mips/include/asm/cpu-features.h:

```
#ifndef cpu_has_llsc
#define cpu_has_llsc                (cpu_data[0].options & MIPS_CPU_LLSC)
#endif
#ifndef kernel_uses_llsc
#define kernel_uses_llsc            cpu_has_llsc
#endif
```

可以看出，如果没有定义 cpu_has_llsc，将通过(cpu_data[0].options & MIPS_CPU_LLSC)进行条件判断。在没有初始化 cpu_data[0]前，将执行上述原子操作 atomic_add()的 C 语言路径，而初始化后，如果处理器提供了 *ll* 和 *sc* 指令，将执行汇编语言那条路径。而且即使初始化后，也会进行多余的条件判断，产生潜在的分支预测失败，带来不可预测的延迟。

如果定义了相关宏：

```
#define cpu_has_llsc                1
```

那么，编译时将只保留上述原子操作 atomic_add()的汇编指令部分，不仅减少了内核映像文件大小，消除了多余的条件分支从而减少潜在的分支预测失败导致的延迟，提升系统响应能力。

龙芯 Linux 内核刚开始并没有 cpu-feature-overrides.h 文件，因此本文作者添加了 arch/mips/include/asm/mach-loongson/cpu-feature-overrides.h，并在其中定义了各种宏以描述龙芯处理器的特性。

在 arch/mips/kernel 下，相关宏的使用多达 100 多处，定义这些宏后将减少 100 多个条件分支，并且使得内核映像文件减少 200 多 KB。而由于这些原子操作被频繁调用，因此，条件分支的消除将提升整个系统实时性能。

2. 加速中断分派过程

下面是龙芯平台的中断分派过程：

arch/mips/loongson/lemote-2f/irq.c:

```
void mach_irq_dispatch(unsigned int pending)
{
        if (pending & CAUSEF_IP7)
                do_IRQ(LOONGSON_TIMER_IRQ);
        else if (pending & CAUSEF_IP6) { /* North Bridge,  Perf counter*/
```

```
                do_IRQ(LOONGSON2_PERFCNT_IRQ);
                bonito_irqdispatch();
        } else if (pending & CAUSEF_IP3) /* CPU UART */
                do_IRQ(LOONGSON_UART_IRQ);
        else if (pending & CAUSEF_IP2)   /* South Bridge */
                i8259_irqdispatch();
        else
                spurious_interrupt();
}
```

由于龙芯 2F 的性能计数器与北桥共享一个中断线，因此，在没有开启 OPROFILE 支持时，应该完全禁止对 do_IRQ(LOONGSON2_PERFCNT_IRQ)的调用。

```
        else if (pending & CAUSEF_IP6) {            /* North Bridge,  Perf counter */
#ifdef CONFIG_OPROFILE
                do_IRQ(LOONGSON2_PERFCNT_IRQ);
#endif
                bonito_irqdispatch();
        }
```

由于串口一般只用于调试，可以把对串口的处理移到最后，而让来自南桥的中断得到优先处理。

```
                bonito_irqdispatch();
        } else if (pending & CAUSEF_IP2)            /* South Bridge */
                i8259_irqdispatch();
        else if (pending & CAUSEF_IP3)   /* CPU UART */
                do_IRQ(LOONGSON_UART_IRQ);
```

由于调用 mach_irq_dispatch()的函数 plat_irq_dispatch()非常小，可以考虑把 mach_irq_dispatch()声明为 inline 函数，让 gcc 在编译时把其直接嵌入到 plat_irq_dispatch()中，从而减少不必要的函数调用开销。

```
inline void mach_irq_dispatch(unsigned int pending)
```

3. 关闭一些无关驱动

龙芯上的几个平台驱动用于管理处理器温度、风扇转速、显示器亮度等，由于这些中断可能很频繁，而且相应的中断处理例程开销很大，在确保处理器温度和风扇转速正常的情况下，可以考虑关闭这些平台驱动。如果目标平台只需要网络支持，可以关闭显卡驱动，如果不需要网络支持，可以把停用网卡驱动，而一些其他驱动，比如 USB 等，如果不用也可以停用从而避免带来潜在的延迟。

4. 通过 Ftrace 优化

在开启 Irqoff 和 Preemptoff Tracer 时，发现测试结果很大，而且都在 cpu_idle()中。通过比较 X86 等其他平台的实现后发现，当 CPU 利用率很低时，大部分执行时间将耗费在 cpu_idle()里头，所以 Irqsoff 和 Preemptoff 的跟踪结果较大。由于执行 cpu_idle()意味着此时处理器处于空闲状态，追踪的结果对于实时优化没有意义，因此需要关闭对 cpu_idle()函数的跟踪。

arch/mips/kernel/process.c:

```
/* Don't trace irqs off for idle */
stop_critical_timings();
if (cpu_wait) {
    (*cpu_wait)();
}
start_critical_timings();
```

5. 优化信号处理过程

David Daney最近提供了一个内核补丁用于实现MIPS下的vdso[40][41]，不过目前只支持信号处理，它通过把两个信号处理相关的系统调用提前从内核空间映射到了进程空间，减少了信号延迟。该补丁提交给 2.6.34 内核，不过，本文作者已经把它移植到rt/2.6.33 内核。

6. 使能龙芯特定的 gcc 优化选项

GCC 4.4 以及后续版本提供了龙芯处理器的特定优化选项，因此，编译内核时强烈建议使用相应的 GCC 版本并开启该特定优化选项。

arch/mips/Makefile:

```
cflags-$(CONFIG_CPU_LOONGSON2F) += \
        $(call cc-option，-march=loongson2f，-march=r4600)
```

7. 开启-O3 优化

-O3 比-O2 提供了更进一步的优化，而内核默认采用了-O2 选项，因此可以替换为-O3。为确保使用-O3，配置内核时还应关闭 CONFIG_CC_OPTIMIZE_FOR_SIZE 选项。

Makefile:

```
~~KBUILD_CFLAGS   += -O2~~
KBUILD_CFLAGS   += -O3
```

8. 关闭或重定向内核日志输出

默认情况下内核会把各种日志打印到日志缓冲区，并通过 syslogd 等 daemon 进程把日志记录到磁盘上，如果这种记录比较频繁，会产生较大开销，因此，可以考虑关闭 syslogd 的输出或者通过网络输出到另外一台机器上。

9. 把模块编译进内核

默认情况下，MIPS 的模块与内核的地址空间不一致，如果模块要调用内核中的函数，那么需要长跳转，而长跳转与短跳转相比，不仅指令数增多，而且性能明显不如短跳转。

10. 网络优化

对于网络方面的优化，由于 TCP 协议为了更好地控制有效性和拥塞，会带来较大延迟，因此，如果不需要 TCP 协议，可以考虑采用其他协议，比如 UDP，即使不关闭，在程序设计时亦可考虑使用 TCP_NODELAY 等来减少延迟，另外，netstat 和 ethtool 等工具可以用于监控和调节网络配置，比如 ethtool 的-C 选项通过收集多个包然后发一次中断可以有效减少中断次数，而-A 能够通过减少 I/O switches 能有效地减少冲突。

11. 内核配置

开启实时抢占需要的各种配置	CONFIG_PREEMPT_RT=y CONFIG_PREEMPT=y CONFIG_PREEMPT_SOFTIRQS=y CONFIG_PREEMPT_HARDIRQS=y CONFIG_TREE_PREEMPT_RCU=y
使能高精度时钟	CONFIG_HIGH_RES_TIMERS=y CONFIG_CEVT_R4K_LIB=y CONFIG_CEVT_R4K=y CONFIG_CSRC_R4K_LIB=y CONFIG_CSRC_R4K=y
动态 tick 以及较高的 HZ	CONFIG_NO_HZ=y CONFIG_HZ_1000=y
I/O 调度策略	CONFIG_IOSCHED_DEADLINE=y CONFIG_DEFAULT_DEADLINE=y
关闭针对映像文件大小的优化，从而允许开启-O3	# CONFIG_CC_OPTIMIZE_FOR_SIZE is not set
仅仅开启EXT2 文件系统支持，不使用日志文件系统，见资料[36]	CONFIG_EXT2_FS=y
使用更简单的内存分配器，见资料[42][43][44]	CONFIG_SLOB=y
使用 64 位而不是 32 位	CONFIG_64BIT=y

第6章 性能测试与分析

该章对前面移植与优化的实时抢占补丁进行性能评测与分析。首先介绍性能评测的内容、环境、方法与工具，并提出评测时需要注意的事项，随后给出每个性能指标的评测原理与测试结果并对结果进行分析。最后对另外一个平台（威盛）上的实时抢占补丁进行了性能测试并与龙芯平台进行了横向比较。

6.1 评测内容

首先评测 Ftrace 中用于获取时间戳的 sched_clock()函数的开销，接着评测实时操作系统的各项实时性能指标，有中断延迟、调度延迟、正文切换延迟、时钟精确性、基于优先级的各种实时调度策略的正确性、同步、互斥和通信的延迟、是否支持优先级继承等。不仅要反映它们在最坏情况下（有负载）的时间，还需要反映 Jitter、平均性能和标准偏差。为了反映负载对实时性能的影响，还将在无负载的情况下测试这些性能指标，以便进行比较与分析。

6.2 评测环境

进行数据采样时需要构建一个最坏的运行环境，即尽可能地增加处理器、内存和 I/O 的负载。可以通过产生大量外部事件，启动大量应用程序等来实现。

下表是本文的测试环境：

机器	龙梦(Lemote) 福珑 6002	
处理器	Loongson-2F 797M	
内存	DDR2 512M	
发行版	Debian GNU/Linux squeeze/sid	
内核	2.6.33.1-rt11 (http://dev.lemote.com/code/rt4ls rt/2.6.33/loongson) 注：SCHED_DEADLINE 调度策略采用的内核是 rt/2.6.31/loongson	
内核配置	arch/mips/configs/lemote2f_rt_defconfig	
Glibc	2.10.2	
系统负载	无负载	默认启动，没有 X windows，没有额外负载
	有负载	默认启动，没有 X windows，通过后文的 load.sh 产生负载

6.3 评测方法和工具

测试实时系统性能的方法与工具很多，资料[24]列举了大量可用工具，资

料[1][20][23]介绍了各种常见实时性能指标的测试方法。但这些评测方法与工具都有一定局限性。例如，为了达到较高测试精度，有些方法需要构建复杂的软、硬件环境；有些集成测试工具可能是商业产品，无法获取；虽然有大量开放源码工具可用，但这些工具测试的内容不够全面，或者依赖目标系统硬件环境，或者依赖实时系统提供的API，无法在其他系统中使用。

因此，在评测特定的实时系统时，现有的工具与方法可能不能直接重用，需要进行一定改造。本文测试时采用的工具基于以下工具改造：

- Real time testcases of ltp(Linux Test Project)

一个开放源码的测试套件，基于 GPL 协议，用于测试实时 Linux，并由 IBM 公司的实时团队维护。之前该测试套件在 http://rt.wiki.kernel.org 维护，目前已经合并到 Linux 测试项目(http://ltp.sourceforge.net/)中，存放在 testcases/realtime。

- rt-tests

rt-tests 最初由 Thomas Gleixner 编写，用于测试其开发的 Linux 高精度时钟子系统，后来由 Clark Williams 维护，并不断集成了来自更多开源团体和个人贡献的实时测试组件。该套件主要贡献来自 Redhat 和 OSADL，其项目仓库如下：git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

- 其他工具

工具名	简介
Chrt	设置任务（包括进程、线程）的优先级和调度策略。
Taskset	指定任务到某个线程到某个处理器执行（用于多核）。
Ping	用于产生大量的网络负载。
Find	用于产生磁盘负载。
hackbench	产生内存和处理器负载，hackbench 在 rt-tests 中提供。
Gnuplot	用于对结果进行可视化分析。

由于上述工具测试内容不够全面，本文作者基于它们编写了额外的驱动、自动化脚本和采样数据的图形化分析工具，整理了一个自动化测试工具集。这些驱动和工具都提交到了 git://dev.lemote.com/rt4ls.git 的 rt/2.6.33/loongson 分支下。

相关模块存放在 drivers/platform/rt/目录下，该文用到的模块包括：

模块	配置	简介
sched_clock_overhead.ko	SCHED_CLOCK_OVERHEAD=m	测试 sched_clock 开销
interrupt_latency.ko	INTR_LAT=m	测试中断延迟、调度延迟

相关工具存放在 tools/rt/目录下，主要包括：

目录	简介
rt-tests/	即上文提到的 rt-tests，为方便统计采样数据，部分工具有修改。后文用到其中的 cyclictest, hackbench, ptsematest, signaltest 和 svsematest。

scripts/	auto-test.sh 用于在无负载下自动测试本文涉及到的部分测试内容；auto-test-100load.sh 与 auto-test.sh 类似，但测试时需要执行 load.sh 添加系统负载；gramp-samples.sh 用于对采样结果进行绘图。
more/	本文作者编写的几个测试工具，包括测试中断延迟与调度延迟的 interrupt_latency（需要配套上面提到的 interrupt_latency.ko 模块），测试时钟精度的 rtclock，测试正文切换延迟的 rtcontext-switch 等。
results/	包括后文的所有测试结果，以及相应的 gnuplot 脚本。

需要说明的是，为了方便比较，下文所有绘图结果并不是由 graph-samples.sh 直接产生，而是先由 graph-samples.sh 的-S 参数得到 gnuplot 脚本，然后把脚本手工合并到 results/目录下的*.gnuplot 脚本中，然后再绘图。另外，下文中所有以 tools/rt/开头的路径都是指这里的 tools/rt 目录。

6.4注意事项

为确保测试的正确性，需要注意以下事项：

1. 所有测试用例应该遵循特定的实时开发标准

测试用例本身要遵循实时应用的开发标准，可参考资料[25][26]。例如，如果没有利用支持优先级反转的互斥机制，那么优先级反转的测试就会失败。同样地，如果没有采用高精度时钟系统提供的API，那么各种结果的测试精度很差，不能准确反应系统的真实性能。

2. 对于同步、互斥和通信的测试

由于多个任务的初始化顺序不一样，第一个测试结果可能包含另外一个任务的初始化延迟，因此，测试时需通过同步措施确保所有任务都初始化。

3. 实时任务需要配置合适的优先级

可以通过 chrt 工具来配置，也可以在代码中通过 sched_setscheduler()或者 pthread_getschedparam()函数来设置。实时任务的优先级应比内核线程的优先级高，至少在 50 以上，比如 90。

4. 必须采样足够样本

因为测试的随机性，采样数据越多越能够模拟真实的环境。

5. 测试结果不能完全反应真实情况

由于无法完全模拟真实环境以及采样数据的有限性，因此，测试结果不能完全反应真实情况，所以测试数据只能提供参考，不能作为最终是否采用该产品的依据。

6.5 测试原理、结果与分析

6.5.1 sched_clock()开销

该节测试本文移植的两种不同版本的 sched_clock()函数的开销，并与getnstimeofday()函数的开销进行比较。

测试时需要用到本文作者编写的 sched_clock_overhead 驱动，该驱动为rt/2.6.33/loongson 源码中的 drivers/platform/rt/sched_clock_overhead.c。编译内核时需要开启 SCHED_CLOCK_OVERHEAD=m 选项。

默认情况下会使用基于 cnt32_to_63()的版本，如果要测试基于累加算法的版本，需要在 arch/mips/kernel/csrc-r4k.c 中删除该行：

```
#define CNT32_TO_63
```

测试步骤：

```
// 进入模块存放目录
$ insmod sched_clock_overhead.ko
$ mknod /dev/sched_clock_overhead c 253 0
$ echo 5000 0 1 > /dev/sched_clock_overhead   // sched_clock()
$ dmesg | grep "^[0-9]" > sched_clock_overhead.log
$ echo 5000 1 1 > /dev/sched_clock_overhead   // getnstimeofday()
$ dmesg | grep "^[0-9]" > getnstimeofday_overhead.log
```

测试原理：

```
T1 = sched_clock();
(void)clock_func();  /* sched_clock() 或者 getnstimeofday() */
T2 = sched_clock();
Overhead = T2 – T1
```

采样结果统计：

Clock 函数/开销(ns)	最小值	平均值	最大值	抖动	标准偏差
sched_clock(cnt32_to_63)	105	116.2	236	131	9.5
sched_clock(累加算法)	193	200.9	243	50	2.9
getnstimeofday()	160	167.1	437	277	15

采样结果绘图后如下：

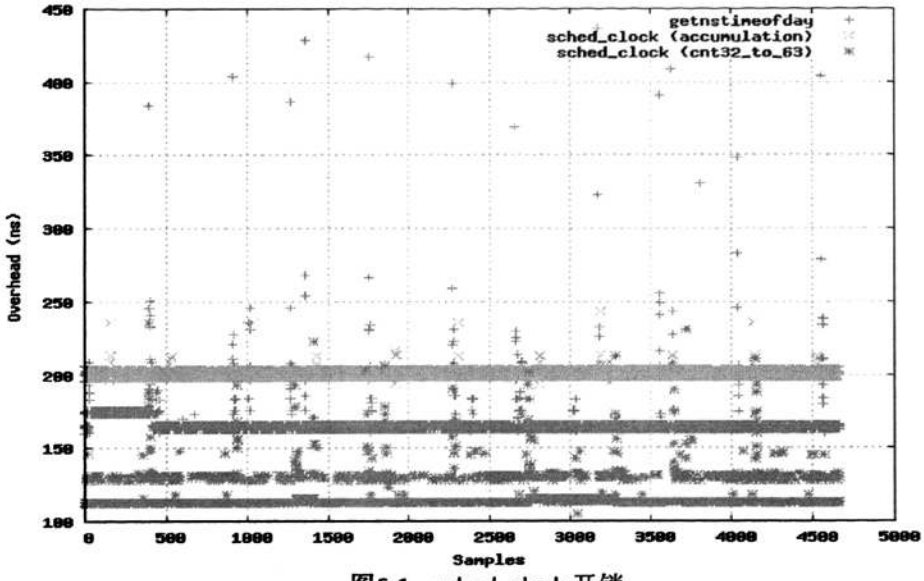

图6-1 sched_clock 开销

通过表格与图片得出，无论是最大开销还是平均开销，基于 cnt32_to_63()的 sched_clock()效果更好，而 getnstimeofday()的最大开销不如基于累加算法的 sched_clock()，但是平均性能更好，不过基于累加算法的 sched_clock()的稳定性更好。因此，为了最大程度地降低 Ftrace 开销，最终选择基于 cnt32_to_63()版本的 sched_clock()。

6.5.2 时钟精确性

该节测试时钟获取函数 clock_gettime()的精度与 clock_nanosleep()的延迟，测试例程优先级设置为 98，测试 clock_nanosleep()的时间间隔为 125us。

- clock_gettime()精度

测试步骤：

```
$ cd tools/rt/more/rtclock/
$ make
$ ./clock_gettime > clockgettime.log
// 通过 graph-samples.sh 对采样结果绘图
$ ../../scripts/graph-samples.sh -i clockgettime.log –t “Accuracy of clock_gettime”
```

测试原理：

```
clock_gettime(CLOCK_MONOTONIC,  &t1);
clock_gettime(CLOCK_MONOTONIC,  &t2);
/* calcdiff_us()求 t2 与 t1 的时间差，单位为 us */
delta = calcdiff_us(t2,  t1);
```

测试结果：

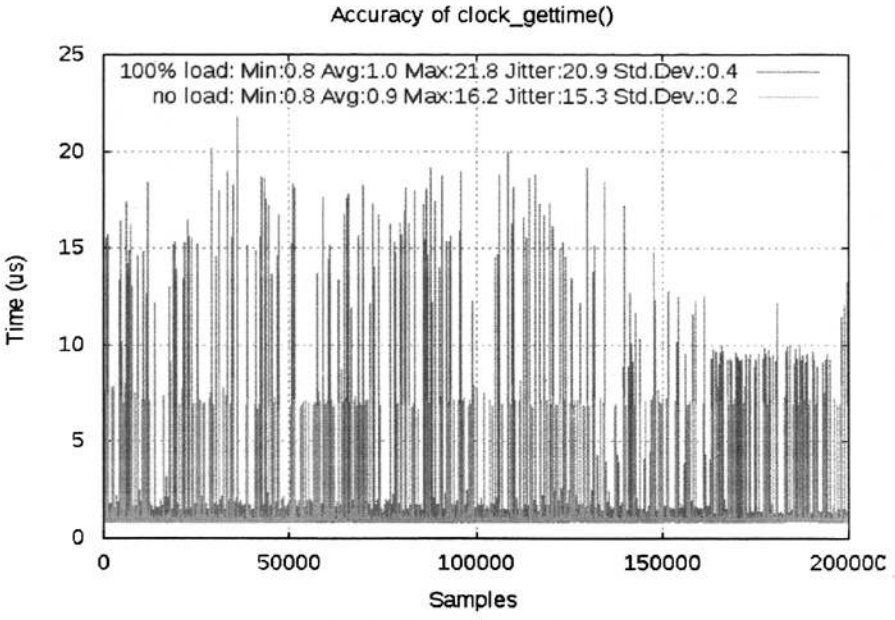

图6-2 clock_gettime()精度

结果表明，无负载下最大时钟精度损失是 15.3us，而有负载下，损失的精度增大到 20.9us，但还在 25us 内，并且平均性能相当。

- clock_nanosleep()延迟

通过 rt-tests 中的 cyclictest 测试 clock_nanosleep()的延迟。

测试步骤：

```
$ cd tools/rt/rt-tests/
$ make clean; make
$ ./cyclictest –p98 –t1 –m –n –i125 –l200000 –v > cyclictest.log
// 只保留延迟信息
$ cat cyclictest.log | cut –d ':' –f3 | tr –d ' ' | egrep –v "^$|^0" > cyclictest-latency.log
$ ../scripts/graph-samples.sh –i cyclictest-latency.log –t "cyclic_nanosleep() latency"
```

测试原理：

```
clock_gettime(CLOCK_MONOTONIC,  &t1);
clock_nanosleep(interval);
clock_gettime(CLOCK_MONOTONIC,  &t2);
delta = calcdiff_us(t2,  t1) – interval / 1000;
```

测试结果如下：

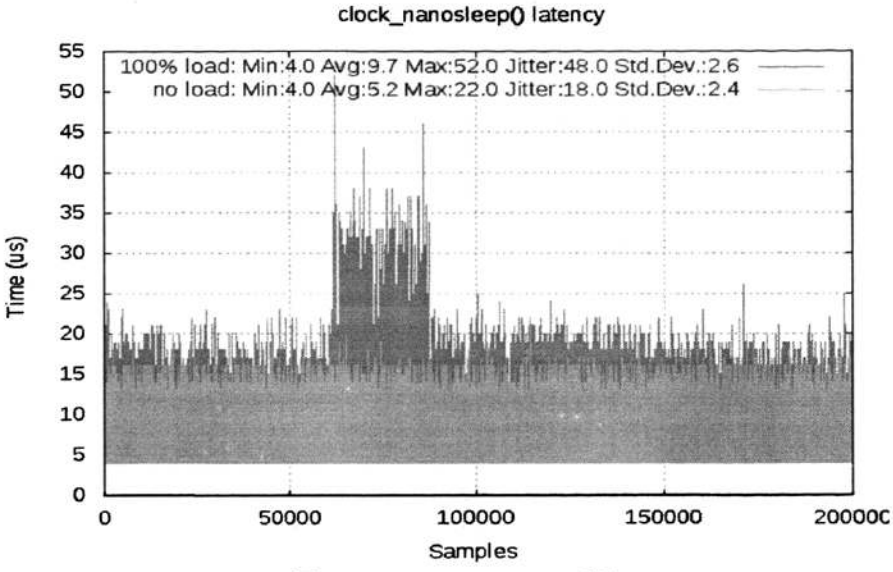

图6-3 clock_nanosleep 延迟

无负载下 clock_nanosleep()的延迟只有 22us，随着负载增加，延迟达到了52us，不过还在 55us 内。

6.5.3 中断延迟

由于没有现成工具可以测试中断延迟，因此本文作者专门编写了一个测试驱动。该驱动通过设置福珑 2F 的外部多功能时钟（MFPT）产生中断，并测试从中断发出到中断处理线程开始执行这段时间。线程优先级设置为 98。

该驱动为 drivers/platform/rt/interrupt_latency.c，编译内核时需要配置 INTR_LAT=m，采集数据要用到 tools/rt/more/interrupt_latency/latency_tracer.c。

测试步骤：

```
// 进入模块存放目录
$ insmod interrupt_latency.ko
$ mknod /dev/interrupt_latency c 253 0
$ chrt –p 98 $(pgrep interrupt)
$ cd tools/rt/more/interrupt_latency/
$ ./latency_tracer -i125 –l200000 –v1 –p95 > interrupt.log
$ ../../graph-samples.sh –i interrupt.log –t “Interrupt latency”
```

测试原理：

```
irq_handler () {
    /* th 为中断处理例程开始执行的时间 */
    do_gettimeofday(&th);
```

```
    irq_disable();
   /* interval+PERIOD 为中断产生的间隔，单位为 us,PERIOD 为时钟计数的到期
时间 */
    msleep(interval / 1000);
   /* ti+PERIOD 为中断发出时的时间 */
    do_gettimeofday(&ti);
    irq_enable();
}
```

```
delta = calcdiff_us(th,  ti) – PERIOD;
```

测试结果绘图如下：

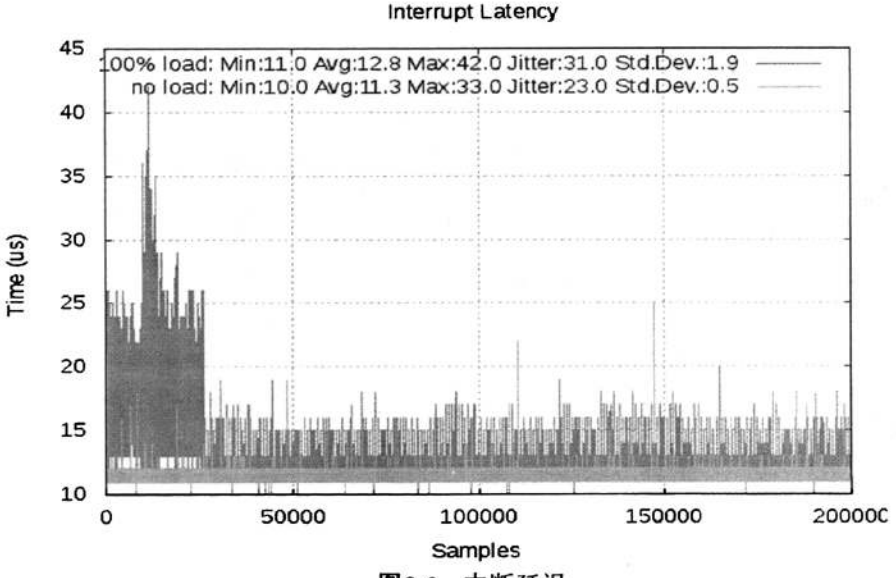

图6-4　中断延迟

通过比较发现，无负载下最大中断延迟为 33us，随着负载增加，中断延迟增加到了 42us，但还在 45us 内。可以发现，无负载时的最大值为第一个采样结果，而后续结果都在 25us 以内，相关原因会在后面分析。

6.5.4　调度器延迟

调度器延迟从任务被唤醒为就绪状态到进入调度器，这部分需要通过修改内核才能测试，测试工作比较困难，因此，一般都直接测试调度延迟，即从任务被唤醒（就绪态）到任务开始执行（运行态），等于调度器延迟加上任务调度时间。下面将测试的正文切换时间相当于任务调度时间，因此调度器延迟可以通过调度延迟减去正文切换延迟反应出来，这里不单独测试。

在测试中断延迟的基础上，通过编写一个字符设备驱动来测试调度延迟，把字符设备的读操作设置成阻塞模式直到在中断处理线程中获得中断后才唤醒。在中断处理线程记下唤醒时的时间为 t1，而用户空间的实时任务读取字符设备，读

取到内容后马上记录当前时间为 t2，这两个时间之差即调度延迟。

测试步骤：

```
// 接着测试中断延迟的步骤
$ cd tools/rt/interrupt_latency
// 用户空间的实时任务优先级比中断处理线程的 98 要低，设置为 95
$ ../latency_tracer –i 125 –l200000 –v3 –p95 > scheduling.log
$ ./graph-samples.sh –i "schedule.log" –t "Schedule latency"
```

测试原理：

字符设备驱动（中断线程的优先级设置为 98）：

```
/* irq_on 用于标记中断是否发生 */
static irq_on = 0;
/* 初始化一个等待队列 */
static wait_queue_head_t wq;
init_module() {
        init_waitqueue_head(&wq);
}
/* 当中断发生时，设置全部变量 irq_on 为 1，并唤醒实时任务 */
irq_handler () {
    irq_on = 1; /* 标记有中断发生 */
    do_gettimeofday(&t1);
    wake_up_interruptible(&wq);  /* 唤醒实时任务 */
     msleep(interval / 1000);  /* msleep 会进行任务调度 */
}
/* 阻塞式的读操作 */
device_read() {
    /* 等待中断发生，即等待 irq_on 为 1 */
    if (wait_event_interruptible(wq,  irq_on == 1))
        return –ERESTARTSYS;
    irq_on = 0;
}
```

用户空间的实时任务（优先级设置为 95）：

```
while (1) {
```

```
    read(path_to_char_dev,  timestamp,  sizeof(timestamp));
    gettimeofday(&t2,  NULL);
    sscanf(timestamp,  "%d %d\n",  t1.tv_sec,  t1.tv_usec);
    delta = calcdiff_us(t2,  t1);
}
```

测试结果如下：

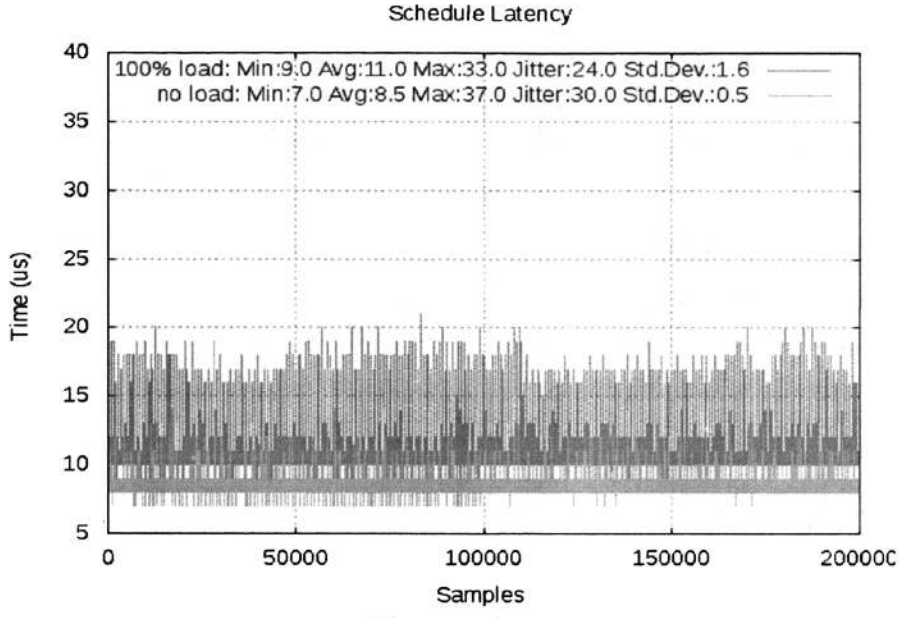

图6-5 调度延迟

上图显示，调度延迟在无负载下，最大为 37us，而在有负载下，最大为 33us，并没有随着负载增大而增大，但是平均时间有所增加，不过都在 40us 内。另外，最大值也出现在第一次，而后续结果都在 25us 以下，具体原因将在后面分析。

6.5.5 正文切换延迟

正文切换延迟是从调度器选好要调度的任务后到把进程上下文切换到另外一个进程所花时间。对于 Linux 内核，即 context_switch()函数的开销。

通过采用 SCHED_RR 调度策略，轮转调度两个实时任务（优先级都设置为 98），这两个任务本身仅调用 usleep(0)触发系统调度，所以 usleep(0)的开销即任务调度时间，又因为此时任务中只有两个最高优先级任务，调度器选择任务的时间很短，所以测试结果能够基本反映正文切换开销。

测试步骤：

```
$ cd tools/more/rtcontext_switch
$ ./context_switch > context_switch.log
```

```
$ ../../graph-sample.sh –i context_switch.log –t "Context Switch latency"
```

测试原理如下：

```
实时任务一：
for (i = 0; i < LOOPS; i ++) {
        clock_gettime(CLOCK_MONOTONIC,  &t1);
        usleep(0);      /* schedule */
        clock_gettime(CLOCK_MONOTONIC,  &t2);
        delta = calcdiff_us(t2,  t1) / 2;
}
实时任务二：
for (i = 0; i < LOOPS; i ++) {
        usleep(0);   /* schedule */
}
```

测试结果：

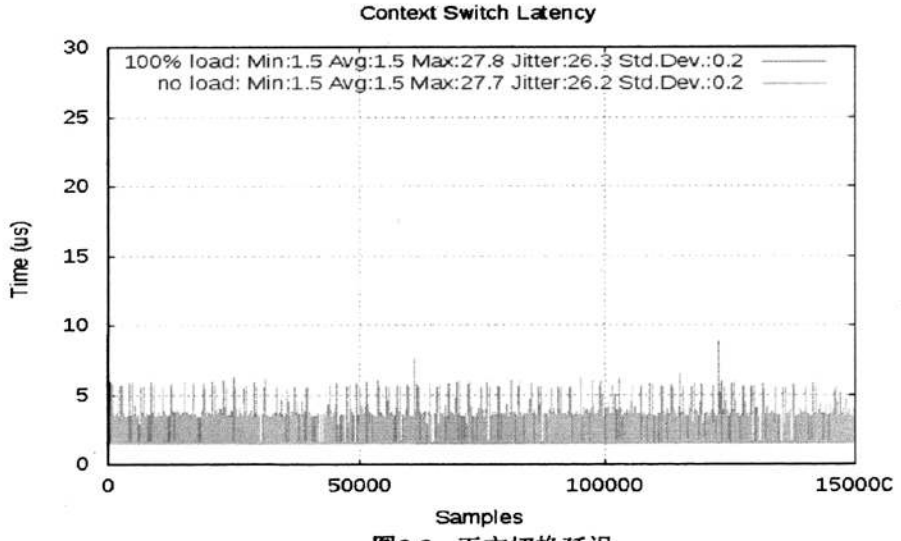

图6-6 正文切换延迟

由上图可知，无负载下正文切换最大延迟为 26.2us，而增大负载后延迟为 26.3us，随着负载增大，正文切换延迟变化不大。同样地，最大值都出现在第一次，而其他情况下没有超过 10us。具体原因也到后面统一分析。

6.5.6 优先级调度

实时操作系统采用基于优先级的调度策略确保实时任务优先得到执行。如果这种调度策略没有正确实现，将出现低优先级任务抢占高优先级任务的情况，并导致高优先级任务的执行时间不可预测，这对于实时操作系统来说是不允许的。

下面通过 Ftrace 的 sched_switch Tracer 跟踪四个不同优先级任务（对于 SCHED_DEADLINE 调度策略，优先级通过任务最早截止时间反应出来）在三种不同调度策略下的调度过程，进而验证这些调度策略的正确性。

详细的测试步骤请参考 tools/analyze-sched/README，工具 tools/more/rtprio/prio.c 用于自动产生四个不同优先级的任务用于测试 SCHED_FIFO 和 SCHED_RR 调度策略，而对于 SCHED_DEADLINE 调度策略的测试任务，可以通过 tools/test-deadline/periodic.c 来自动产生。

测试原理：

1. 通过 Ftrace 的 sched_switch Tracer 跟踪四个不同优先级任务的执行情况，测试完后导出测试结果到 sched_switch.txt 中。
2. 通过资料[33]中的sched_switch工具（即tools/analyze-sched/analyze.c）把sched_swtich.txt文件中的内容转换为一份vcd格式的数据，继而通过gtkwave分析该数据，然后把数据导出到一个pdf文件中。

测试结果如下：

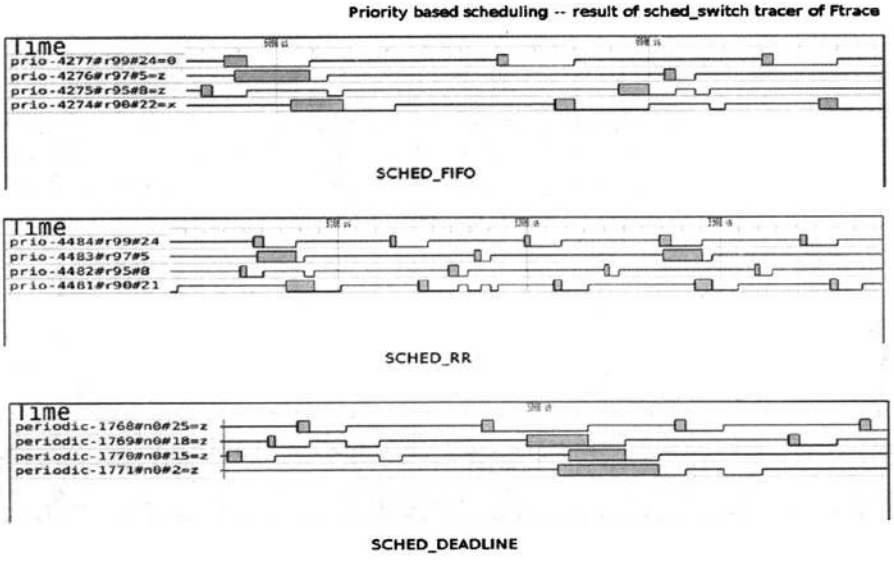

图6-7 优先级调度策略的正确性

如上图所示，对于三种不同调度策略，位于最上面的任务优先级最高(99)，位置越往下优先级越低，最下面的为 90，阴影部分是调度延迟，随后的凹槽是指该任务得到执行。从图中可以看出，只存在高优先级任务抢占低优先级任务，说明三种调度策略都满足基于优先级的调度。

6.5.7 信号（同步）

通过 rt-tests 中的 signaltest 测试从 pthread_kill()发出信号到从 sigwait()接收到信号这段时间。测试用例的优先级被设置为 98。

测试步骤：

```
$ cd tools/rt-tests
$ signaltest –l200000 –p98 –t2 –m –v > signaltest.log
$ cat signaltest.log | cut –d' ' –f3 | cut –d':' –f3 | tr –d ' ' > signalest-latency.log
$ ../../graph-samples.sh –i signaltest-latency.log –t "pthread_kill -> sigwait latency"
```

测试原理：

```
任务一(thread1):
clock_gettime(CLOCK_MONOTONIC ,  &t1);
pthread_kill(thread2,  SIGRTMIN);
任务二(thread2):
sigwait(&sigset,  &sig);
clock_gettime(CLOCK_MONOTONIC,  &t2);
delta = calcdiff_us(t2,  t1);
```

测试结果如下：

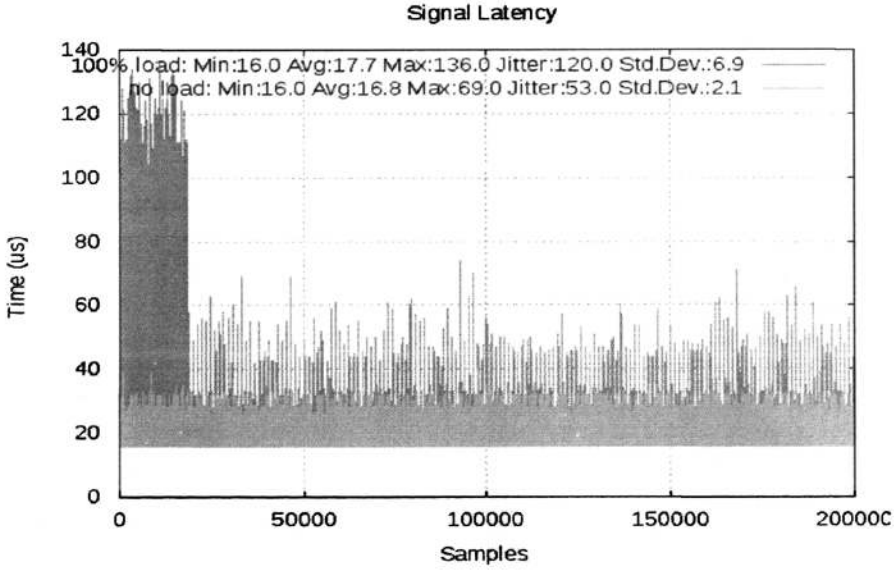

图6-8 pthread_kill -> sigwait 信号延迟

如图所示，无负载时最大延迟是 69us，有负载时延迟增大为 136us，不过无负载下延迟都在 70us 内，而有负载下延迟比 100us 稍大，但还在 150us 内。

6.5.8 信号量（互斥）

通过 rt-tests 下的 ptsematest 进行测试信号量延迟。由于原测试程序只能在屏幕固定位置动态地输出采样数据，不方便统计，因此测试时对代码进行了修改。测试用例的优先级设置为 98。

测试步骤：

```
$ cd tools/rt-tests
$ ./ptsematest -i125 –t1 –p98 –l200000 > ptsematest.log
$ ../../graph-samples.sh –i ptsematest.log –t “Semaphore latency
```

测试原理如下：

```
任务一(thread1):
clock_gettime(CLOCK_MONOTONIC ，  &t1);
pthread_mutex_unlock(&mutex);
任务二(thread2):
pthread_mutex_lock(&mutex);
clock_gettime(CLOCK_MONOTONIC，  &t2);
delta = calcdiff_us(t2，  t1);
```

测试结果如下：

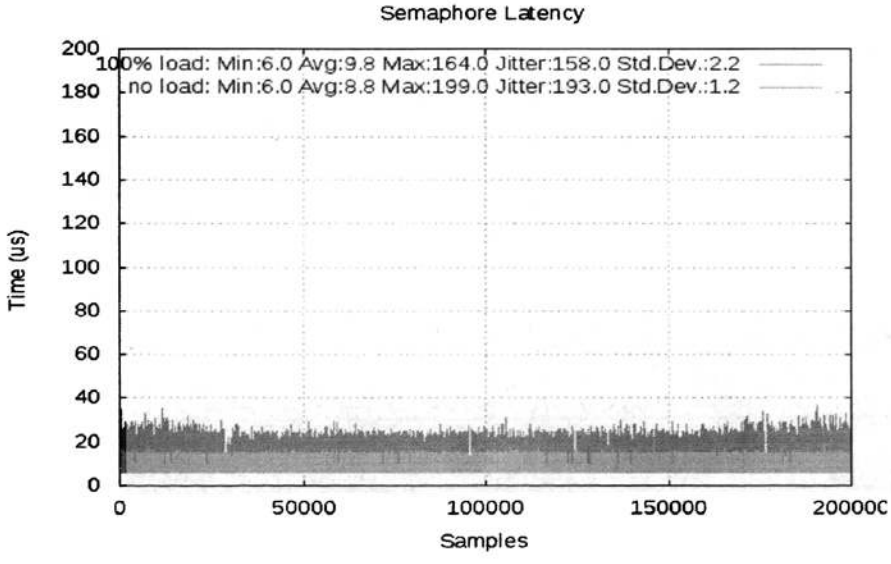

图6-9 信号量延迟

如上图所示，无负载下信号量最大延迟为 **164us**，增大负载后延迟为 **199us**，随着负载增加信号量延迟并没有明显增大。同样地，最大值都出现在第一次，后面都在 **40us** 以下，具体原因将在后面分析。

6.5.9 共享内存（通信）

Linux 内核提供了各种丰富的进程通信机制，但并不是所有的通信方式都提供了实时保障。考虑到共享内存效率更高，本文仅以它为例进行测试。测试时要用到 rt-tests 中的 svsematest。同样地，为方便产生采样数据，本文作者对原程序的代码进行了修改。

测试步骤：

```
$ cd tools/rt-tests
$ ./svsematest -i125–p98 –l200000 > svsematest.log
$ ../../graph-samples.sh –i svsematest.log –t “Shared memory latency
```

测试原理如下：

```
任务一(thread1)：
clock_gettime(CLOCK_MONOTONIC ，  &t1);
sb.sem_op=SEM_UNLOCK; sb.sem_num = SEM_WAIT_FOR_SENDER;
semop(semid, &sb, 1);
任务二(thread2):
sb.sem_op=SEM_LOCK; sb.sem_num = SEM_WAIT_FOR_SENDER;
semop(semid, &sb, 1);
clock_gettime(CLOCK_MONOTONIC，  &t2);
delta = calcdiff_us(t2，  t1);
```

测试结果如下：

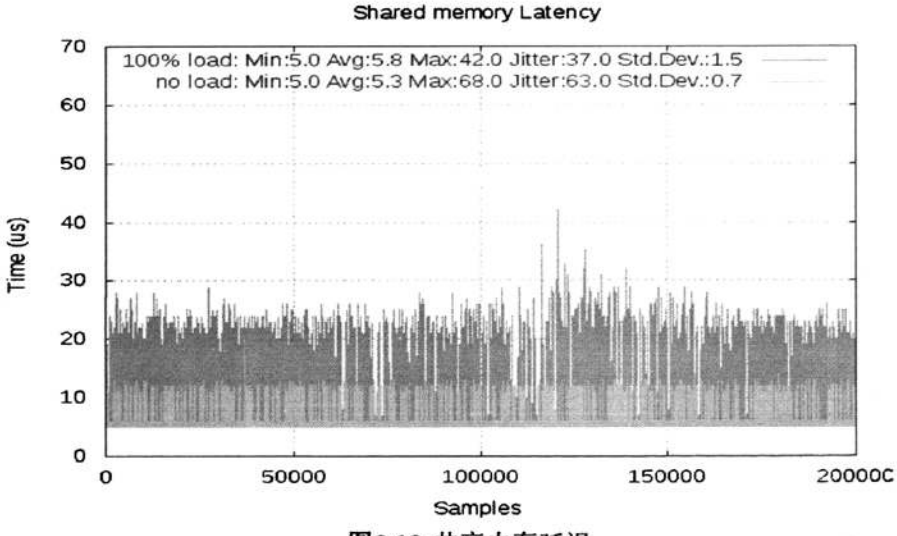

图6-10 共享内存延迟

通过上图发现，无负载时最大延迟为 68us，增加负载后最大延迟为 42us，最大延迟并没有增大，但平均延迟有所增加，不过都在 70us。同样地，对于无负载情况，第一次测试结果为最大值。

6.5.10 优先级反转

根据第二章图 2-10关于优先级反转的描述，如果不支持优先级继承协议，优先级反转可能导致高优先级任务被低优先级任务不可预期的阻塞，因此，实时操

作系统需要实现优先级继承协议以避免这种情况，Linux下实现了优先级继承协议，这里测试该协议是否正常实现。

首先通过 ltp 中的 testcases/realtime/func/pi_tests/testpi-0.c 测试当前的 pthread 库是否支持优先级继承。

测试步骤：

```
$ cd testcases/realtime/func/pi_tests/
$ ./testpi-0
```

测试原理如下：

```
if (sysconf(_SC_THREAD_PRIO_INHERIT) == -1) {
    printf("No Prio inheritance support\n");
}
printf("Prio inheritance support present\n");
```

测试结果：

```
$ testcases/realtime/func/pi-tests/testpi-0
LIBC_VERSION: glibc 2.10.2
LIBPTHREAD_VERSION: NPTL 2.10.2
Prio inheritance support present
```

可见，该测试环境的 pthread 库支持优先级继承。

接着测试通过 pthread_mutexattr_setprotocol()函数设置 mutex 的属性分别为 PTHREAD_PRIO_INHERIT 与 PTHREAD_PRIO_INHERIT 时高优先级任务从请求锁到获得锁所花费时间。测试时，需要明确制造一个优先级反转现象，即低优先级任务先获得锁，其后，高优先级任务也需要获得该锁，紧接着有中间优先级任务启动。如果支持优先级继承协议，那么高优先级任务请求锁时将提升低优先级任务的优先级为自身优先级，从而阻止中间优先级任务抢占，反之，如果不支持优先级继承，那么中间优先级任务将抢占低优先级任务，并导致高优先级任务在中间优先级任务执行完后才有机会获得锁，从而使高优先级任务的执行不可预测。

测试原理：

Thread1(high_thread):

```
/* 设置优先级为最高 95，设置调度策略为 SCHED_FIFO */
pthread_setschedparam(pthread_self(), SCHED_FIFO, NULL);
pthread_setschedprio(pthread_self(), 95);
pthread_getschedparam(pthread_self(), &policy, &param);
sigemptyset(&sigset);
sigaddset(&sigset, SIGUSR1);
```

```
sigprocmask (SIG_BLOCK, &sigset, NULL);
/* 等待中间优先级任务(mid_thread)唤醒 */
sigwait(&sigset, &sigs);
/* 记录请求锁的时间 */
clock_gettime(CLOCK_MONOTONIC, &tr);
pthread_mutex_lock(&mut);
/* 记录获得锁的时间 */
clock_gettime(CLOCK_MONOTONIC, &tg);
/* 计算从请求锁到获得锁之间的时间间隔 */
diff = calcdiff_ns(tg, tr);
```

Thread2(mid_thread):

```
/* 设置优先级为中间大小 92，设置调度策略为 SCHED_FIFO */
pthread_setschedparam(pthread_self(), SCHED_FIFO, NULL);
pthread_setschedprio(pthread_self(), 92);
pthread_getschedparam(pthread_self(), &policy, &param);
sigset_t sigset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGUSR2);
sigprocmask (SIG_BLOCK, &sigset, NULL);
/* 等待 low_thread 唤醒 */
sigwait(&sigset, &sigs);
/* 马上唤醒 high_thread */
pthread_kill(high_thread, SIGUSR1);
/* 执行无限制长的时间， 测试时，loops3 只被初始化为 1 */
while (loops3 > 0) {
    loops3 --;
}
```

Thread3(low_thread):

```
/* 设置调度策略为 SCHED_FIFO，优先级为 90 */
pthread_setschedparam(pthread_self(), SCHED_FIFO, NULL);
pthread_setschedprio(pthread_self(), 90);
pthread_getschedparam(pthread_self(), &policy, &param);
```

```
/* 获得锁 */
pthread_mutex_lock(&mut);
/* 唤醒 mid_thread */
pthread_kill(mid_thread， SIGUSR2);
/* 释放锁 */
pthread_mutex_unlock(&mut);
```

Thread1， thread2， thread3 的创建顺序：

```
pthread_create(&high_thread， NULL， thread1， NULL);
pthread_create(&mid_thread， NULL， thread2， NULL);
pthread_create(&low_thread， NULL， thread3， NULL);
```

关于优先级继承协议的设定：

```
enum
{
  PTHREAD_PRIO_NONE，   /* 不支持优先级继承 */
  PTHREAD_PRIO_INHERIT /* 支持优先级继承 */
};
pthread_mutex_t mut;
pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
/* 支持优先级继承，如果设置 PTHREAD_PRIO_NONE 则不支持 */
pthread_mutexattr_setprotocol(&mutex_attr， PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&mut， &mutex_attr);
```

测试结果：

负载	协议/ 延迟 (us)	最小值	平均值	最大值	抖动	标准偏差
有负载	PTHREAD_PRIO_INHERIT	5.2	7	47	41.8	3.2
	PTHREAD_PRIO_NONE	1248.2	1538.6	52191.1	50942.9	1440.8
无负载	PTHREAD_PRIO_INHERIT	5.3	7.2	47.9	42.6	2.6
	PTHREAD_PRIO_NONE	1299.5	1447.9	1517.2	217.7	39.6

如上表所示，不管是否有负载，即使存在优先级反转，如果给 mutex 设置了优先级继承协议，那么高优先级任务获得锁的时间可确定，说明优先级继承协议被正确实现。如果没有设置优先级继承协议，那么获得锁的时间跟中间优先级任务的执行时间有关，是不确定的，并且随着负载增加，获得锁的时间更加不确定。

6.5.11 测试结果总结

首先分析前文提到的有趣现象：即大部分延迟的最大值都为第一个采样结果。可能的原因包括如下两个方面：

第一个方面即注意事项中提到的对于多任务的测试，可能存在多个任务没有同时初始化的情况，测试结果包含了其中一个任务等待另外一个任务初始化的时间，即使引入同步机制，也可能包括同步措施的延迟；另外一个方面，即第一次测试时，相关指令和数据还没有缓存，因此比后续结果都要大。在实际应用中，往往会等到整个系统或者任务初始化完以后才会开始工作，因此，第一个测试结果可以忽略不计。

下面对采样结果进行总结：

1、综合第 4 章的Ftrace移植效果以及该章的sched_clock()测试结果可知，龙芯平台上的Ftrace能以较小开销提供高精度时间戳信息，而且包括静态Function Tracer、动态Function Tracer、图形化Function Tracer在内的函数追踪工具都能够在龙芯平台上正常工作，并且能够有效辅助实时抢占补丁的调优。

2、如果忽略第一个采样结果，除了 pthread_kill->sigwait 的信号延迟在有负载下超过 100us 外，其他所有测试结果，包括中断延迟、调度延迟、正文切换延迟、clock_getime()和 clock_nanosleep()、信号、信号量和共享内存都在 100us 以内，甚至更小。并且随着负载增大，系统都能够正常工作，而且延迟没有明显增大，还在可确定范围内。

3、包括内核现有的 SCHED_FIFO 和 SCHED_RR 调度策略以及目前正在开发的 SCHED_DEADLINE 调度策略在内，都正确实现了优先级调度。

4、结合 pthread 提供的优先级继承 Mutex（PI-mutex），Linux 能够解决不可预测的优先级反转问题。

因此，龙芯平台的实时抢占补丁已经能够满足一般的实时应用，并且无负载下的测试结果表明，如果进行合理的配置（运行环境，使用的外部设备等），还能够满足一定的硬实时要求。

但是，由于Linux系统太复杂，无法通过形式化验证达到较高的安全级别[27]，因此，在一些Safety Critical环境（涉及到人身安全、巨大的财产安全等）中，还是应该尽量避免使用。

6.6 跟其他平台的比较

上面测试了龙芯平台上实时抢占补丁的各种性能指标，为了比较龙芯平台跟其他平台的不同，该节还测试了另外一款平台。

该平台的环境：

机器	威盛(VIA) EPIA-miniITX
处理器	VIA Nehemiah 1002M

内存	256M
发行版	Debian GNU/Linux 5.0
内核	2.6.33.1-rt11 ， http://dev.lemote.com/code/rt4ls
内核配置	arch/x86/configs/via_rt_defconfig
Glibc	2.7
系统负载	通过上文的 load.sh 自动产生大量的系统负载。

考虑到本文篇幅限制，该节仅以rt-tests中流行的cyclctest工具测试clock_nanosleep()的延迟，因为它不仅能够反映时钟精确性，而且能够潜在地反映一些其他性能指标，比如中断延迟、调度延迟等。测试的基本原理同6.5.2。

测试结果：

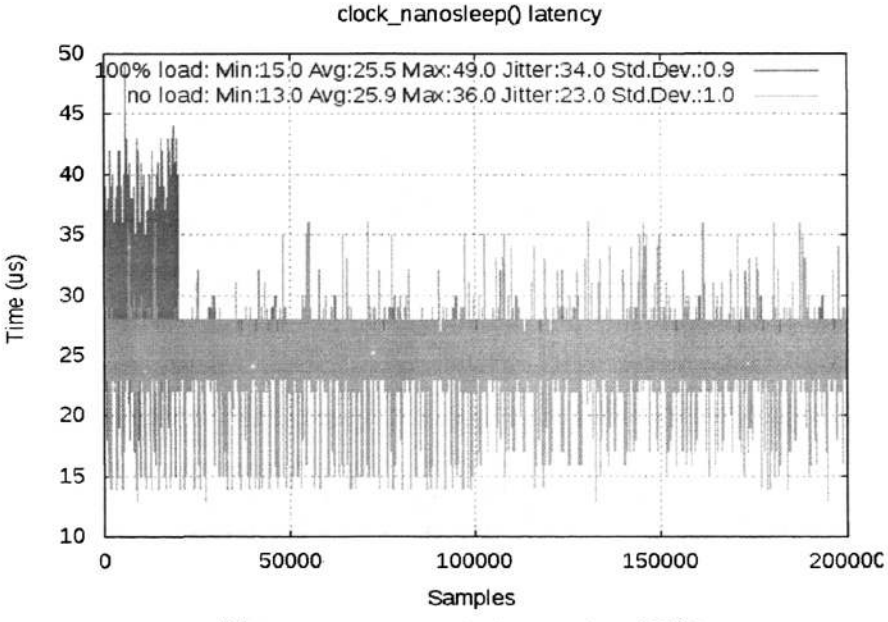

图6-11 EPIA mini-ITX:clock_nanosleep()延迟

由于EPIA mini-ITX和福珑 2F除了处理器不一样以外，外围硬件配置也不一致，所以上述测试结果不能完全比较两者的实时性能。不过，通过比较上图与图 6-3可以看出，虽然VIA的主频比龙芯 2F要高，但是EPIA mini-ITX在无负载时的实时性能明显不如福珑 2F，即使有负载，福珑 2F的平均性能与最小延迟结果也比EPIA mini-ITX要好，但是，随着负载增加，EPIA mini-ITX的延迟增加不大，而福珑 2F增加较大，可能原因是福珑 2F上的相关驱动，包括网卡、显卡驱动带来了较大延迟，另外，福珑 2F上的正文切换负载也可能更大。不过这些都需要进一步的实验验证，考虑到篇幅关系，本文不做进一步讨论。

如果想与更多其他平台的实时抢占补丁的测试结果进行比较，请参考：

https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch

第7章总结与展望

7.1总结

本文在阐述实时操作系统的基本原理之后，以实时抢占补丁为例分析了低延迟/自愿抢占技术、抢占技术、中断线程化、高精度时钟、实时调度策略、临界区抢占、优先级继承等实时改造技术以及 Ftrace、Perf 等实时调试与优化技术。接着探讨了 MIPS 平台上 Ftrace 的移植，并以龙芯处理器平台为例讨论了实时抢占补丁的移植与优化，最后进行了性能评测与分析。

通过性能评测与分析发现，本文移植与优化的实时抢占补丁在 MIPS（龙芯）上能够满足一定的硬实时要求，也验证了龙芯作为一款国产高性能处理器，具有一定的实时处理能力，能够潜在地应用于工业自动化、数字控制和汽车电子，甚至是国防、航空航天等领域中。

另外，本文的大部分研究成果目前以被 Linux 实时抢占补丁项目的官方接收，这意味着龙芯将得到 Linux 实时抢占补丁项目官方的支持。

7.2展望

虽然本文取得了一定的成果，但是在以下几个方面还有很多后续工作：

1. 理论研究

本文对实时抢占补丁的部分关键实时改造技术进行了深入分析，但是对包括可抢占 RCU、优先级继承协议、实时调度策略、Perf 在内的其它技术还有待进一步研究。

例如，研究 RCU 的工作原理与应用场景；分析 SCHED_DEADLINE 调度策略（最早截止时间优先）的实现原理，把它移植到最新的实时抢占补丁（rt/2.6.33）上，并争取实现包括 LLF（最低松弛度优先）在内的其它实时调度策略，以满足更多实时应用环境的需求；研究 Perf 的工作原理和平台相关性，以便移植到更多平台上进而辅助实时抢占补丁的优化。

2. 实时系统优化

虽然龙芯平台上的实时抢占补丁已经具备了一定的实时处理能力，但是还有待进一步优化。例如，理解 MIPS 架构的 Cache 与 TLB 管理机制，对 MIPS 平台的上下文切换进行优化；分析 VDSO 的工作原理，在当前 MIPS 的 VDSO 的工作基础上，研究加速其他系统调用的可行性；更详细地分析龙芯平台上各种外部负载（比如网络、图形）对实时性能的影响，进行针对性的优化。

3. 实时系统测试

虽然本文详细介绍了各种实时性能指标的测试原理与方法，但是相关工具还比较零散，并没有完全自动化。因此，后期可以考虑整合当前的各种实时测试用

例，开发出一个集成的、具有良好扩展性和可移植性的实时测试框架并利用该框架进行更深入的性能评测与分析。

4. 实时应用开发

本文主要介绍的是实时操作系统方面的内容，实时应用开发方面的内容介绍得较少而且不够深入。后期希望在进行大量实时应用开发实践的基础上，深入了解各种潜在的实时应用需求，进而加深对相关内容的理解。

参考文献

[1] Peter Feuerer. *Benchmark and comparison of real-time solutions based on embedded Linux. Diploma thesis*, Jul 30, 2007

[2] 白树伟. *嵌入式实时 Hypervisor:XtratuM.* 硕士学位论文. May, 2009

[3] Regnier, P., Lima, G., and Barreto, L. *Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems.* Operating Systems Review (to apear), pages 1-12, 2008

[4] RTAI. https://www.rtai.org

[5] Xenomai. http://www.xenomai.org

[6] XtratuM. http://www.xtratum.org

[7] Preempt-RT. http://rt.wiki.kernel.org

[8] RTLinux. http://www.rtlinux-gpl.org

[9] Real Time Linux Workshop: http://www.realtimelinuxfoundation.org

[10] Comp.realtime: *Frequently Asked Questions (FAQs) (version 3.5).*

http://www.faqs.org/faqs/realtime-computing/faq/

[11] C.M Krishna, Kang G. Shin. *Real-time Systems.* 清华大学出版社, 2001

[12] The Single UNIX® Specification, Version 2.

http://www.opengroup.org/onlinepubs/007908799/xsh/realtime.html

[13] Kushal Koolwal. *Myths and Realities of Real-Time Linux Software Systems.* Proceedings of Eleventh Real-Time Linux Workshop, pages 13-18, 2009

[14] Michael Opdenacker. *Introduction to rt-preempt patches.* http://free-electrons.com, Jul 11, 2006

[15] Max-Gerd Retzlaff. *Kernel Preemption.* Linux Internals Seminar WS 2003/2004

[16] Paul E. McKenney. *'Real Time' vs. 'Real Fast': How to Choose?.* Proceedings of Eleventh Real-Time Linux Workshop, pages 1-12, 2009

[17] LUI SHA, RAGUNATHAN RAJKIMAR, JOHN P. LEHOCZKY. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* IEEE TRANSACTIONS ON COMPUTERS, VOL, 39, NO. 9. pages 1175-1184, SEPTEMBER. 1990

[18] Steven Rostedt and Darren V. Hart. *Internals of the RT Patch.* Proceedings of the Linux Symposium, pages 161-172, 200

[19] 杨燚. *Linux 实时技术与典型实现分析.* 2005

http://www.ibm.com/developerworks/cn/linux/l-lrt/part1,
http://www.ibm.com/developerworks/cn/linux/l-lrt/part2

[20] Jan Altenberg. *Using the Realtime Preemption Patch on ARM CPUs.* Proceedings

of Eleventh Real-Time Linux Workshop, pages 229-236, 2009

[21] Johnnie Hancock. *Finding Sources of Jitter with Real-Time Jitter Analysis.* Application Note 1448-2, Agilent Technologies, Mar 23, 2007

[22] 邵鹏 *Jitter 寻根溯源.* 2009

[23] B. Ip. *Performance Analysis of VxWorks and RTLinux.* Technical report, Department of Computer Science, Columbia University, 2002

[24] Realtime Testing Best Practices.

http://elinux.org/Realtime_Testing_Best_Practices

[25] Bill Gallmeister. *Posix.4 Programmers Guide: Programming for the Real World.* O'Reilly Media; 1st edition, Jan 1, 1995

[26] A S Prakash. Real Time Programming with Pthreads

[27] Lijuan Wang, Chuande Zhang, Zhangjin Wu, Nicolas Mc.Guire and Qingguo Zhou. *SIL4Linux: An attempt to explore Linux satisfying SIL4 in some restrictive conditions.* Proceedings of Eleventh Real-Time Linux Workshop, pages 111-115, 2009

[28] Clark Williams. *Linux Scheduler Latency.* Embedded Systems (Europe), May, 2002

[29] 白中英. 计算机组成原理（第三版）. 科学出版社. 2000

[30] S. Baskiyar ad N. Meghanathan, *A survey of contemporary Real-Time Operating Systems*, Informatica 29, pages 233-240, 2005

[31] R. Yerraballi, *Real-Time Operating Systems: An Ongoing Review*, presented at 21st IEEE Real-Time Systems Symposium, Orlando, 2000

[32] Jinggang Wang. *Real-Time Linux Kernel Design, Minimization and Optimization.* Virginia Tech CS5204 Operating System Project

[33] Herman ten Brugge. Sched_switch.c.

http://www.osadl.org/Visualize-the-temporal-relationship-of-L.taks-visualizer.0.html

[34] StrongMail Systems. *Extreme Linux Performance Monitoring and Tuning.* 2006

[35] Red Hat Inc. *Red Hat Enterprise Linux: Performance Tuning Guide*, 2004

[36] Red Hat Inc. *MRG Realtime 1.0: Realtime Tunning Guide.* 2008

[37] Richard S. Herbel and Dang N.Le. *Tuning Linux to Meet Real Time Requirements.* Proceeding of SPIE--The International Society for Optical Engineering.May, 2007

[38] Fank Rowand. *Adventures In Real-Time Performance Tuning.* November 7. 2008

[39] Cisco Systems. *Design Best Practices for Latency Optimization.* 2007

[40] AnswerGuy. *A bit like the VDSO in reverse.* http://lwn.net/Articles/172224/. Feb 16, 2006

[41] David Daney. *MIPS: Preliminary vdso.*

http://patchwork.linux-mips.org/patch/975/. Feb 2010

[42] Matt Mackall. *slob: introduce the SLOB allocator.* http://lwn.net/Articles/157944/. Nov 1, 2005

[43] Corbet. *The SLUB allocator.* http://lwn.net/Articles/229984/. April 11, 2007

[44] M. Tim Jones. *Anatomy of the Linux slab allocator.*

http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/. May 15, 2007

[45] Eric Steven Raymond. *The Cathedral and the Bazaar.*

http://catb.org/esr/writings/homesteading/. Feb 19, 2010

[46] Paul E. McKenney. *Attempted summary of "RT patch acceptance" thread, take 2.* http://lwn.net/Articles/143323/. Jul 11, 2005

[47] Rick Lehrbaum. *Real-time Linux Software Quick Reference Guide.* Jan 19, 2001

[48] 李凤宪,郭锐锋,李家霁. *实时 Linux 操作系统的分析和比较.* 中国科学院计算技术研究所第七届计算机科学与技术研究生学术讨论会. 2002

[49] Realtime Linux: OSADL.

http://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

[50] S. Dietrich and D. Walker. *The Evolution of Real-Time Linux.* Proceeding of 7th Real-Time Linux Workshop, Nov. 2005

[51] Klaas van Gend. *Real Time Linux patches: history and usage.* FOSDEM 2006

[52] Dario Faggioli, Fabio Checconi, Michael Trimarchi and Claudio Scordino. *An EDF scheduling class for the Linux kernel.* Proceedings of Eleventh Real-Time Linux Workshop, pages 197-204, 2009

[53] SCHED_DEADLINE homepage: http://gitorious.org/sched_deadline/pages/Home

[54] TLSF: Memory Allocator for Real-Time. http://rtportal.upv.es/rtmalloc/

[55] Thomas Gleixner, Douglas Niehaus. *Hrtimers and Beyond: Transforming the Linux Time Subsystems.* Proceedings of the Linux Symposium. July 19th-22nd 2006

[56] 刘冀. Tickless 技术研究及其在嵌入式系统中的实现. 硕士学位论文. May, 2009

[57] M. Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler.* IBM developer works. Dec 15, 2009

[58] Ankita Garg. *Real-Time Linux Kernel Scheduler.* linuxjournal.com. Aug 01, 2009

[59] Ingo Molnar. *Real Time group scheduling.* Linux 2.6.33:

Documentation/scheduler/sched-rt-group.txt. 2008

[60] Steven Rostedt. *RT-mutex implementation design.* Linux 2.6.33:

Documentation/rt-mutex-design.txt. 2006

[61] Steven Rostedt. *RT-mutex subsystem with PI support*. Linux 2.6.33: Documentation/rt-mutex.txt. 2006

[62] xvMalloc Memory Allocator. http://code.google.com/p/compcache/wiki/xvMalloc

[63] Steven Rostedt. *LOGDEV device and utilities*. http://rostedt.homelinux.com/logdev/README. 2005

[64] Steven Rostedt. *mcount tracing utility*. http://lwn.net/Articles/264029/

[65] Ingo Molnar. Latency Tracer. http://lkml.org/lkml/2008/2/8/435

[66] Theodore Ts'o, Li Zefan and Tom Zanussi. *Event Tracing*. Linux 2.6.33: Documentation/trace/events.txt. 2009

[67] Steven Rostedt. *Ftrace- Function Tracer*. Linux 2.6.33: Documentation/trace/ftrace.txt. 2008

[68] Mike Frysinger. *Function tracer guts*. Linux 2.6.33: Documentation/trace/ftrace-design.txt. 2009

[69] Dominic Sweetman. *See MIPS Run(Second Edition)*. Elsevier Inc. 2007

[70] Loongson.cn. 龙芯 2F 用户手册: http://www.loongson.cn/uploadfile/file//20080821113149.pdf. 2008

[71] Lemote.com. 龙芯内核移植开发手册: http://dev.lemote.com/drupal/node/42. 2009

[72] Wu Zhangjin, Nicholas MC.Guire. *Porting RT-preempt to Loongson2F*. Proceedings of Eleventh Real-Time Linux Workshop, pages 169-174, 2009

[73] Thomas Gleixner. 2.6.33-rt6: http://lwn.net/Articles/378395/. 12 Mar 2010

[74] Steven Rostedt. *Finding Origins of Latencies Using Ftrace*. Proceedings of Eleventh Real-Time Linux Workshop, pages 117-130, 2009

[75] 刘明. *Ftrace 简介*. http://www.ibm.com/developerworks/cn/linux/l-cn-ftrace/. 2009

[76] Steven Rostedt. *Debugging the kernel using Ftrace*. lwn.net. 2009

[77] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc. 2008

[78] The Perl Programming Lanuage. http://www.perl.org

[79] Jan Kiszka. *Towards Linux as a Real-Time Hypervisor*. Proceedings of Eleventh Real-Time Linux Workshop, pages 205-214, 2009

[80] Requeue-PI. *Making Glibc Condvars PI-Aware*. Proceedings of Eleventh

Real-Time Linux Workshop, pages 215-228, 2009

[81] Hubertus Franke, Fusty Russell, *Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux*. OLS. 2002

[82] Ulrich Drepper. *Futexes Are Tricky*. Aug 1, 2009

[83] Steven Rostedt. *Linux 2.6.33: Lightweight PI-futexes*. 2006

作者研究生期间科研成果

发表论文

1. **Wu Zhangjin**, Nicholas MC.Guire. Porting RT-preempt to Loongson2F. Proceedings of Eleventh Real-Time Linux Workshop, pages 169-174, 2009
2. **Wu Zhangjin**, Huang Xingwen, Ding Ying, Zhou Rui, Zhou Qingguo. A CGI+AJAX+SVG Based Monitoring Method for Distributed and Embedded System. Proceedings of 2008 the 1st IEEE International Conference on Ubi-Media Computing and Workshops, pages 144-148, 2008
3. Lijuan Wang, Chuande Zhang, **Zhangjin Wu**, Nicolas Mc.Guire and Qingguo Zhou. SIL4Linux: An attempt to explore Linux satisfying SIL4 in some restrictive conditions. Proceedings of Eleventh Real-Time Linux Workshop, pages 111-115, 2009

参与项目

1. 龙芯 Linux 操作系统开发，2009 年 2 月至今

   这是作者在江苏龙芯梦兰科技股份有限公司（Lemote）实习期间开展的项目，作者负责维护了该公司产品（包括福珑盒子、逸珑上网本、玲珑一体机在内）的最新 Linux 内核支持，包括功耗管理、热键管理、内核压缩支持等，并把相关的内核支持提交到了 Linux 官方。目前作者还在维护该项目，其首页为：

   http://dev.lemote.com/code/linux-loongson-community

2. 龙芯平台的实时操作系统研发，2009 年 2 月至今

   这也是作者在江苏龙芯梦兰科技股份有限公司实习期间开展的项目，是论文相关项目。项目期间，作者最早把 linux-2.6.26.8-rt16 的实时抢占补丁移植到了龙芯处理器平台，为龙芯处理器开发了第一款实时操作系统。目前，相关研究成果已经提交到了实时抢占补丁的官方：

   git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git rt/2.6.33

   而最新的研发进展则以 GPL 协议发布与维护于：

   http://dev.lemote.com/code/rt4ls

3. 移植 GDB Tracepoint 到 Cell B.E.，2007 年

该项目是将一款实时调试工具GDB Tracepoint移植到IBM公司的CELL BE多核处理器。这是作者在刚上研时参加的一个IBM全球高校比赛项目，所有项目源码以GPL的形式发布于http://tp4cell.sourceforge.net。

4. SIL4/Linux，2006 年至 2008 年

这是与西门子公司合作的一个研究型项目。项目旨在找到一种有效的方法验证Linux是否满足SIL4(Safety Integrity Level 4)。本人的主要工作是设计与实现了用于辅助Linux进行SIL4验证的SIL4Linux数据库系统，包括所有后台自动化工具、数据库与Web接口的设计与实现，该系统首页：

http://sil4linux.dslab.lzu.edu.cn

5. 基于 XtratuM 的实时容错系统研究，2008 年 8 月到 2009 年 1 月

项目期间，作者研究了实时操作系统的各种容错技术，包括主备份冗余机制（Master & Slaver），投票机制等，并基于XtratuM这一实时虚拟机(Hypervisor)设计了操作系统冗余的实时容错系统架构，并实现了一个控制步进电机的原型系统。

6. VnstatSVG：网络流量监测系统 Vnstat 的一款 Web 前端

这是作者在发表《*A CGI+AJAX+SVG based monitoring method for distributed and embedded system*》论文时展开的一个项目，项目是对论文提出的原理的具体实现。Vnstat作为一款轻量级的网络流量监测系统，非常适合在嵌入式领域的使用，但是相关的Web前端，比如Vnstat PHP frontend不方便在嵌入式，特别是分布式的嵌入式系统中使用。于是，作者提出了基于CGI+AJAX+SVG的实现方案，并基于该方案实现了VnstatSVG，该方案非常适合在大型分布式与嵌入式系统中使用。

该项目以GPL协议发布于：http://vnstatsvg.sourceforge.net

致 谢

该论文是在我的指导老师 Nicholas MC.Guire 教授和周庆国副研究员的悉心指导下完成的，在此，谨向他们致以崇高的敬意和感谢！三年来，DSLab 实验室的老师们在学术研究上给予了精心的指导，在生活中给予了无私的帮助，同学们经常进行学术讨论，相互关心和支持，他们对该文的完成提出了许多建设性的意见，在此致以深深的谢意！

在江苏龙芯梦兰科技股份有限公司实习期间，进行该项目的过程中，得到了张福新总经理的指导，也得到了其他公司同事的关心和帮助，在此一并感谢。

在提交研究成果到 Linux 自由软件社区时，得到了来自社区的 Thomas Gleixner，Ralf Baechle，Steven Rostedt，Ingo Molnar 等 Linux 内核开发人员的支持，感谢他们对 Linux 的贡献以及对其他内核开发人员的无私帮助。

还需要深深地感谢我的父母和弟妹，谢谢他们多年来对我的关心和鼓励。

最后，把真挚的谢意献给其他所有的老师、亲人、同学、朋友，感谢你们在成长的路上，一路伴我走来。

Linux实时抢占补丁的研究与实践

作者：吴章金
学位授予单位：兰州大学