第 42 卷 第 10 期 计 算 机 工 程 2016 年 10 月
Vol. 42 No. 10 Computer Engineering October 2016

·体系结构与软件技术· 文章编号：1000-3428(2016)10-0080-06 文献标志码：A 中图分类号：TP399

手持终端 Linux 实时性机制与分类调度策略研究

王铭铖，陆 阳

（合肥工业大学 计算机与信息学院，合肥 230009）

摘 要： 针对手持终端高实时性和多任务的特点，在内核中增加编译实时互斥锁、线程化中断、高精度定时器和动态时钟等实时机制。基于 RM 算法和 SCHED_FIFO 算法相结合的思想，提出任务集分类调度策略，根据截止期调度算法硬实时性高的特点，分别使用截止期调度算法与实时组调度算法调度硬实时任务与软实时任务。理论推导和实验结果表明，改进后 Linux 内核的实时性能指标符合硬实时任务的微秒级要求，并且当系统过载时能有效降低任务的截止期错失率，在 1.2～2.8 的 CPU 负载区间内平均降幅达到 18%。

关键词： 手持终端；内核实时机制；动态时钟；实时组调度；截止期调度；分类调度

中文引用格式： 王铭铖，陆 阳. 手持终端 Linux 实时性机制与分类调度策略研究[J]. 计算机工程，2016，42(10)：80-85，90.

英文引用格式： Wang Mingcheng，Lu Yang. Research on Linux Real-time Mechanism and Classified Scheduling Strategy of Handheld Terminal[J]. Computer Engineering，2016，42(10)：80-85，90.

Research on Linux Real-time Mechanism and Classified Scheduling Strategy of Handheld Terminal

WANG Mingcheng，LU Yang

（School of Computer and Information，Hefei University of Technology，Hefei 230009，China）

【Abstract】 According to the high real-time performance and multitasking features of the handheld terminal，some real-time mechanisms are added to the kernel，such as real-time mutual exclusion lock，threaded interruption，high precision timer and dynamic clock. Based on the idea of the RM algorithm combined with SCHED_FIFO algorithm，a taskset classified scheduling strategy is proposed. Considering the deadline scheduling algorithm has the feature of high hard real-time performance，deadline scheduling algorithm is used to schedule hard real-time task and real-time group scheduling algorithm is used to schedule soft real-time task. Theoretical derivation and experimental results show that the real-time performance of the improved Linux kernel meets the microsecond requirements of hard real-time tasks，and it also can reduce the tasks' deadline missing rate effectively when the system is overloaded. The average deadline missing rate is dropped by 18% when CPU load range is between 1.2 and 2.8.

【Key words】 handheld terminal；kernel real-time mechanism；dynamic clock；real-time group scheduling；deadline scheduling；classified scheduling

DOI： 10.3969/j.issn.1000-3428.2016.10.014

1 概述

随着工业 4.0 的飞速发展，工业物联网[1]中手持式终端的作用越来越重要。手持式终端[2]一般应用于现场控制、音视频数据采集、射频识别和数据实时传输等领域[3]。手持式终端系统具有以下特点：(1)单处理器系统，不用考虑处理器之间的协调问题；(2)系统应具有较高的实时性；(3)系统处理的任务集包括少量硬实时任务、大量软实时任务和一定量普通任务，要求系统在优先处理实时任务的同时要尽量保障普通任务的执行；(4)作为手持式设备系统功耗要相对较低。现有的手持式终端系统大多数基于源代码开放、裁剪方便、支持多种处理器架构、系统稳定的 Linux 内核，但 Linux 为分时系统，应用于手持式终端有以下问题：(1)实时性机制不足，存在内核不完全可抢占、中断不可

基金项目： 国家“863”计划基金资助项目(2011AA060406)；国家国际科技合作专项基金资助项目(2014DFB10060)；安徽省自然科学基金资助项目(1408085MKL80，1408085MKL79)。

作者简介： 王铭铖(1990—)，男，硕士研究生，主研方向为嵌入式系统；陆 阳，教授、博士生导师。

收稿日期： 2015-09-25 **修回日期：** 2015-11-17 **E-mail：** 574147560@qq.com

抢占和优先级反转等问题,使中断响应时间和上下文切换时间都无法满足实时任务的微秒级要求;(2)实时调度策略不足,存在实时组调度无法根据截止期属性调度任务和截止期调度无法使用 fork 的问题,使实时调度策略不完全适用于手持式终端任务集。

目前对上述问题的研究大多是在 Linux 2.6 内核中实现速率单调调度(Rate Monotonic Scheduling,RMS)算法[4]、最早截止期优先(Earliest Deadline First,EDF)调度算法[5]或最小空闲时间优先(Least Slack First,LSF)调度算法[6]等实时调度算法。但这些改进存在以下问题:(1)Linux 2.6 内核与Linux3.18 相比实时性机制较差;(2)对 Linux 2.6 内核做大量改动会破坏 Linux 内核自身的稳定性与健壮性;(3)由于 Linux 自身的优先级继承机制问题,当使用这些改进的调度算法时,会使进程无法使用 fork,这使得改进后的系统不能有效进行实际应用。

本文针对以上问题,通过分析 Linux 3.18 内核的实时性机制并结合动态时钟机制,设计一套适用于手持式终端的实时性机制。根据手持式终端任务集的特点,提出分别使用截止期调度算法调度硬实时任务、实时组调度算法调度软实时任务和公平组调度算法调度普通任务的分类调度策略,并对实时组调度算法做了相应改进。需要说明的是本文设计的实时性机制可以用于各种设备的 Linux 系统的实时性改进,但本文提出的分类调度策略不能有效地调度具有大量硬实时任务或庞大任务量的任务集,只适用于和手持式终端特点相似的任务集。

2　实时性机制设计

2.1　实时互斥锁

Linux 2.6.25 内核引入了实时互斥锁[7](rt_mutex),其结构体如下:

```
struct rt_mutex{
raw_spinlock_t wait_lock;
struct plist_head wait_list;
struct task_struct * owner;
};
```

owner 指针指向拥有该锁的进程;wait_list 是一个优先级队列。队列中的元素是一些被封装为 struct rt_mutex_waiter 的进程描述符,在 wait_list 队列中按照进程的优先级排序。

rt_mutex 采用优先级继承方式[8],将锁持有者的优先级提升至与等待该锁的最高优先级的进程一样。为了实现优先级继承,在 struct task_struct 中增加了 2 个属性:

```
struct task_struct{/* 省略无关代码 */
struct plist_head pi_waiters;
struct rt_mutex_waiter * pi_blocked_on;
};
```

pi_waiters 保存锁持有者拥有的资源阻塞的进程中优先级最高的那些进程;pi_blocked_on 用来指示进程被阻塞在哪个实时互斥锁上。

内核中存在一定不可抢占的底层临界区,必须由不可抢占的自旋锁保护。例如,硬件寄存器保护锁、调度器运行队列保护锁和其他不可抢占的自旋锁嵌套的锁等。除去上述不可抢占的底层临界区外,将其他临界区的自旋锁替换为实时互斥锁。在用实时互斥锁代替自旋锁后,高优先级进程会抢占低优先级进程对锁的使用权,提高了内核的可抢占性,同时实时互斥锁还采用优先级继承的方式,解决了优先级反转问题。

2.2　中断线程化

在实时互斥锁代替自旋锁后,为中断的线程化[9-10]提供了可行性。Linux 2.6.39 内核引入了 irq_setup_forced_threading 函数,可以通过#define force_irqthreads(true)强制使中断线程化。这样中断线程化的实现就变得非常简单,中断线程化的实现过程如图 1 所示。

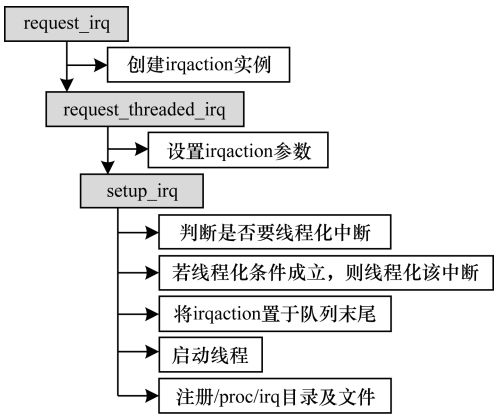

图 1　中断线程化实现过程

在中断线程化后,实时任务拥有比中断更高的优先级,因此,可以确保高优先级实时任务的运行不被中断打断。同时低优先级的任务可以被中断打断,也保证了中断能尽快得到处理。

2.3　动态时钟

Linux 内核中的时间一直都是由周期时钟提供。即使在高分辨率定时器模式下,也是通过周期时钟仿真提供周期时钟中断。该方法简单有效,但周期时钟要求系统在一定的频率下,周期性地处于活动状态。因为长时间的休眠是不可能的,这对于关注耗电量的手持式终端是不实用的,所以本文采取动态时钟来改善该问题。

动态时钟[11]是指只有在有些任务需要实际执行时,才激活周期时钟;否则会临时禁用周期时钟。采用动态时钟,在系统空闲时减少了时钟中断的次数,使系统功耗得到一定程度的降低。

Linux 内核支持低分辨率和高分辨率 2 种定时器,2 种定时器都可以实现动态时钟[12],如图 2 所示,但有些嵌入式设备的硬件不支持高分辨率定时器,所以,本文不再对定时器进行展开讨论。采取在内核编译时统一把高分辨率定时器编译进内核的方案,使支持高分辨率定时器的嵌入式设备可以使用高分辨率定时器,不支持的可以继续使用传统的低分辨率时钟。

	动态时钟	周期时钟
高分辨率定时器	高分辨率动态时钟	高分辨率周期时钟
低分辨率定时器	低分辨率动态时钟	高分辨率周期时钟

图 2 计时器的可能配置

从 Linux 2.6.39 开始,大内核锁正式被内核排除,所以,本文不再对大内核锁的抢占性进行研究。

3 实时调度算法分析

3.1 实时组调度分析

Linux 2.6.26 内核引入了实时组调度机制。调度组(task_group)可以包含任意调度类别的进程,具体有实时进程和普通进程,实时进程又可细分为截止期进程和非截止期进程。调度组的优先级与组内最高优先级进程相等。

3.1.1 实时组高度的相关数据结构

为实现实时组调度,增加了实时调度实体(sched_rt_entity)数据结构。因为在进行组调度时,被调度对象可能有 2 种:(1)调度组,如图 3 的标注1 部分所示;(2)单个进程,如图 3 的标注 2 部分所示。所以要用一个抽象的调度实体结构进行表示,具体如下:

```
struct sched_rt_entity {/* 省略无关代码 */
struct list_head run_list;
struct sched_rt_entity * back;
struct sched_rt_entity * parent;
struct rt_rq * rt_rq;
struct rt_rq * my_q;
};
```

如果调度实体代表调度组,则 my_q 指针指向该调度组的实时运行队列;如果调度实体代表单个进程,则 my_q 指针为空。

调度组结构体如下:

```
struct task_group{ /* 省略无关代码 */
struct sched_rt_entity * * rt_se;
struct rt_rq * * rt_rq;
struct rt_bandwidth rt_bandwidth;
struct task_group * parent;
struct list_head siblings;
struct list_head children;
};
```

rt_se 指向该调度组的调度实体;rt_rq 指向该调度组的实时运行队列;rt_bandwidth 结构体表示该调度组的 *rt_period_us* 和 *rt_runtime_us* 等时间参数。

每个 CPU 都对应一个根调度组(如图 3 的task_group(root)所示),根调度组的 rt_se 指针为空,rt_rq 指针指向该 CPU 的实时运行队列(rt_rq)。本文讨论适用于手持式终端的单 CPU 系统,故只有一个根调度组。实时组调度涉及的数据结构及其之间的关系如图 3 所示。

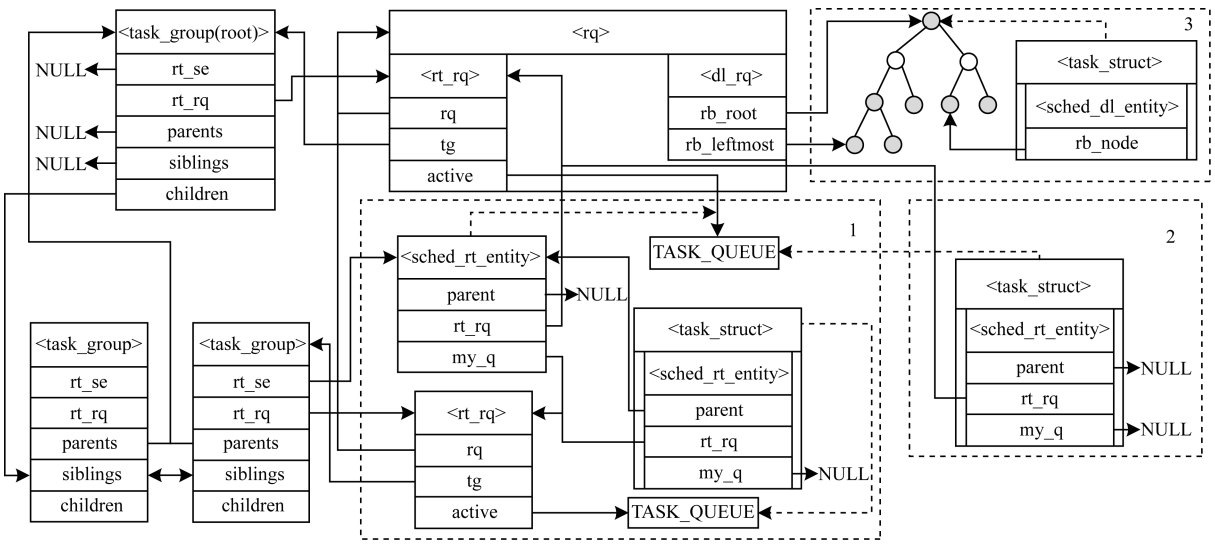

图 3 实时组调度和截止期调度的数据结构关系

由图 3 分析可知,无论分组与否,实时组调度程序总是在所有 TASK_RUNNING 状态的实时进程中选择优先级最高的进程运行。

3.1.2　相关机制

实时组调度在/proc/sys/kernel/目录下增加了 *sched_rt_period_us* 和 *sched_rt_runtime_us* 2 个系统文件,分别记录了 *sched_rt_period_us* 和 *sched_rt_runtime_us* 2 个值,表示在 *sched_rt_period_us* μs 的周期内,系统所有就绪的实时进程的运行时间总和不能超过 *sched_rt_runtime_us* μs,周期内剩余的运行时间留给普通进程运行。*sched_rt_period_us* 的系统默认值为 1 000 000 μs(1 s),*sched_rt_runtime_us* 的系统默认值为 950 000 μs(0.95 s),表示在 1 s 的周期内,系统中所有就绪的实时进程的运行时间总和不能超过 0.95 s,普通进程可获得的运行时间至少有 0.05 s。

实时组调度也为每个调度组设置了 *rt_period_us* 和 *rt_runtime_us* 2 个值,表示调度组在 *rt_period_us* μs 的周期内,组内所有就绪的实时进程的运行时间总和不能超过 *rt_runtime_us* μs,周期内剩余的运行时间留给组内的普通进程运行。上述 4 个值之间的关系分析如下:

设存在调度组 G,其子调度组集合为 $\{G_1, G_2, \cdots, G_n\}$,则有如下关系:

$$\begin{cases} \sum_{i=1}^{n} rt\_period\_us(G_i) \leqslant rt\_period\_us(G) \\ \sum_{i=1}^{n} rt\_runtime\_us(G_i) \leqslant rt\_runtime\_us(G) \end{cases}$$

若调度组 G 为根调度组,则有:

$$\begin{cases} rt\_period\_us(G) = sched\_rt\_period\_us \\ rt\_runtime\_us(G) = sched\_rt\_runtime\_us \end{cases}$$

其中,$rt\_period\_us(X)$ 和 $rt\_runtime\_us(X)$ 分别表示调度组 X 的 *rt_period_us* 和 *rt_runtime_us*。

3.1.3　实时组调度的优缺点

实时组调度的优点分析如下:(1)在未分组之前,Linux 实时调度策略在有实时进程就绪时会马上执行实时进程,很容易造成普通进程“饿死”;在分组后,每个 *rt_period_us* 周期普通进程都会得到相应的运行时间,提高了系统效率。(2)在实时进程中大多数是周期性进程,而且各进程的周期大多数不相等。例如,一个视频程序要求每 40 ms 刷新一帧,而音频程序要求每 20 ms 刷新一帧。在未分组之前,要采用统一的周期进行处理,浪费系统资源;在分组后,通过 *rt_period_us* 设置不同的周期,有效解决了此问题。

实时组调度的缺点分析如下:假设有 2 个优先级相同的调度组 A 和 B,都拥有 50% 的父调度组资源。A 的 *rt_period_us* = 100 000,*rt_runtime_us* = 10 000。B 的 *rt_period_us* = 50 000,*rt_runtime_us* = 10 000。假设 A 有一个 while(1)循环要执行 50 000 ms,若 A 先执行,则 B 会被“饿死”一个周期。

3.2　截止期调度分析

Linux 3.17 进一步扩充了进程优先级的取值范围。引入截止期进程,截止期进程的优先级小于 0(内核中设置为 -1)。使用截止期调度算法(SCHED_DEADLINE)调度截止期进程。

3.2.1　截止期调度的相关数据结构

为实现截止期进程的调度,增加 sched_dl_entity 和 dl_rq 2 个数据结构,分别如下:

```
struct sched_dl_entity{/*省略无关代码*/
struct rb_node rb_node;
u64 dl_runtime;/*最大执行时间*/
u64 dl_deadline;/*相对截止期*/
u64 dl_period;/*周期*/
u64 dl_bw;
s64 runtime;/*剩余执行时间*/
u64 deadline;/*绝对截止期*/
unsigned int flags;
};

struct dl_rq{/*省略无关代码*/
struct rb_root rb_root;
struct rb_node *rb_leftmost;
};
```

其中,dl_bw 表示该进程占用的 CPU 带宽,取值为最大执行时间和相对截止期的比值。

截止期调度涉及的数据结构及数据结构之间的关系如图 3 标注 3 部分所示。由图 3 分析可知,截止期任务按照截止期早晚在一棵灰白树上排序。调度类总是选择最左的叶节点(即截止期最早的任务)执行。基于该特点,使不完全服从传统 EDF 算法的实时任务模型的进程也可以被高效调度。

3.2.2　CBS 算法

截止期调度算法是 EDF 算法和 CBS(Constant Bandwidth Server)算法结合的调度算法。EDF 算法的说明文献很多,本文不再对其进行分析,只对 CBS 的实现机制进行分析。

设截止期任务 T_i 由一个五元组 $\{C_i, D_i, P_i, d_i, r_i\}$ 组成。其中,C_i 为最大执行时间;D_i 为相对截止期;P_i 为周期;d_i 为调度截止期(等价于 EDF 算法中的绝对截止期);r_i 为剩余执行时间。C_i,D_i 和 P_i 在进程 T_i 创建时由系统调用 sched_setattr( )进行设置。进程 T_i 的运行状态由 d_i 和 r_i 标识,它们的初值为 0。

CBS 的实现机制如下:

(1)当 T_i 就绪时,系统检测是否满足式(1):

$$\frac{r_i}{d_i - CLOCK(t)} > \frac{C_i}{P_i} \quad (1)$$

其中，$CLOCK(t)$表示系统当前时间。

(2)若 $d_i < CLOCK(t)$或式(1)成立，则更新 d_i 和 r_i 的值；否则，d_i 和 r_i 保持不变。

$d_i = CLOCK(t) + D_i$

$r_i = C_i$

(3)每个周期时钟中断或者发生进程抢占时，系统会更新 r_i的值：

$r_i = r_i - totaltime\_curr$

其中，$totaltime\_curr$ 为 T_i 本次调度运行的总时间。

(4)当 $r_i \leqslant 0$ 时，T_i 将被阻塞。被阻塞的 T_i 在 d_i 之前不可以再被调度，并将 T_i 的补充时间(*rep_time*)设置为 d_i。

(5)当 $CLOCK(t) = rep\_time$ 时，系统将更新 d_i 和 r_i的值：

$d_i = d_i + P_i$

$r_i = r_i + C_i$

通过上述机制，CBS 算法将优先调度那些带宽在其自身声明的 CPU 带宽范围内的周期性进程，保证其不错过截止期。对于非周期性进程、不定期产生的进程和试图超过其申请 CPU 带宽资源的进程都会被延期，导致这些进程可能错过截止期，但不会影响任何其他进程的执行。

现阶段 CBS 的初始带宽借用实时组调度的 *sched_rt_runtime_us* 和 *sched_rt_period_us* 参数表示，初始带宽为两者的比值(本文只讨论单 CPU 的情况)。

3.2.3 截止期调度的优缺点

截止期调度的优点分析如下：(1)截止期调度算法的实现使 Linux 内核具有根据任务截止期属性进行调度的硬实时能力。(2)通过 CBS 机制，使得使用截止期调度的各任务之间在截止期属性上相互独立。(3)CBS 机制申请系统固定的 CPU 带宽资源，保证了普通进程不会被“饿死”。

截止期调度的缺点分析如下：经典的 Linux 内核只使用优先级对进程的调度进行标识，其进程优先级的继承机制非常成熟和稳定。由于截止期调度算法增加了截止期属性，导致使用 fork 产生新进程时，子进程无法准确继承父进程的截止期属性，规定截止期进程无法使用 fork，因此截止期调度算法不能灵活地调度大批量的进程。

4 分类调度策略设计与实现

4.1 实时组调度的改进

解决实时组调度缺点的理想办法是实现 EDF 算法，但由于 Linux 自身机制原因，在使用 EDF 算法时无法使用 fork，这就大大限制了 Linux 系统的性能，因此本文使用 RM 算法与 SCHED_FIFO 算法结合的思想，改进实时组调度算法。

实时组调度策略的改进思想：(1)调度实体在插入相应的优先级队列时，根据其所属的调度组的周期参数(rt_bandwidth)，按照从小到大的顺序插入就绪队列。(2)采用 SCHED_FIFO 调度算法调度实时进程。由于 SCHED_FIFO 算法不会抢占同优先级的任务，因此改进方法不用进行可调度性处理。上述改进需要修改实时组调度插入函数 enqueue_rt_entity()，具体修改如下：

```
static void _enqueue_rt_entity(struct sched_rt_entity * rt_se,bool head)
{/*省略其他代码*/
if (queue)
{
    tmp = list_entry(queue,struct sched_rt_entity,run_list);
/*函数 tg_of_rt_se(),返回 rt_se 所属的调度组 */
while(tg_of_rt_se(tmp)→rt_bandwidth.rt_period < tg_of_rt_se(rt_se)→rt_bandwidth.rt_period)
    {
        prev = queue;
        queue = queue→next;
        tmp = list_entry(queue,struct sched_rt_entity,run_list);
    }
    list_add(&rt_se→run_list,queue);
}
else
    list_add_tail(&rt_se→run_list,queue);
}
```

改进后的实时组调度算法在一定程度上考虑了实时任务的截止期属性，降低了 3.1.3 节所述问题出现的机率，而且保留了内核原有的 $O(1)$调度框架，没有破坏调度的稳定性与高效性。

4.2 分类调度策略设计

截止期调度类(dl_sched_class)优先级高于实时组调度类(rt_sched_class)，实时组调度类优先级高于公平调度类(fair_sched_class)，如图 4 所示。

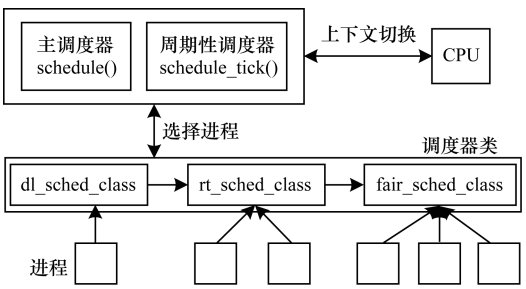

图 4 调度子系统各组件关系

根据对实时组调度和截止期调度算法的分析，

以及手持式终端具有少量硬实时任务、大量软实时任务和一定量普通任务的特点,本文提出分类调度策略。

分类调度策略的主要思想为:采用截止期调度策略对手持式终端的少量硬实时任务进行调度,最大程度地保证硬实时任务的截止期。采用实时组调度对手持式终端的大量软实时任务进行调度,在保证软实时任务优先执行的同时保证普通任务可以获得一定的执行时间,从而提高系统的性能。采用公平调度策略调度普通任务,保证普通任务能公平地获得 CPU 时间。

5　实时性能测试

本文阐述了一种实用型实时系统设计与改进方案,并且直接在相应的硬件平台上进行测试。测试的硬件平台为基于 S3C2440A 芯片的手持设备,CPU 为 ARM9,频率设置为 400 MHz,内存为 256 MB,Linux 内核版本为 Linux 3.18.14。

5.1　实时性机制的设计效果测试

本文把主动内核抢占、线程化中断、高精度定时器和动态时钟编译进内核。采用代码插桩法[13-15],分别在系统空载和满载时对中断响应时间和上下文切换时间进行测试。实验分别进行 5 000 次测试,取中断响应时间的最大值和上下文切换时间的平均值。实验结果如表 1 所示。由实验结果可知,中断响应时间和上下文切换时间都在微秒级,符合实时任务的要求。

表 1　实时性能参数对比　　μs

测试项目	空载	满载
中断响应时间	419	704
上下文切换时间	321	569

5.2　分类调度策略的效果测试

本文对分类调度策略的截止期错失率进行测试,由于实时组调度算法没有对截止期属性进行判断,因此本文采用定时器模拟截止期的方式[16-17]对所有进程的截止期完成情况进行统计,该实验中的样本和参数具体解释如下:

(1)实验样本由 100 个进程组成,并规定进程的产生时间是随机的。样本包括硬实时进程、软实时进程和普通进程。

(2)硬实时进程的优先级为 -1,软实时进程的优先级在 0 ~ 99 之间随机选取,普通进程的优先级在 100 ~ 139 之间随机选取,并且都服从正态分布。

(3)限定硬实时进程的发生概率为 0.05 左右,软实时进程的发生概率为 0.8 左右,普通进程的发生概率为 0.15 左右。

(4)负载系数(r)取值在 0 ~ 4.8 之间,本文不研究负载过重时的情况。

截止期错失率的测试结果如图 5 所示。在负载较低($r<1$)时,改进前和改进后的截止期错失率基本相同。在负载较重($1<r<4$)时,改进后的截止期错失率明显优于改进前,在 1.2 ~ 2.8 的负载区间内截止期错失率的平均降幅达到 18% 。主要原因是改进后内核的软实时进程按照周期从小到大的顺序插入到相应优先级队列,使截止期错失率有所降低。但随着负载的继续增加,其改进的效果明显下降。在正常情况下,Linux 的负载介于 0.7 ~ 2.0 之间,当负载超过 5.0 时,系统会崩溃,因此,该改进方式可行。

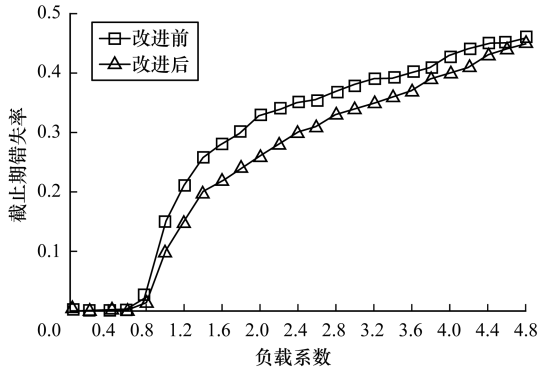

图 5　截止期错失率比较

6　结束语

本文结合 Linux 实时组调度算法和截止期调度算法的优缺点,提出分类调度策略,并且分析和设计了 Linux 内核的实时性机制。实验结果表明,实时性机制和分类调度策略改进了工业物联网下手持式终端的实时性能,具有较高的实用价值。下一步将对截止期调度算法时间属性的 fork 继承问题进行研究,以改进 Linux 内核的硬实时性能。

参考文献

[1] Xu Lida, He Wu, Li Shancang. Internet of Things in Industries: A Survey[J]. IEEE Transactions on Industrial Informatics, 2014, 10(4): 2233-2243.

[2] 曹　松,刘春煌,史　宏. 移动手持终端系统的设计与实现[J]. 中国铁道科学,2011,32(3):136-141.

[3] 宋执环,杜往泽,李　斌,等. 基于图像检测的除尘风机嵌入式控制系统[J]. 仪器仪表学报,2014,35(5):1192-1200.

[4] 田　聪,段振华. 基于命题投影时序逻辑的单调速率调度算法模型检测[J]. 软件学报,2011,22(2):211-221.

[5] Gamini A V, Tari Z, Zeephongsekul P, et al. Performance Analysis of EDF Scheduling in a Multi-priority Preemptive M/G/1 Queue[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(8): 2149-2158.

(下转第 90 页)

和多组软件的多个变体进行多次输入下的实验,证明该胎记具有良好的可信性和可靠性。在保证高可靠性和可信性的同时,本文方法还体现出 2 个优点:(1)利用二进制插桩工具 pin,不需要源码就可以对程序进行监测,使得该动态胎记的使用范围更加广泛;(2)利用基本块自身的指令数能很好地表达程序语义。本文对 2 种胎记的相似度求均值作为最终胎记的相似度,下一步将对 2 种胎记相似度的加权系数进行研究,以使得可靠性和可信性均达到最佳效果。

参考文献

[1] Tamada H, Nakamura M, Monden A, et al. Java Birthmarks-detecting the Software Theft [J]. IEICE Transactions on Information and Systems, 2005, E88-D(1): 2148-2158.

[2] Ginger M, Christian S C. K-gram Based Software Birthmarks[C]//Proceedings of ACM Symposium on Applied Computing. Santa Fe, USA: ACM Press, 2005: 314-318.

[3] 邓小鸿,拜亚萌,黄 斌,等. 一种可检测 Java 程序盗版的动态胎记技术[J]. 计算机工程与应用, 2010, 46(17): 69-71, 84.

[4] 陈 林,刘粉林,芦 斌,等. 基于 k-gram 频数的静态软件胎记[J]. 计算机工程, 2011, 37(4): 46-48.

[5] Liu Fenlin, Lu Bin, Chen Lin. A Software Birthmark Based on Weighted k-gram[C]//Proceedings of 2010 Conference on Intelligent Computing and Intelligent Systems. Washington D. C., USA: IEEE Press, 2010: 400-405.

[6] 马世鑫,刘粉林,罗向阳,等. 基于互信息的 k-gram 软件胎记选取[J]. 计算机工程, 2012, 38(22): 43-46.

[7] 罗养霞,房鼎益. 基于聚类分析的软件胎记特征选择[J]. 电子学报, 2013, 41(12): 2334-2338.

[8] 李菲菲,周清雷. 基于信息增益的软件特征技术[J]. 计算机应用研究, 2014, 31(7): 2082-2084, 2087.

[9] Myles G, Collberg C. Detecting Software Theft via Whole Program Path Birthmarks[C]//Proceedings of the 7th International Conference on Information Security. Washington D. C., USA: IEEE Press, 2004: 404-415.

[10] Tamada H, Okamoto K, Nakam M, et al. Dynamic Software Birthmarks to Detect the Theft of Windows Applications[C]//Proceedings of International Symposium on Future Software Technology. Washington D. C., USA: IEEE Press, 2004: 125-136.

[11] 孙 光,樊晓平,刘钟理. 扩展频繁执行路径上的 n-gram 软件胎记[J]. 信息网络安全, 2013(2): 57-60.

[12] 韩兰胜,高昆仑,赵保华,等. 基于 API 函数及其参数相结合的恶意软件行为检测[J]. 计算机应用研究, 2013, 30(11): 3407-3410, 3425.

[13] 范 铭,刘 均,郑庆华,等. 基于栈行为动态胎记的软件抄袭检测方法[J]. 山东大学学报:理学版, 2014: 9-16.

[14] Lim H, Park H, Choi S, et al. Detecting Theft of Java Applications via a Static Birthmark Based on Weighted Stack Patterns[J]. IEICE Transactions on Information and Systems, 2008, E91-D(9): 2323-2339.

[15] Lim H, Park H, Choi S, et al. A Method for Detecting the Theft of Java Programs Through Analysis of the Control Flow Information[J]. Information and Software Technology, 2009, 51(9): 1338-1350.

[16] 周清雷,张凤萍. 基于程序数据属性的联合软件特征[J]. 计算机应用与软件, 2014, 31(4): 308-311.

编辑 索书志

(上接第 85 页)

[6] 任小西,赵公怡. 基于动态抢占阈值的 LSF 调度算法[J]. 计算机工程, 2012, 38(4): 275-277, 280.

[7] Yan Liping, Song Kai. Improvement of Real-time Performance of Linux 2.6 Kernel for Embedded Application[C]//Proceedings of International Forum on Computer Science-technology and Applications. Washington D. C., USA: IEEE Press, 2009: 71-74.

[8] 凌云卿. 基于 Linux 的嵌入式实时系统的研究与实现[D]. 武汉:华中科技大学, 2012.

[9] He Jianfeng, Li Yufeng, Zhang Wei, et al. Real-time Optimization and Application of the Embedded ARM-Linux Scheduling Policy[C]//Proceedings of International Conference on Information Technology. Washington D. C., USA: IEEE Press, 2011: 134-138.

[10] 姜 南. Linux 系统的实时性研究[D]. 长春:吉林大学, 2010.

[11] Wolfgang M. 深入 Linux 内核架构[M]. 郭 旭, 译. 北京:人民邮电出版社, 2010.

[12] 尹嘉鹏,徐志祥. 针对少实时任务应用的嵌入式 Linux 改进[J]. 计算机工程, 2013, 39(10): 49-52.

[13] Koh Jae-hwan, Choi Byoung-wook. Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions[J]. International Journal of Control and Automation, 2013, 6(1): 235-245.

[14] 吴 讯,马 媛,董勤鹏. 实时操作系统实时性能测试技术研究[J]. 系统仿真学报, 2013, 25(2): 313-316.

[15] 张晓龙,郭锐锋,陶耀东,等. Linux 实时抢占补丁的研究及实时性能测试[J]. 计算机工程, 2014, 40(10): 304-307, 314.

[16] Li Yang, Zhu Qingyan. A Kind of New Real-time Scheduling Algorithm for Embedded Linux[J]. TELKOMNIKA Indonesian Journal of Electrical Engineering, 2014, 12(6): 4444-4450.

[17] 黄姝娟,朱怡安,李兵哲,等. 具有依赖关系的周期任务实时调度方法[J]. 计算机学报, 2015, 38(5): 999-1006.

编辑 陆燕菲