Linux动态调度算法的研究与实现

洪 伟[1] 苏晓龙[1] 王香婷[2]

(1.中国矿业大学计算机学院，江苏 徐州 221116;
2.中国矿业大学信电学院，江苏 徐州 221116)

【摘 要】Linux因具有内核源码公开、性能稳定、兼容UNIX、支持多种处理器、网络功能强、安全性高、内核可以剪裁等一系列优点，正迅速进入实时控制领域。但Linux天生并不是真正的实时操作系统，所以必须对其进行实时化提升。文章将EDF动态调度算法引入Linux2.6.25.8内核，并把修改后的内核经过编译后移植到嵌入式开发板TQ2440开发板中，对标准内核和修改后的内核分别进行轻负载和重负载的测试和对比。实验表明改进的Linux内核的实时性能比标准的Linux内核有较大的改善。

【关键词】实时操作系统；实时调度；EDF调度算法
【中图分类号】TP316.2 【文献标识码】A 【文章编号】1008-1151(2010)07-0028-03

(一) 问题的提出

Linux内核的改进是相当频繁的，几乎每个月都在变。自从1991年推出第一个版本Linux 0.01后，至今已有19年的历史。在实时系统中，任务调度策略是内核设计的关键部分，如何进行任务调度，保证各个任务能按要求完成是实时操作系统研究的一个重要领域。目前，Linux最新的正式发布版本是2.6版。在实时性能方面linux2.6已经做了一定的改进，O(1)调度能够保持调度时间的确定性，实时调度性能得到了提高，但是对实时进程的调度算法并没有做出改进，还是沿用linux2.4内核的SCHED_FIFO和SCHED_RR两种调度策略。

静态优先级调度算法还是使实时进程以固定优先级方式进行调度，但是在实时应用中，实时进程的优先级往往与其运行时间、等待时间等时间因素密切相关，也就是说，实时进程的优先级需要随时间的变化而发生变化，系统内核需要为实时进程计算动态优先级。目前内核中的静态优先级调度算法是不能满足这个要求的，所以限制了系统的实时调度性能。

(二) EDF算法

EDF算法，即最早截止期限优先算法，是一种动态优先级的调度算法。算法将最高优先级分配给具有最早截止期的周期任务。当有多个任务可供执行时，具有最早截止期的周期任务最先得到执行权。其中的截止期是一个随着任务执行而动态改变的时间段。因此，EDF算法调度的任务的优先级也是随时间动态变化的。对于每一个可调度时刻，将根据任务的最早截止期限，动态改变任务的优先级。对于一个给定的任务集，当且仅当CPU利用率满足下面条件，可应用EDF调度：

其中C_i为每个P_i任务在每个周期的最大执行时间，T_i为P_i任务的运行时间。C_i/T_i表示CPU的利用率。

基于上述的2.6内核调度系统存在的不足，本文将对Linux的调度器进行改进，采用引入具有动态优先级性能的EDF调度算法，达到进程的动态优先级随时间和相对截止时间等属性动态变化的目的，进一步提高系统的实时性能。

(三) 改进的Linux2.6.25内核调度算法

$$\sum_{i=1}^{n}\frac{C_i}{T_i}\le 1$$

1. 改进的算法的设计

EDF算法在不破坏O(1)调度特性的基础上进行改进，具体原理如下：

(1) 进程调度的动态性体现在相对截止期的不断变化上，即在同一优先级队列中遍历所有的实时进程按照相对截止期从小到大进行排序，而非原来的FIFO形式。这样，新的实时进程调度的依据变为进程的优先级越大，越先得到调度；优先级相同，相对截止期越小，越先得到调度。

(2) Linux2.6系统的结构以及next候选进程的选取方法，实时进程的优先级prio属性值一经初始设定后保持不变，但实施进程的优先级将不再由prio完全决定，而是由prio和Dt两属性共同决定。

(3) 原系统将进程插入到优先级队列中的插入函数enqueue_rt_entity()，是将进程插入到相应优先级队列的末尾。而在改进系统中，由于实时进程在优先级队列中是按照Dt大小进行排列的，因此该函数将不再适用于实时进程，必须为实时进程编写相应的入队函数enqueue_rt_entity()。

2. 数据结构的扩展

由于EDF调度算在调度过程中需用到绝对截止期、剩余时间和任务执行时间这些动态的属性。

(1) task_struct结构体中增加上述属性后如下所示：

```
typedef struct task_struct      /*Append for EDF scheduler */
unsigned long Deadline_time ;  /*relative deadline*/
unsigned long Rest_time;       /* rest case computation time*/
```

剩余执行时间Rest_time (初值为估计执行时间)；

(2) 原有系统的将进程插入到优先级队列中的插入函数

【收稿日期】2010-03-19

【作者简介】洪伟(1985—)，女，安徽宿州人，中国矿业大学计算机科学与技术学院2007级研究生，研究方向为嵌入式应用；苏晓龙(1952—)，男，江苏徐州人，中国矿业大学计算机学院副教授，研究方向为计算机网络，自动化理论；王香婷(1952—)，女，河北保定人，中国矿业大学信电学院教授，研究领域为自动化理论。

enqueue_rt_entity()是将进程插入到相应优先级队列的末尾，原代码在/kernel/sched_rt.c 中：而在改进系统中，实时进程按照在优先级队列中是根据相对截止期从小到大进行排序，我们重新编写相应的入队函数 enqueue_rt_entity ()，实现实时进程按照空闲时间大小插入相应优先级队列的功能。具体做法：比较实时任务的固定优先级，如果优先级相同，则按照相对截止期从小到大进行排序扩展后的函数代码及说明注释如下：

```
static void enqueue_rt_entity(struct sched_rt_entity *rt_se)
{
 struct rt_rq *rt_rq = rt_rq_of_se(rt_se);
 struct rt_prio_array *array = &rt_rq->active;
 struct rt_rq *group_rq = group_rt_rq(rt_se);
 if (group_rq && rt_rq_throttled(group_rq))
      return;
      struct list_head *t;
     struct task_struct *q;
    t=array->queue + rt_se_prio(rt_se);
   if(t) /*如果该优先级数组不为空，需要确定插入位置*/
   {
     q=list_entry(t,task t,run list);
    while(p->current-Deadline>q->current-Deadline){
          prev=t;
          t=t->next;
q=list_entry(t,task_t,run_list); }
_list_add(&p->nm list,prev,t);
}else
 list_add_tail(&rt_se->run_list, array->queue + rt_se_prio(rt_se));
        //将进程插入队尾
 __set_bit(rt_se_prio(rt_se), array->bitmap);
 inc_rt_tasks(rt_se, rt_rq);}
```

（3）实时进程时间属性的更新

原系统中时间属性处理比较简单，体现在函数 task_tick_rt 中，由于 EDF 算法是一种动态调度算法，主要体现在决定任务优先级的运行时间、相对截止期在任务执行过程中是不断变化的。因此，在任务调度过程中必须对任务的运行时间、相对截止期等动态属性进行调整。

所以在改进系统中，当前进程的剩余执行时间 Rest_time 随着时钟嘀嗒而相应的减少，其相对截止期 Deadline_time 不变其他就绪进程剩余执行时间 Rest_time 不变，但相对截止期 Deadline_time 却随着时钟嘀嗒而相应的减少。因此，在时钟中断函数 kernel/sched_rt.c/ task_tick_rt 中，需要增加相应的处理代码。

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
 update_curr_rt(rq);
 watchdog(rq, p);
    p->Rest_time —; /*更新当前进程 p 的剩余执行时间*/
     for(i=0; i<100; i++){/*0--99 为实时进程的优先级范围*/
for(t=rq->active->queue[i]; t->next; t=t->next)
{
q=list_entry(t,task_t,run_list);
q->Deadline_time--;
}
/*      * RR tasks need a special form of timeslice management.
  * FIFO tasks have no timeslices.
  */
 if (p->policy != SCHED_RR)
      return;
 if (--p->rt.time_slice)
      return;
 p->rt.time_slice = DEF_TIMESLICE;
 /*
  * Requeue to the end of queue if we are not the only element
  * on the queue:
  */
 if (p->rt.run_list.prev != p->rt.run_list.next) {
      requeue_task_rt(rq, p);
      set_tsk_need_resched(p);
 }
```

（四）测试环境

本次测试是把改进后的内核移植到天嵌公司出品的 TQ2440开发板上。TQ2440是一款基于三星公司ARM9处理器 S3C2440A,支持ARM-Linux、Windows CE等操作系统的嵌入式硬件平台。平台的主要硬件资源有:一片64M SDRAM,一片64M Nand Flash,一片1M Nor Flash,一个串口COM0,一个USB 接口,一个标准JTAG接口,一个水晶头网线等等。平台支持 Linux2.6.25内核版本，32bit数据总路线，SDRAM时钟频率高达100MHz MB

1.搭建交叉开发环境

首先要有主机开发环境，通常采用Linux操作系统。在本文中，通过在Windows XP上安装VMware虚拟机软件，然后在虚拟机中安装Linux达到要求。其中VMware版本是6.0.3 build-80004英文版，Linux版本是RetHat9.0。接下来就要在Linux操作系统中安装交叉开发工具链。

从网站上下载 EABI-4.3.3_EmbedSky_20091210.tar 解压此文件到根目录下用 tar.xvf EABI-4.3.3_EmbedSky_20091210.tar -C/，解压出 opt/EmbedSky/4.3.3/bin。

把交叉编译器的路径加入到 PATH，修改/etc/bashrc 文件中 PATH=/ opt/EmbedSky/4.3.3/

bin:$PATH,保存后编写一个简单的helloworld程序验证工具安装是否正确。源码保存在/root 下，直接执行 arm-linux-gcc hello.c -o hello 进行编译，运行后得到可执行 RAM 版的文件，使用 file 命令查看文件属性以确认交叉工具链是否安装成功。

2.内核的配置和编译

（1）从 ftp://ftp.kernel.org/pub/linux/kernel/v2.6/下载 linux-2.6.25.8.tar.bz2 版本；

（2）把该文件放到根目录下面，用#tar.jxvf.linux-2.6.27.9.tar.bz2 命令解压缩该文件；把文件源码修改后，因为要移植到 ARM 体系结构的开发板上，所以还得在系统中添加对 ARM 的支持，进入内核源码，修改 Makefile 文件 ARCH?=(SUBARCH) 和 “ CROSS_COMPLILE?== 将其修改为 ARCH=arm 和 CROSS_COMPILE=arm-linux-gcc，然后保存。

（3）进入 Linux 源码的根目录，执行 make menuconfig 进入内核配置界面。在配置中主要选择处理器的类型和目标板的类型，这样才会编译针对目标平台的源码。其次还要选择需要的驱动程序。

```
（4）#make                    /*编译文件*/
（5）#make modules            /*编译模块*/
```

(6) #make modules_install /*安装模块*/

(7) #make install /*安装文件*/

(8) 把 usr//src/linux-2.6.25.8/arch/arm/boot/bzImage 复制到/root 目录下，使用串口把开发板与主机相连，用 USB 拷贝所需要的 bootloarder，bzImage，根文件系统到 NAND FLASH 中，然后重启开发板。

3. 测试程序设计

下面分别针对标准内核和实时内核进行一定次数的测试，并将测试结果进行比较。

测试程序的伪代码如下，num 即是所求值：

```
for(m=0;m<6000;m++)
{
gettimeofday(&t1,NULL);      /*调度前时间点*/
usleep(Tus);                 /*期望的睡眠时间*/
gettimeofday(&t2,NULL);      /*调度后时间点*/
num=t2 - t1 - T;             /*num 为调动延迟*/
}
```

4. 测试结果

轻负载：Linux 操作系统只是运行了初始化进程、shell 进程。表 1 所示

重负载：在重负载的情况下，拷贝一个大容量文件。表 2 所示

表 1 轻负载的测试结果

测试对象	最长调度延迟时间(us)	平均调度延迟时间(us)
标准 Linux 内核	1890	890
改进 Linux 内核	385	160

从表 1,2 可以看出标准的 Linux 在轻负载下的最长调度误差是 1890us，在重负载下更是达到了 24510us，而我们改进的 Linux 内核不论在轻负载下还是在重负载下是最长调度延迟都在 470us 以下。所以改进的 Linux 内核的实时性能比标准的 Linux 内核有较大的改善。

表 2 重负载测试结果

测试对象	最长调度延迟时间(us)	平均调度延迟时间(us)
标准 Linux 内核	24510	9200
改进 Linux 内核	467	219

（五）小结

本章首先分析了 Linux2.6 内核实时性不强的关键因素，为引入动态调度算法 EDF 做准备；其次结合 EDF 算法的原理改进了 Linux2.6.25.8 内核的进程调度，对 Linux 调度算法进行设计，以及数据结构进行扩展；最后把改进的内核成功移植到嵌入式开发板(TQ2440)，并对内核进行实时性能的测试，与标准的内核进行比较，结果表明改进后的内核具有较好的实时性能，性能较改进前有了一定的提高。

【参考文献】

[1] 宋凯,甘岚等.嵌入式 Linux 内核实时性研究及改进[J].微计算机信息,2008,9-2:16-18.

[2] 吴娇梅,李红艳等.改善嵌入式 Linux 实时性能的方法研究[J].微计算机信息,2006,1-2:72-74.

[3] 刘谦.基于 LINUX 的调度机制及其实时性研究[D].四川:西南交通大学,2007.

[4] Chetto H, Chetto M. Some results of the earliest deadline scheduling algorithm[J]. IEEE Trans On Software Engineering.1989,15(10):1261-1269.

[5] 陈媛,杨武.面向用户的进程调度策略研究与实现[J]. 计算机工程,2008,34(10):78-82.

(上接第 47 页)以看出，当加入水印后，PSNR 值有所下降，但对视频质量的影响比较小，仍可有效保证视频质量，满足了水印的不可见性。

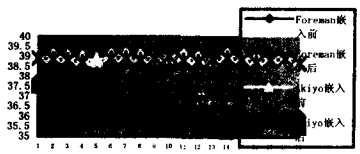

图 4 Foreman、Akiyo 视频嵌入前和嵌入后的 PSNR 值

2. 水印的鲁棒性分析

下面是使用 Foreman 视频序列作为测试视频序列，将嵌入水印后的 Foreman 视频序列后再进行 H.264 编码压缩，重编码后提取的水印图像如图 5 所示。在经历一次重编码后仍可提取较清晰可辨的水印图像，可见本算法有较高的抗 H.264 重编码攻击的能力。

图 5 重压缩攻击后提取的水印图像

（四）总结

本文对提出的基于 H.264 标准的视频水印算法进行仿真实验，由仿真实验得到的数据可以看出，本算法在保证视频质量的前提下，嵌入的水印序列具有较好的不可见性，同时对重压缩攻击等具有一定的鲁棒性，并实现了盲检测。

算法在 H.264 视频的每个 I 帧中都嵌入相同的水印序列，不修改 P 帧和 B 帧，而 I 帧不可以被跳跃或删除，即使视频受到帧删除或帧重组攻击仍可提取出鲁棒水印序列，从而说明本算法对帧编辑处理具有鲁棒性。

【参考文献】

[1] Kim J, BENTLEY P J.Negative selection and niching an artificial immune system for network intrusion detection [C]//Genetic and Evolutionary Comutation Conference. Orlando, Florida:[s.n], 1999: 149-158.

[2] F.Hartung and B.Girod. Watermarking of uncompressed and compressed video[J]. Signal Proeessing, 1998, 66(8): 283-301.

[3] Jung Y J, Kang H K, Yong M R. Novel Watermark Embedding Technique Based on Human Visual System. In: Proc of SPIE Series. Tokyo, Japan: SPIE Press, 2001,475-482

[4] Watson, A., "DCT quantization matrices visually optimized for individual images", Proceedings of SPIE, Vol. 1913, 1993, pp.202-216.

[5] D.Marpe, H. Schwarz, G. Blattermann,et al. Context-based Adaptive Binary Arithmetic Coding in JVT/H.26L. IEEE International Conference on Image.