第47卷第8期 中国科学技术大学学报 Vol.47,No.8
2017年8月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Aug. 2017

文章编号:0253-2778(2017)8-0635-09

一种 Linux 用户态实时多任务调度框架

张 旭[1,2,3],顾乃杰[1,2,3],苏俊杰[1,2,3]

(1. 中国科学技术大学计算机科学与技术学院,安徽合肥 230027;
2. 安徽省计算与通信软件重点实验室,安徽合肥 230027;
3. 中国科学技术大学先进技术研究院,安徽合肥 230027)

摘要:Linux 内核调度器的调度开销巨大,无法满足实时应用需求.为此设计并实现了基于多核 Linux 的用户态实时多任务调度框架 ULight.ULight 共包括三个核心模块:多任务调度模块、定时器模块以及用户态中断处理模块.多任务调度模块在 Linux 用户态提供基于优先级可抢占的实时多任务调度方案,旨在减少任务调度和切换开销;定时器模块则为多任务调度提供高精度的定时服务,以支持分时调度和任务休眠,并提供更多的抢占点;用户态中断处理模块通过在内核态和用户态之间构造中断处理的快速通道,使用户态任务可以直接处理硬件中断,保证中断处理的实时性和高效性.实验表明,ULight 的任务切换效率明显优于 Linux 的线程切换效率;定时系统可以提供精度为 20 μs 的稳定的定时服务;用户态中断处理模块能够在用户态完成对硬件中断的快速响应.

关键词:多任务调度;实时性;高精度定时器;用户态中断处理

中图分类号:TP301 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2017.08.002

引用格式:张旭,顾乃杰,苏俊杰. 一种 Linux 用户态实时多任务调度框架[J]. 中国科学技术大学学报,2017,47(8):635-643.

ZHANG Xu, GU Naijie, SU Junjie. A real-time multi-task scheduling framework in Linux user space [J]. Journal of University of Science and Technology of China, 2017,47(8):635-643.

A real-time multi-task scheduling framework in Linux user space

ZHANG Xu[1,2,3], GU Naijie[1,2,3], SU Junjie[1,2,3]

(1. *School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*;
2. *Anhui Province Key Laboratory of Computing and Communication Software, Hefei* 230027, *China*;
3. *Institute of Advanced Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: Task scheduling in Linux kernel has tremendous overhead, and thus cannot fulfill the requirement of real-time applications. ULight, a real-time multi-task scheduling framework running in Linux user space was proposed to conquer this problem. ULight consists of three core modules: multi-task scheduling module, timer module and user-mode interrupt handling module. The multi-task scheduling module provides priority-based preemptive scheduling in Linux user space to reduce the overhead of task switch and scheduling; the timer module introduces a high-resolution timer system to support time-sharing scheduling, enable task sleep and increase preemption points; the user-mode interrupt handling module builds up an channel between kernel and user space, which enables user-mode threads to handle interrupts

收稿日期:2016-09-11;**修回日期**:2017-04-11

作者简介:张旭,男,1990年生,博士研究生,研究方向:计算机网络协议设计、网络服务操作系统优化. E-mail: jessexz@mail.ustc.edu.cn

通讯作者:顾乃杰,博士/教授. E-mail: gunj@ustc.edu.cn

directly and efficiently in Linux user space. The experiment results show that, ULight brings much less overhead than Linux Pthread in terms of task scheduling; and keeps the precision of the timer stable within 20 μs; and can response to interrupts rapidly in user space.

Key words: user-mode task scheduling; real-time; high-resolution timer; user-mode interrupt handling

0 引言

互联网后台服务系统的任务调度效率一直是影响互联网应用响应速率的主要因素之一,对用户体验产生很大影响.此外,军事、航天航空等高尖端技术中同样存在很多对实时性要求很高的应用[1].实时系统一般具有如下特性:较小的中断延迟和调度延迟、可抢占性以及高精度的时间计量[1].主流的实时操作系统包括VxWorks[2]、LynxOS[3]等,这些实时操作系统的进程调度延迟和中断延迟很低,满足实时应用的要求,然而它们多是商用的非开源系统,软件的开发和使用成本很高,而且支持的硬件平台都有一定限制,使得在其上开发的软件大多不具可移植性[1].Linux是一个完全开源的操作系统,具有良好的稳定性和可靠性,开发使用成本较低.此外,Linux支持多种体系结构的处理器,使得软件具有很强的可移植性;而且Linux上有完善的模块机制和编译配置机制,具有很高的可定制性[4],因此Linux常作为服务操作系统的首选,然而Linux毕竟是为桌面分时系统设计的,更注重公平性和交互性,任务调度性能和实时性相对不足.

Linux 2.6引入了$O(1)$的调度器[5],使得调度时间和进程数无关,然而Linux调度器始终是运行在内核态的,调度性能仍然受到大内核锁、自旋锁以及中断屏蔽等因素的影响[5],实时性无法保证.更重要的是,所有应用的服务进程都是运行在用户态的,这些进程在切换前必须陷入内核态,由调度器选择运行下一个运行进程,完成进程切换后再从内核态切换回用户态继续运行.进程切换自身需要完成页表更新、堆栈更新、硬件上下文更新等复杂操作[5],再加上内核态和用户态的来回切换,无疑严重影响了任务调度效率.

改善Linux调度实时性的主流方法有三种:中断线程化方法[6]、双内核架构[7]以及用户态任务库[8].中断线程化是将中断处理程序作为内核线程运行,并给中断处理线程赋予不同的实时优先级,与其他实时进程一起被调度,目的是减少处理中断时的中断屏蔽时间.由于硬件中断的处理代码非常繁杂,线程化代价很大且会影响部分硬件的响应速率,使得这种方法在实际应用中受到很多限制.双内核架构的原理则是实现一个微型实时内核,并把原有的Linux系统作为普通进程运行在实时内核之上.双内核结构的主要产品包括RTLinux[9]、RTAI[10]等.双内核机制使得系统的实时性有很大提高,然而这种方式割裂了Linux系统本身与实时进程的联系,在很大程度上增加了实时应用开发的难度.相比之下,用户态任务系统不会对原有的Linux内核作出太多的改变,更易应用部署且兼容性更强,因而出现了很多优秀的用户态任务系统,如Pth[11]、Libtask[12]等.Pth采用基于优先级的事件驱动的非抢占调度,且支持多核负载均衡,但该任务库采用的非抢占调度算法和过大的任务切换开销使其并不适合实时应用.Libtask则采用简单的先进先出的非抢占式调度,使得任务切换开销较小;然而Libtask过于简单的非抢占式调度并不满足实时应用的需求.此外现有的用户态任务系统大多不提供定时服务和用户态中断处理,应用场景受到了极大限制.

本文设计和实现了一种运行于多核平台下的Linux用户态实时多任务调度框架ULight.该调度框架共包括三个核心模块:多任务调度模块、定时器模块以及用户态中断处理模块.实验表明,ULight的任务切换效率明显优于Linux的线程切换效率;定时系统可以提供精度为20 μs的稳定的定时服务;用户态中断处理模块可以高效迅速地响应硬件中断.

1 ULight框架设计

如图1所示,ULight实时多任务调度系统共包括三个模块:多任务调度模块、定时器模块以及用户态中断处理模块.多任务调度模块运行在用户态,定时器模块运行在内核态,而用户态中断处理模块则贯穿内核态和用户态.定时器模块为多任务调度模块提供基本的定时服务,以支持任务的休眠操作,而休眠操作为任务调度提供了抢占点;而且定时器模块可以提供用于模拟时钟中断的定时信号,以支持任务的分时调度.多任务调度模块和定时器模块通

过共享内存相互传递信息，而定时器模块会在有定时器到期时向多任务调度模块发送指定信号.用户态中断处理模块则通过搭建内核态和用户态之间的中断处理快速通道，将中断传递给多任务调度模块，并由多任务调度模块中的用户态线程中完成对中断的最终处理.

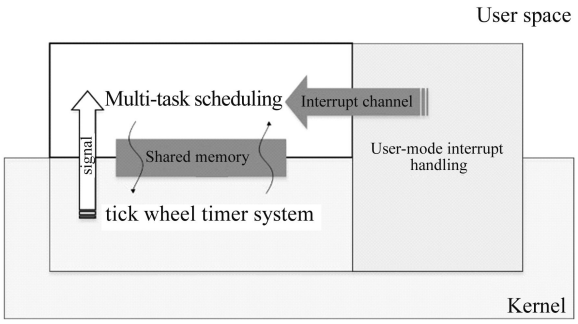

图 1　ULight 框架设计示意图

Fig.1　Architecture of ULight framework

2　多任务调度

多任务模块启动时，会创建一个用户态进程作为基础进程，然后根据处理器的核数 N 在该进程中创建(N-1)个线程，本文称之为基础线程.为了提升多核平台的资源利用率和系统运行性能，每一个基础线程都会绑定在一个的独立的处理器核上.这样可以提升数据局部性，有利于提升缓存命中率.需要注意的是，并不是所有的处理器核上都会绑定基础线程.由于定时器系统工作任务繁重，所以 ULight 会预留一个处理器核供定时器模块使用，图 2 为 4 核处理器上的多任务调度模块的运行架构示意图.基础线程之间有全局的锁和核间信号量；基础线程内部也有独立的核内信号量，用于控制任务对资源的互斥访问.每一个任务都有独立的 ID，所有的任务都运行在基础线程内.为了保证调度效率，创建后的任务不会被 ULight 动态迁移，但用户可以根据需要使用 ULight 提供的程序接口自行迁移任务.当基础线程内没有需要运行的任务时，就会简单地让 CPU 休息，以节省功耗.ULight 中所有的基础线程都被设置为 Linux 实时线程，以保证任务的执行效率和对中断的响应速率.综上，ULight 启动时需要初始化的资源主要包含两个部分：

(Ⅰ)全局资源，由基础进程完成初始化.全局资源包括所有的基础线程，任务 ID 的管理单元，用于核间任务迁移、任务互斥的全局锁和核间信号量以及用于构造分时调度时间片的周期触发定时器等.

(Ⅱ)核内资源，由基础线程完成初始化.每一个处理器核(基础线程)都有自己独立的调度体系，继而拥有独立的调度管理结构，包括各种任务状态队列及核内信号量等.对于普通任务来说，基础线程相当于一个虚拟的处理器核.

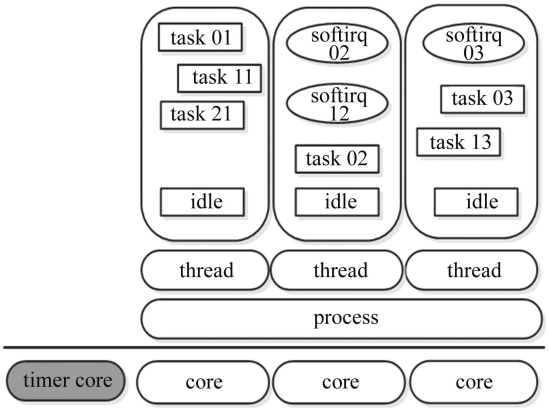

图 2　ULight 多任务运行架构

Fig.2　Muti-task running architecture

2.1　任务状态队列

每个基础线程都有独立的调度体系，都会创建 4 种不同的任务状态队列，以应对 4 种不同的任务状态：

(Ⅰ)就绪队列.ULight 支持 64 个优先级，每一个优先级都有相应的就绪队列.如图 3 所示，就绪队列以双向循环链表的形式组织任务描述符，并使用一个长度为 64 的数组存储就绪队列的队列信息，优先级的值越小表示任务优先级越高.图中灰色的任务表示该就绪队列中上一次得到运行机会的任务，队列信息中存储着指向该灰色任务的指针，如果指针为 NULL 说明该就绪队列中的现有任务都没有得到运行机会.每当需要调度时，ULight 会根据优先级位图调度算法[13]找出优先级最高的有任务的

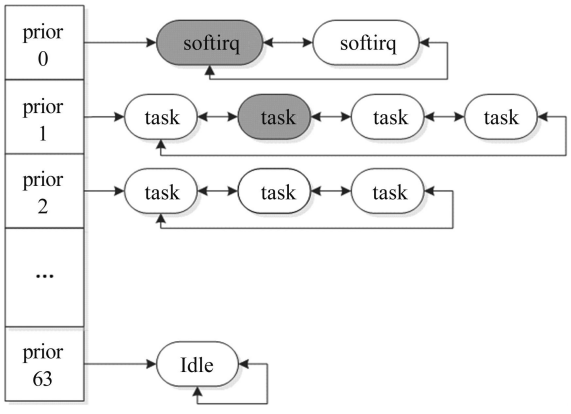

图 3　就绪队列

Fig.3　Running queue

就绪队列;如果该队列中没有灰色任务则将该队列的第一个任务取出运行,否则将灰色任务的下一个任务取出运行,这样就保证了所有优先级相同的任务都能得到平等的运行机会.优先级最低的任务为 idle,当处理器核上没有其他任务处于就绪状态时,idle 任务就会得被调度运行,该任务使基础线程处于挂起状态,减少系统功耗的同时使其他不属于 ULight 系统的程序有运行的机会.

(Ⅱ)休眠队列.为等待某一事件的发生或状态的转变,有些任务会主动休眠.ULight 的休眠队列是一个优先队列,任务会根据其睡眠到期的时间被插入休眠队列的相应位置.

(Ⅲ)阻塞队列.当任务被互斥操作阻塞时,就会放弃 CPU 的控制权,进入阻塞队列.等到互斥量释放时,系统就会将任务从阻塞队列中取出并放入就绪队列.

(Ⅳ)死亡队列.任务结束运行时被放入死亡队列并触发任务调度.系统在每次任务调度时会检查死亡队列中的任务所占用的资源是否被其他运行任务使用,以便及时清理任务所占空间.

以上 4 种不同的运行队列已经基本满足实时任务的运行状态需求,同时避免了不必要的内存开销和队列管理开销.

2.2 抢占与轮询

ULight 中抢占的时机主要包括以下 4 种:①为任务创建.系统在任务创建后会触发任务调度,如果运行任务的优先级低于新创建任务的优先级,则新任务会抢占运行;②为互斥量释放.互斥量释放时会将被阻塞的任务从阻塞队列中取出,取出后的任务可以抢占低优先级的任务;③为休眠结束.系统将任务从休眠队列中取出后触发任务调度;④为用户态中断处理.该过程中创建的软中断可以抢占普通任务的执行权.

为了保证同一优先级的任务有相同的运行机会,ULight 采用了分时轮询机制,以保证同一优先级的所有任务都有公平的运行机会.时间片的长度为定时系统精度的整数倍,与 Linux 时间片长度保持一致.

2.3 任务调度与切换优化

任务调度与切换开销主要包含两个部分:调度算法的执行和任务上下文的保存与恢复.调度算法执行过程的主要开销在于任务查找.ULight 将就绪队列的数目固定在 64,使用位图算法,构造两级 8 比特位的位图映射表实现对优先级最高的有任务的就绪队列的快速查找.第一级共 1 个字节(即 8 个比特位),每个比特位都对应第二级中的 1 个字节.图 4 给出了优先级位图调度算法的执行示例.首先读取第一级字节,得到的结果是 10,通过查询位图映射表可知该字节中被置位的最小的比特位为 1;读取比特位 1 对应的第二级字节,得到的结果是 100,通过查询位图映射表可知被置位的最小的比特位为 2;所以这种情况下优先级最高的有任务的就绪队列的优先级为 10(即 $8\times1+2$).找到正确的就绪队列后,根据队列信息中存储的灰色任务的位置很快就能找到下一个需要运行的任务.因为每一个就绪队列都是由双向循环链表构成,所以当任务运行结束时,调度算法无需遍历链表就可以将任务从队列中删除.易知,调度算法的时间复杂度为 $O(1)$,与系统中的任务数量无关.

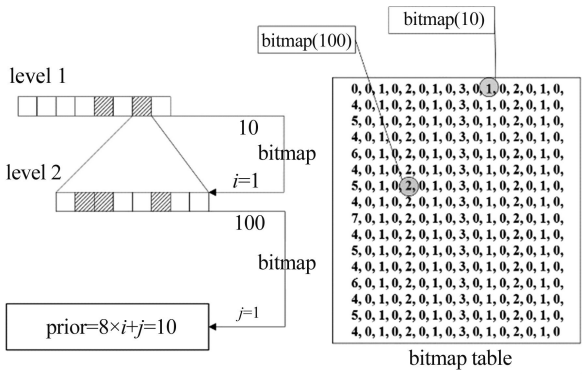

图 4 位图调度算法示例

Fig.4 An example of bitmap scheduling algorithm

ULight 中的任务创建和切换都是依赖于 GNU C 函数库中的 System V contexts 相关函数接口,这些函数接口操作的任务上下文包括任务堆栈、通用寄存器、浮点寄存器、信号屏蔽字等.任务堆栈和通用寄存器的保存和恢复是无法避免的,但绝大多数的任务都无需屏蔽信号且信号屏蔽字的读写开销很大,因为 ULight 提供了功能健全的互斥操作及定时服务,完全可以避免任务间的信号发送,所以 ULight 选择将其省略.

3 定时系统

Linux 内核中的经典定时系统通过访问内核变量 jiffies 来查询时间,定时精度为 1～10 ms,且其定时器到期处理函数只在时钟中断完成后的中断下半部中被当作软中断来执行,定时器精度很低.

为了提升 Linux 系统中定时器的精度,研究人

员设计了一种内核态高精度定时系统 hrtimer[14]，其硬件时钟源是本地处理器上的 APIC，其精度和稳定性都远胜于经典定时系统使用的 PIT[4]，其时间精度可达到纳秒级别.由于 Linux 内核可以在中断上下文中直接执行 hrtimer 的定时器到期处理函数，这也进一步提高了定时器精度.目前 hrtimer 可用的用户态定时器接口并不高效，反而会给系统带来巨大开销.首先，这些接口的调用都会触发系统调用；其次，每一个定时器的设置都会对时钟源进行操作，触发不规律的中断.无论是系统调用还是时钟源中断都会引起用户态和内核态的切换.频繁地切换会给 Linux 系统带来巨大的开销，而且大量的不规律中断甚至会造成中断延迟，严重影响定时精度.为了解决这些问题，ULight 框架引入了一个多核 Linux 平台下的基于 hrtimer 的高精度用户态定时系统，即刻度轮定时系统(tick wheel)[15].

如图 5 所示，刻度轮由若干个刻度和一个刻度指针组成，刻度指针会周期性的指向下一个刻度，指针的移动周期 cycle 被设置为 20 μs.在每一个刻度上，都有一个双向循环链表用于管理所有挂在该刻度上的定时器；定时器的每一个所有者都有一个独立的定期器到期队列.刻度轮定时系统支持的定时器中共有两种：单次触发定时器和周期触发定时器.单次触发定时器用于任务的主动休眠；周期触发定时器则用于模拟时间片以支持相同优先级任务的分时调度.系统在创建它们时需要根据其定时长度和当前刻度指针所在位置计算出定时器到期前指针需要转动的圈数(circles)和定时器到期时指针应指向的刻度(target_tick)，并将装有 circles 的定时器描述单元挂到 target_tick 的链表上.

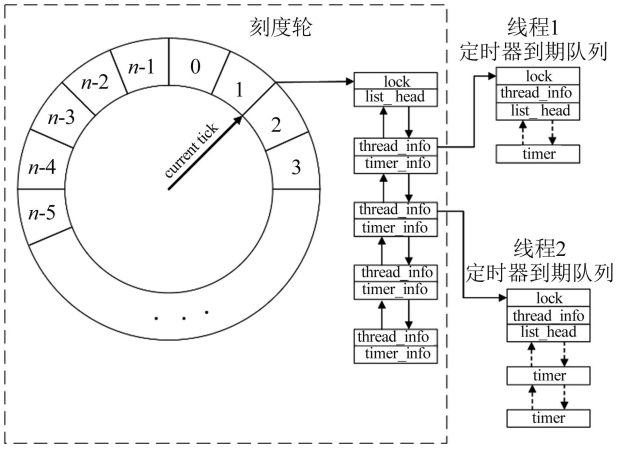

图 5　刻度轮定时系统

Fig.5　Tick wheel timer system

刻度轮定时系统启动时会创建一个内核线程 Timer，负责检查定时器的到期情况并管理所有挂在刻度轮上的定时器.Timer 线程绑定到一个指定的处理器核上，如图 2 中的 timer core，该处理器上的高精度时钟源 APIC 会以 cycle 为周期产生时钟中断，该时钟中断处理函数的核心操作就是将刻度轮上的刻度指针移向下一个刻度以保证刻度轮指针正常转动.在将刻度指针移动后，定时系统会检查新指向的刻度上是否有定时器到期，将到期定时器从刻度轮中摘除并插入其所有者的定时器到期队列中，然后向每一个到期定时器的所有者发送定制信号.通过构造这样的刻度轮定时系统，处理器核上的硬件时钟源就会周期性地产生时钟中断，避免了使用 nanosleep 等 hrtimer 用户态接口致使时钟源产生的不规律硬件中断，减少了系统的运行开销和硬件损耗，进而保证了刻度轮定时系统的定时精度和稳定性.

刻度轮结构是通过共享内存的方式被 Timer 内核线程和基础线程并发访问.在使用刻度轮定时器前，基础线程将刻度轮结构的共享内存区映射到自己的地址空间并在共享内存中创建一条属于自己的独立的定时器到期队列.如果基础线程需要创建定时器，只需从共享内存中分配一个定时器的描述单元并根据定时长度将其挂到正确的刻度上.单次触发定时器到期时，Timer 线程将会从其刻度轮上摘除并插入其所有者的到期队列中，因此所有的操作都是在共享内存中完成，不会引入额外的系统调用，这大大减小了内核态和用户态的切换频率.

4　用户态中断处理

现有的用户态任务库大多是非实时且不可抢占的，一个重要的原因是这些任务系统不能直接处理中断，使得任务系统的实际应用场景较小.ULight 提出了一种用户态中断处理方法，即真实的中断处理函数位于基础线程的用户态地址空间中.这种方法在提升系统完备性的同时，也为任务调度提供了更多合理的抢占点.为方便下文描述，本文定义一些术语：内核态中断处理函数、用户态中断处理函数、用户态中断处理线程.默认情况下，Linux 在内核态对中断进行响应和处理，通过调用 Linux 内核接口，每一个中断向量都可以注册一个对应的内核态中断处理函数.用户态中断处理线程是用户态的普通线程(即 ULight 基础线程)，用户态中断处理函数位

于用户态中断处理线程的地址空间内.

Linux中断处理的默认方式如下:当一个中断发生时,如果系统运行在用户态,它会立即陷入内核态,进入默认的中断处理路径并执行Linux中断处理入口函数do_IRQ();do_IRQ()会根据中断向量找到其注册的内核态中断处理函数,然后在内核态执行该处理函数[5].使能用户态中断处理最直观的方法是在中断发生时,Linux无需陷入内核态而直接跳转到用户态中断处理函数的入口地址;然而这种方法与Linux的设计原理相违背,且与软硬件无法兼容,并不实际.传统的用户态中断处理方法是通过内核态中断处理函数向用户态线程发送信号实现的,也就是说,当系统因某中断而陷入内核态后,内核首先会执行该中断对应的内核态中断处理函数.该函数的核心操作就是向其用户态中断处理线程发送信号,信号处理函数即为真实的中断处理函数.这种方法的缺陷十分明显:首先,在Linux内核中,信号只在系统调用结束或中断处理结束时Linux系统由内核态返回用户态的路径上才会被处理,所以从内核态中断处理函数发出信号到信号处理函数被执行,信号传递时间很长,导致中断处理延迟过大,无法满足实时要求;其次,这种方法不但需要编写用户态线程的信号处理函数,还需编写内核模块以注册内核态中断处理函数,而且内核模块必须事先知道用户态线程的运行时线程信息以便向其发送信号,操作十分不便,实用价值不高.

ULight在内核态和用户态之间构建了一条用户态中断处理快速通道,实现中断在用户态的快速响应.其大致思路是中断发生后,在内核态迅速构造一个快速跳转栈,由内核态直接切换到用户态中断处理函数中执行,处理结束后,再将上下文恢复到中断发生前的状态.为了控制中断处理延迟、保证中断快速响应,ULight框架规定每一个处理器核在同一时刻只能注册一个用户态中断处理线程,也就是多任务调度模块中每一个处理器核上的基础线程.为支持用户态中断处理,ULight需要在Linux内核空间中为每一个处理器核申请一块数组类型的内存,即用户态中断向量表,以存储中断向量及其对应的用户态中断函数的入口地址,数组的长度是系统支持的中断向量的最大值,数组成员被初始化为NULL.同时需要编写相关的系统调用以实现用户态中断处理函数的注册、注销等操作.本文以X86为例介绍用户态中断处理快速通道的搭建.假设某处理器核上的基础线程为user_thread,该线程注册的中断向量VEC的用户态中断处理函数的入口地址为user_handler.

4.1 快速跳转栈

如上文所述,中断发生后,Linux会迅速陷入内核态执行中断处理的入口函数do_IRQ(),该函数的接口定义如下:

```
unsigned int do_IRQ(struct pt_regs *regs);
```

函数do_IRQ()的参数regs所指向的pt_regs结构体中保存了中断发生前的系统上下文,即系统通用寄存器的值,包括AX、BX、IP指令计数器、CS代码段寄存器、FLAGS标志寄存器等.这些寄存器都是在中断时由系统自动保存在内核态堆栈上的.X86体系结构下,中断返回指令IRET会将位于内核态堆栈中栈顶的5个值依次弹入IP指令计数器、CS代码段寄存器、FLAGS标志寄存器、SP栈指针寄存器以及SS堆栈段寄存器中[5],而IP指令计数器指向的是CPU下一条要执行的指令的地址.

若中断VEC发生时,Linux系统正在运行user_thread且运行于用户态,系统会迅速陷入内核态.ULight首先会保存中断前的系统上下文,并在内核态构造一个快速跳转栈fast_stack,用于完成向user_handler的跳转.fast_stack的长度为5,如图6所示,从栈顶开始对fast_stack的设置方法如下:第一个单元设置成用户态的中断处理函数地址user_handler;第二个单元设置设置成__USER_CS;第三个单元设置成中断发生时FLAGS寄存器的值,该值保存在函数do_IRQ()的参数regs指向的pt_regs结构体中;第四个单元设置成用户态中断处理线程的栈顶指针,该值同样保存在函数do_IRQ()的参数regs指向的pt_regs结构体中;第五个单元设置成__USER_DS.其中__USER_CS和__USER_

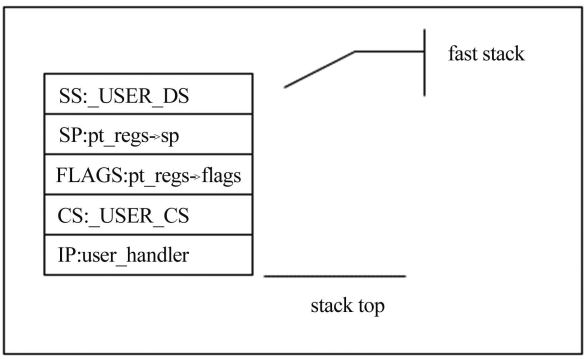

图6 快速跳转栈设置

Fig.6 Setup of fast jump stack

DS 是用于描述代码段和数据段的寄存器，其值在 Linux 中固定不变[5]. 随后 ULight 会将栈指针 SP 设置成 fast_stack 的栈顶地址并执行 IRET 命令，随着 IP、SP 等寄存器的改变，系统就可以跳转到用户态中断处理函数 user_hander 中执行.

4.2　中断推迟处理

当中断 VEC 发生时，如果当前运行线程（即内核态的 current 线程）不是 user_thread 或者 user_thread 正运行在内核态，系统不能完成向用户态立即跳转，用户态中断将被推迟处理.为此，本文引入了两个标志位，NEED_SACRIFICE 和 NEED_UIRQ，来提升被推迟处理的中断进入用户态处理的速度.

如果中断 VEC 发生时，current 不是 user_thread，ULight 会在 do_IRQ()中设置将 current 的 NEED_RESCHED 标志位置位，该标志位为 Linux 原有标志位，用于主动触发进程切换；然后设置 current 的 NEED_SACRIFICE 标志位，Linux 在执行 pick_next_task()函数时会检查该标志位，如果该标志位被置位则会将运行权主动给予该处理器核上的用户态中断处理线程；最后将 user_thread 的 NEED_UIRQ 标志位置位，该标志位强制 user_thread 在由内核态返回用户态的路径上完成对被推迟处理的中断的用户态响应，这些中断的处理时机与信号相同，但优先于信号处理.

如果中断 VEC 发生时，current 是 user_thread，但运行于内核态，内核无法获取 user_thread 的用户态上下文，也同样无法完成直接跳转.此时 ULight 会在 do_IRQ()函数中将 user_thread 进程的 NEED_UIRQ 标志位置位，促使其在由内核态返回用户态的路径上完成对被推迟处理中断的用户态响应.

4.3　中断返回和上下文恢复

中断发生后，系统进入内核态，ULight 会在中断处理入口函数 do_IRQ()中完成对中断发生前系统上下文的保存.用户态中断处理线程在完成对中断的响应处理后，会隐式地调用 switch_back()函数以使系统恢复到中断发生前的状态.调用 switch_back()时，系统会再次进入内核态，根据之前保存的上下文对通用寄存器、浮点寄存器等进行恢复；然后再次构造一个快速跳转栈.不同的是，这次将栈顶的第一个单元填入了之前保存的系统上下文中的 IP 寄存器值，通过将 SP 设置为快速跳转栈的栈顶地址并执行 IRET 指令，系统就会恢复到中断前的位置继续运行.

4.4　用户态中断处理流程

一个中断发生后，系统会陷入内核态，执行中断处理入口函数 do_IRQ()，进入用户态中断处理流程，该流程可以分为以下步骤：

步骤 1　读取存储在内核态栈上的中断向量，以中断向量为索引在用户态中断向量表中查找该中断对应的用户态中断处理函数入口地址，如果地址为 NULL，则继续执行 Linux 内核原生的中断处理流程，跳转至步骤 7；否则，进入步骤 2.

步骤 2　中断发生时，如果当前运行线程是该处理器核上的用户态中断处理线程，并且运行于用户态，则执行步骤 4；否则进入步骤 3.

步骤 3　如果中断发生时，当前运行线程不是该处理器核上用户态中断处理线程，则设置当前运行线程的 NEED_RESCHED 和 NEED_SACRIFICE 标志位，设置用户态中断处理线程的 NEED_UIRQ 标志位，按原生 Linux 内核行为继续运行，跳入步骤 7；如果中断发生时，当前运行线程是该处理器核上的用户态中断处理线程，但其正运行在内核态，则设置线程的 NEED_UIRQ 标志位，按原生 Linux 内核行为继续运行，跳转至步骤 7.

步骤 4　保存系统上下文，构造向用户态中断处理函数跳转的快速跳转栈 fast_stack，将 SP 寄存器设置为 fast_stack 栈顶地址，执行中断返回指令 IRET，完成向用户态中断处理程序的跳转，进入步骤 5.

步骤 5　用户态中断处理函数执行，在函数结尾处隐式执行上下文恢复系统调用 switch_back()，进入步骤 6.

步骤 6　通过使用步骤 4 中保存的系统上下文，完成上下文恢复，回到中断前的系统位置运行，用户态中断处理完成.

步骤 7　Linux 系统继续运行.

在步骤 3 中被推迟处理的中断将会在下一次用户态中断处理线程由内核态返回用户态的路径上被处理，处理时机与信号相同，但优先于信号处理.

4.5　软中断

在中断处理的过程中，系统是需要屏蔽中断的，所以用户态中断处理的过程不宜太长，以防止其他中断被延迟或丢失.ULight 引入了软中断机制以解决这一问题，该机制的设计思想和 Linux 内核中的软中断机制相似.用户态中断处理函数在执行将要结束时可以创建属性为软中断的任务，在中断处理函数执行结束、完成上下文恢复后，系统会触发任务调度.而软中

断的优先级高于普通任务，所以可以抢占当前的运行任务，保证及时被处理，因此如果中断发生后需要完成的工作过多，就可以借助软中断帮助完成，避免中断屏蔽时间过长对系统实时性带来的影响.

5 实验与分析

本节针对评价实时系统性能的几个重要指标对 ULight 进行测试，测试的指标包括任务切换延迟、定时器精度以及中断处理延迟. 实验环境如下：操作系统为 Centos 6.3，内核版本为 Linux 3.4；处理器为 Intel 4 核处理器，主频为 3.4 GHz；内存为 3 GB，内存频率为 1.6 GHz.

5.1 任务切换延迟测试

首先测试任务切换性能.如上文所述，任务切换的时间主要包含两个部分：任务调度和上下文保存与恢复.本节将 ULight 与 Linux pthread 以及两个用户态任务系统 Pth、Libtask 进行比较，对各个任务系统进行乒乓测试，即每个系统都会创建两个相同优先级的任务，两个任务来回切换 10 000 次，统计平均的切换时间，结果如表 1 所示.从表 1 容易看出，Pth 的任务开销最大，在调度中进行了任务的负载平衡、信号检查等一系列复杂操作，调度性能无法保证.Libtask 和 ULight 的任务切换开销要小得多，且相比于 Pthread 有明显的性能优势.Linux 在线程切换时需要完成对页全局目录、内核态堆栈和硬件上下文的切换，再加上用户态和内核态的来回切换，性能开销很大，实时性无法保证；相比于 Libtask，ULight 的任务切换更小，ULight 中使用优先级位图调度算法实现了高效调度，并且对任务上下文进行筛选，避免了不必要的内存操作，合理地减低了切换开销.

表 1 任务切换性能比较

Tab.1 Comparison of task switch speed

任务库	平均任务切换延迟/μs
Linux pthread	0.879
Pth	2.352
Libtask	0.367
ULight	0.132

5.2 定时器精度测试

实验中，将 0 号核设置为 timer core，其他 3 个核中每个核上各启动 10 000 000 个单次触发定时器，且定时时长在 1 ms 至 10 s 的范围内.记定时器设定的定时长度为 T_{set}，记定时器启动时刻到定时器到期处理函数开始执行之间的时间间隔为 T_{real}，统计 T_{real} 和 T_{set} 之间的误差 Err 在不同范围内的数目和比例，结果如表 2 所示.从表 2 容易看出，在启动大规模定时器时，刻度轮定时系统依然保持了良好的稳定性，定时误差超过 20 μs 的情况很少.

表 2 定时精度测试结果

Tab.2 Results of the resolution test of the timer system

	触发总数	Err< 10 μs	10 us<Err <20 μs	Err> 20 μs
1 号核	10 000 000	9 999 971	18	11
2 号核	10 000 000	9 999 981	10	9
3 号核	10 000 000	9 999 963	22	15
总体	30 000 000	29 999 915	50	35
占比	100%	99.9999%	0.0001%	—

5.3 中断处理性能测试

因为现有的用户态多任务系统都没有实现用户态中断处理的功能，所以本节将 ULight 的中断处理与传统用户态中断处理方法（即内核态中断处理加信号传递的方式）进行比较.实验使用的是核间中断，在 X86 体系结构下可调用相关的内核接口向其他处理器核发出核间中断[5]，且用户态中断处理线程被设置为实时线程.因为中断发出的时间不易确定，所以统计从函数 do_IRQ() 开始执行到用户态中断处理函数开始执行之间的时间间隔作为中断响应延迟.

实验中使用 rdtsc 指令统计中断处理延迟，每种方式各处理 10 000 个核间中断，结果如表 3 所示.从表 3 容易看出，两种方式的平均性能差异很大，ULight 的平均中断处理延迟约为 350 个指令周期，实际约为 0.1 μs，证明了方案的可行性.实际上，由于 ULight 搭建中断处理快速通道的开销很小，系统在构建快速跳转栈后可以迅速跳转到用户态的中断处理函数中执行，所以产生的中断处理延迟很小；传统的实现方式在信号传递的过程中存在很多不确定因素，无法保证中断处理的实时性.传统方式中测得最大延迟也远远大于 ULight，ULight 的最大延迟是在中断处理被推迟的情况下产生的，这种情况下 ULight 引入了 NEED_SACRIFICE 和 NEED_UIRQ 两个标志位，加速了对被推迟中断的处理.表 4 对 ULight 中断处理过程中的直接跳转和推迟处理情况进行了统计.由表 4 可以看出，当用户态中断

处理线程被设置为实时线程时，中断处理被推迟的概率很小，约为 0.18%.直接跳转情况下的 ULight 性能非常稳定，最大处理延迟仅为 370 个指令周期.

表 3　用户态中断处理性能比较

Tab.3　Comparison of interrupt handling performance

实现方法	平均延迟(cycle)	最大延迟(cycle)
传统方式	5 221.623	23 655
ULight	353.769	2 537

表 4　ULight 中断处理性能分解

Tab.4　Details of ulight interrupt handling performance

	执行次数	平均延迟(cycle)	最大延迟(cycle)
直接跳转	9 982	350.769	370
延迟处理	18	2 017.500	2 537
总体	10 000	353.769	2 537

6　结论

本文设计和实现了一种多核平台下的 Linux 用户态实时多任务调度框架 ULight.ULight 共包含 3 个模块：多任务调度模块在 Linux 用户态提供基于优先级可抢占的多任务实时调度；定时器模块则为多任务调度提供精度为 20 μs 的定时服务；用户态中断处理模块通过在内核态和用户态之间构造中断处理的快速通道，使线程在用户态直接处理硬件中断.实验表明，ULight 的任务切换效率明显优于 Linux 的线程切换效率；定时系统可以提供稳定的精度为 20 μs 的高精度定时服务；用户态中断处理性能较传统方法有明显性能提升，满足实时系统要求.ULight 实时多任务调度框架为实时系统的设计提供了新思路.

参考文献(References)

[1] 王鹏. Linux 内核实时性的研究与改进[D]. 南京：东南大学，2009.

[2] BEHNAM M，NOLTE T，SHIN I，et al. Towards hierarchical scheduling in VxWorks[C]// Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications. Prague，Czech：IEEE Press，2008：63-72.

[3] KLEINES H，ZWOLL K. Real time UNIX in embedded control-a case study within the context of LynxOS[J]. IEEE Transactions on Nuclear Science，1996，43(1)：13-19.

[4] MAUERER W. Professional Linux Kernel Architecture [M]. Birmingham，UK：Wrox Press，2008.

[5] BOVET D P，CESATI M. Understanding the Linux Kernel [M]. 3ed，Sebastopol，USA：O'REILLY，2005.

[6] OLIVEIRA D B，OLIVEIRA R S. Timing analysis of the PREEMPT RT Linux kernel [J]. Software：Practice and Experience，2015，46(6)：789-819.

[7] KOH J H，CHOI B W. 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)：235-246.

[8] 吴志强，黄章进，顾乃杰，等. 一种多核平台下的用级实时多任务库[J]. 小型微型计算机系统，2015，36(7)：1438-1443.
WU Zhiqiang，HUANG Zhangjin，GU Naijie，et al.A user-level real-time task library on multicore [J]. Journal of Chinese Computer Systems，2015，36(7)：1438-1443.

[9] BARABANOV M. A Linux-based real-time operating system[D]. Socorro，USA：New Mexico Institute of Mining and Technology，1997.

[10] MANTEGAZZA P，DOZIO E L，PAPACHARALAMBOUS S. RTAI：Real time application interface[J]. Linux Journal，2000，2000(72es)：No.10.

[11] ENGELSCHALL R S. Portable multithreading：The signal stack trick for user-space thread creation[C]// Proceedings of the annual conference on USENIX Annual Technical Conference. Berkeley，USA：USENIX Association，2000：20-20.

[12] GRAMOLI V，GUERRAOUI R，TRIGONAKIS V. TM2C：a software transactional memory for many-cores[C]// Proceedings of the 7th ACM European Conference on Computer Systems. Bern，Switzerland：ACM Press，2012：351-364.

[13] AHN H J，CHO M H，JUNG M J，et al. UbiFOS：A small real-time operating system for embedded systems [J]. ETRI Journal，2007，29(3)：259-269.

[14] GLEIXNER T，NIEHAUS D. Hrtimers and beyond：Transforming the Linux time subsystems [C]// Proceedings of the Ottawa Linux Symposium. Ottawa，Canada：ACM Press，2006：333-346.

[15] ZHANG X，GU N J，SU J J. DCUDP：Scalable data transfer for high-speed long-distance networks [J]. Concurrency and Computation：Practice and Experience 2017，29(4)：e3846(1-26).