doi:10.3772/j.issn.1002-0470.2013.02.004

基于混杂模式的无线传感器网络操作系统的研究与设计①

周海鹰②* Hou Kun-Mean** 左德承③* 李 剑* Christophe de Vaulx** 周 鹏*

(*哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)

(**克莱蒙费朗第二大学 LIMOS 国家实验室 克莱蒙费朗 法国 63000)

摘 要 针对无线传感器网络系统资源受限与任务多样性的特点,遵循低资源消耗与环境任务自适应的设计思想,提出了一种基于混杂模式的传感器网络专用操作系统微内核架构,进而设计并实现了适用于资源受限的复杂环境的混合嵌入式实时操作系统(HER-OS)。此微内核架构在体系结构上采用"任务-线程-动作"的模块化设计思想,在运行模式上采用事件驱动结合实时多任务模式,在通信方式上采用基于元组的轻量级 In/Out 元语技术以及资源复用技术。根据环境感知任务的不同,内核可以被选配为事件驱动、实时多任务以及混杂三种不同工作模式。应用实测与对比分析显示,此内核具有低资源需求,满足实时性调度,可支持单任务应用至实时多任务系统。

关键词 无线传感器网络(WSN),操作系统微内核,事件驱动,实时多任务,元组

0 引 言

无线传感器网络(WSN)的节点资源严格受限,应用任务复杂多变,使得其操作系统(WSNOS)必须满足低资源消耗及高适用性的要求。针对 WSNOS 的这种要求,本文研究了其调度机制与通信策略,提出了一种基于混杂模式的操作系统微内核体系结构,设计实现了适用于受限资源与复杂环境的专用系统——混合嵌入式实时操作系统(hybrid embedded real-time operating system,HEROS)。此系统具有资源消耗极低,能适应不同的应用类型的特点,可屏蔽硬件节点、网络协议及应用系统的差异。

1 相关工作

目前,针对 WSNOS 的研究有两种不同思路和技术路线:其一,精简通用实时操作系统(RTOS),使之满足于 WSN 节点的资源限制,代表有 μC/OS-II,VxWorks5.x,QNX,pSOS,LyNxOS,RTLinux,Windows CE.net,SDREAM[1] 等;其二,针对 WSN 特点设计符合 WSN 特色的专用 OS,代表有 TinyOS[2],SOS[3],Mantis OS[4],Contiki[5],EYES OS[6] 等。通用嵌入式 OS 因为受限于节点资源,不能直接应用于 WSN[7];专用 OS 多是直接架构在网络层通信协议之上,严重依赖于节点硬件环境以及下层网络协议。已有的 WSNOS 无法同时支持实时多任务和事件驱动两类典型的 WSN 应用,无法真正实现系统独立于硬件、协议和应用,导致 WSNOS 的通用性受到很大制约,限制了 WSN 应用范围,增加了系统开发成本。

无论是通过精简传统的嵌入式操作系统,还是开发专用的 WSNOS,都已被证明不能很好满足 WSN 低资源节点、以数据为中心以及任务多样性的特点。目前,大多数 WSNOS 研究文献关注于操作系统的体系结构、调度机制以及通信机制等[8,9]:

(1)体系结构:WSNOS 具有三类典型结构:1)组件模型,如 TinyOS:采取组件结构,程序只包含必须部分,减少对内存需求。2)事件驱动模式,如 TinyOS,MantisOS 等:由于 WSN 应用特点是感知事件,对事件做出反应,采用事件驱动模型能有效减少能量消耗。当没有事件发生时,部分系统部件可处于休眠状态。3)有限状态机模型,如 SenOS[10]:系统根据事件类型改变自身状态,从而达到减少能量消

① 国家科技支撑计划(2011BAH04B03)和国际科技合作计划(2010DFA14400)资助项目。

② 男,1975 年生,博士,副教授;研究方向:无线传感器网络,移动计算;E-mail:haiyingzhou@hit.edu.cn

③ 通讯作者,E-mail:zuodc@hit.edu.cn

(收稿日期:2012-04-10)

耗的目的。但是,已有的 WSNOS 多数仅从单节点考虑 OS 设计,缺少对系统统一性和协调性考虑;另一方面,分布式 OS 在任务分解和网络资源消耗的平衡考虑上还处于研究起步阶段,导致系统对分布式、以数据为中心的 WSN 应用特点不能提供很好支持。本文基于模块化思想,提出了基于事件驱动、任务/线程的两级体系结构模型,将 WSN 软件系统分为与硬件无关的系统核心模块(OS 微内核部分)、与硬件相关的驱动资源模块、与通信相关的协议模块,以及应用模块。

(2)调度机制:OS 调度机制与系统体系结构密切相关。当前多数 WSNOS 采用事件驱动模型,导致系统对实时并发任务的处理不理想,无法支持具有实时性要求的任务。WSN 任务具有多样性,既有周期性的数据采集任务,也有实时性要求很高的突发性监测任务。因此,如何在一个 OS 中实现对不同类型任务的支持和调度是一个亟待解决的问题[11]。本文拟实现一个既具有低资源消耗又支持多应用类型的 WSNOS。为了解决周期性任务和突发性任务之间的协调处理问题,实现资源消耗与实时性支持之间的平衡,本文针对任务/线程两级体系结构,设计两级内核调度策略:针对任务的'非抢占式优先级'调度以及针对线程的'抢占式优先级'调度。

(3)通信机制:WSN 以数据为中心的特点,决定 WSNOS 采用基于消息/数据的通信机制,如 TinyOS、SOS、Contiki 等。但上述 OS 都是针对单一节点设计,系统不支持分布式通信。为实现分布式通信,当前较常用的方式是引入中间件技术[12],实现异构 OS 间的相互通信。但通信中间件的引入会增加系统开销,同时也不利于独立出 OS 层。为此,MagnetOS[13]采用网络协议实现模块间通信,但这种方式通信代价大、效率低。目前,WSNOS 对于节点内部(模块软件系统)和节点之间(网络系统)采用的是不同的通信机制和策略。这种相互割离的通信状况不利于统一管理整个 WSN 系统,也不适应于 WSN 系统分布式的应用特点。因此,本文拟实现以数据为中心的分布式通信机制,基于并行程序设计语言-LINDA[14],通过实现轻量 tuple 空间和 In/Out 系统元语来完成系统通信和分布式管理。

2 操作系统体系结构

HEROS 设计遵循以下原则:一是资源相关性原则;二是环境自适应性原则。基于上述需求分析,本研究提出了一款适用于受限资源与复杂环境的 WSN 专用操作系统微内核,主要特点是:体系结构上,采用'任务-线程-动作'的模块化设计方法;运行模式上,采用事件驱动结合实时多任务的混杂模式;通信方式上,采用基于 tuple 的轻量级 In/Out 元语技术。

2.1 系统模型

在分析事件驱动类操作系统与实时多任务类操作系统的基础上,结合 WSN 应用特点,本文提出了基于"任务-线程-动作"($\varepsilon-\delta-\zeta$)的 HEROS 系统架构:系统 Γ 由 N 个相互独立的'任务' ε 所组成;任务 ε 包含 M 个相互关联的'线程' δ;线程 δ 由多个动作 ζ 所组成。

HEROS 采用组件模块化设计,ε 与 δ 均为系统组件,通过动作 ζ 屏蔽感知节点硬件上的差异,并提供统一调用接口。定义符号'‖'为并行操作,'→'为串行操作,则 HEROS 形式化定义如下:

$$\begin{aligned}&\Gamma=\{\varepsilon_i \mid 1\leqslant i\leqslant N,\varepsilon_1\rightarrow\varepsilon_2\rightarrow\cdots\rightarrow\varepsilon_N\}\\&\forall\varepsilon_i,\varepsilon_j\in\Gamma,\ i\neq j \text{ then } \varepsilon_i\cap\varepsilon_j=\varnothing\}\\&\varepsilon=\{\delta_j \mid 1\leqslant j\leqslant M,\ \delta_1\parallel\delta_2\parallel\cdots\parallel\delta_M\}\\&A=\{\xi_t \mid t=1,2,\cdots\}\\&\delta=\{\xi_k \mid 1\leqslant k\leqslant K,\ \xi_k\in A\}\end{aligned}\qquad(1)$$

动作(ζ):HEROS 最小执行单元,是一组特定功能的指令程序封装。HEROS 定义两类基本动作:系统 ζ_{sys}(硬件相关)及功能 ζ_{fun}(硬件无关),构成动作程序库 A。

线程(δ):HEROS 最小组件,执行轻量级任务。构成 ε 的 M 个 δ 采用基于优先级的抢占式调度策略并发执行,可被中断也可被抢占。

任务(ε):封装独立的感知应用。ε 采用基于事件驱动的调度策略,执行'run-to-completion'的运行规则,由事件 ζ 激活,可被中断但不可被抢占。ε 分为两类:周期性任务 ε_p,为定时性的感知信息获取或激励控制类任务,具有短暂的执行时间和确定的响应时间;突发性任务 ε_s,为时间约束类任务,具有可预测的响应时间以符合实时要求。

HEROS 将系统组件定义为一个具有五元组属性集 $<\alpha,\vartheta,\tau,\rho,\kappa>$ 的实体对象:

α:组件对象实体(如任务 ε,线程 δ),描述了组件的功能与行为,为代码实现与程序封装;

ϑ:组件对象状态属性,定义五种状态 $<$ *Sleep*, *Ready*, *Execute*, *Suspend*, *Terminate* $>$,其中'*Suspend*'为线程专属;

τ：组件对象时间属性，表示为一个四元组的时间变量集 $< e, p, r, x >$，分别描述组件对象在一个运行周期内的完全执行时间 e、周期大小 p、组件对象的初始释放时刻 r，以及组件对象的实际执行时间 x；

ρ：组件对象优先级，采用基于静态设置与动态调整的‘任务—线程’的优先级策略：组件对象在初始化时设置其静态优先级，在运行调度时可动态调整优先级；

κ：组件对象空间属性，标识组件对象私有内存空间，采用基于元组（tuple）的动态内存分配管理策略，系统组件间的通信均基于元组空间。

基于上述架构，根据应用与环境自适应的设计原则为 HEROS 选配不同运行模式。考虑下述两种特殊情况：

（1）假定一：若实例 Γ 仅含有一个任务 ε，即

if $(|\Gamma| == 1)$, then $\Gamma = \{\varepsilon_1\}$

$= \{\delta_{1,j} \mid 1 \leqslant j \leqslant M, \delta_{1,1} \parallel \delta_{1,2} \parallel \cdots \parallel \delta_{1,M}\}$

$\Rightarrow \Gamma$

是一个典型的实时多任务类系统。

（2）假定二：若组成实例系统 Γ 的任意任务 ε 均仅含有一个线程实体 δ，即

if $(\forall M_i == 1)$, then $\Gamma = \{\varepsilon_i \mid 1 \leqslant i \leqslant N\}$

$= \{\delta_{i,1} \mid 1 \leqslant i \leqslant N; \delta_{1,1} \rightarrow \delta_{2,1} \rightarrow \cdots \rightarrow \delta_{N,1}\}$

$\Rightarrow \Gamma$

是一个典型的事件驱动类系统。

2.2 组件时间属性与系统调度机制

2.2.1 组件时间特性

通过分析环境感知应用的时间特性，为简化处理，本文给出如下假设：

（1）组成 HEROS 系统的任务 ε 都是相互完全独立的周期性任务 ε_p，突发性任务 ε_s 被视为具有唯一实例（第一个）的周期性任务，即 $\varepsilon_s = \varepsilon_{p,1}$；

（2）组件的相对截止时限 D_i 与组件执行周期 p_i 一致，即满足 $D_i = p_i$。

根据组件的时间约束特性，在任务 ε_i 第一次释放时刻 φ_i，需静态地配置 ε_i 及构成该任务的线程组 $\varepsilon_i = \{\delta_{ij}, 1 \leqslant j \leqslant M\}$ 的时间属性集 τ。对于线程 $\delta_{i,j}$，由系统用户根据线程的时间约束进行设定，线程周期 $p_{\delta(i,j)}$ 为常数，为线程的相对截止时限 $D_{\delta(i,j)}$；线程的初始释放时刻 $r_{\delta(i,j)}$，即为当前时刻 $\varphi_{\delta(i,j)}$；线程的完全执行时间 $e_{\delta(i,j)}$ 由系统用户根据线程的应用特点与其动作构成进行设定；线程的实际执行时间 $x_{\delta(i,j)}$ 初始化为 0。对于任务 ε_i，任务周期 $p_{\varepsilon(i)}$ 为常数，即为任务的相对截止时限 $D_{\varepsilon(i)}$，由系统用户根据任务时间约束进行设定；任务初始释放时刻 $r_{\varepsilon(i)}$ 即为当前时刻 φ_i；任务的完全执行时间 $e_{\varepsilon(i)}$ 及实际执行时间 $x_{\varepsilon(i)}$ 均为统计变量，分别表示为各线程的完全执行时间之和以及实际执行时间之和，式为

$$\begin{aligned} e_{\varepsilon(i)} &= \sum_{j=1\cdots M} e_{\delta(i,j)} \\ x_{\varepsilon(i)} &= \sum_{j=1\cdots M} x_{\delta(i,j)} \end{aligned} \tag{2}$$

任务周期性实例的时间属性描述如下：定义 $r_{\varepsilon(i,k)}$ 与 $d_{\varepsilon(i,k)}$ 为任务 ε_i 的第 k 个实例的释放时间与绝对截止时间，任务 ε_i 第一个实例释放时间 φ_i 为 0，因此，有

$$\begin{aligned} \varphi_i &= r_{1,k} \\ r_{i,k} &= \varphi_i + (k-1) \cdot p_i \\ D_i &= p_i \\ d_{i,k} &= r_{i,k} + D_i = \varphi_i + k \cdot p_i \end{aligned} \tag{3}$$

2.2.2 组件优先级设置

根据组件对象的时间属性集 τ，HEROS 通过静态设置与动态调整设置组件对象的优先级。

在组件释放初始化阶段，任务优先级 ρ_ε 根据任务的绝对截止时间 d_i，基于“最早截止时间优先（earliest-deadline- first，EDF）”的原则进行静态设定，即“距离绝对截止期限最近的任务对象具有最高优先级”，如表 1（左）所示。在组件释放初始化阶段，线程优先级 ρ_δ 由用户根据同源线程间关联关系进行设定。由于线程间存在相关性，为避免线程之间因互锁出现‘死锁’，确保线程具有可预测的调度，ρ_δ 设定遵循以下原则：

表 1 HEROS 组件优先级设置策略

静态任务优先级分配策略	动态线程优先级修正策略
$C = \{d_i \mid 1 \leqslant i \leqslant N\}$;	$C = \{\Delta t_k \mid 1 \leqslant k \leqslant M\}$;
For $i \leftarrow N$, 1 do	For $j \leftarrow 1$, M do
//设置 N 个就绪（*Ready*）任务的优先级 ρ_ε	//调整任务 ε 的 M 个就绪线程的优先级 ρ_δ
$d_j = MIN(C)$;	$\Delta t_m = MAX(C)$;
$C = C - \{d_j\}$;	$C = C - \{\Delta t_m\}$;
$\rho_{\varepsilon(j)} = i$;	$\rho_{\delta(m)} = \rho_\varepsilon + j$;

（1）线程 δ 仅接收来自同源线程内部事件 ζ_{in} 触发，禁止来自于 I/O 的外部事件 ζ_{out}，即

$$\exists \delta_i, \delta_j \in \varepsilon, (\delta_i) \xleftarrow{\vee \zeta_{in}} (\delta_j), (\delta_i) \xleftarrow{\times \zeta_{out}} (IO)$$

（2）若线程 δ_j 被同源线程 δ_i 触发，则认为线程

δ_j 具有仅次于线程 δ_i 的优先级,依此类推。即

$$\exists \delta_i,\delta_j \in \varepsilon,\ \text{if } sus(\delta_i) \xleftarrow{\zeta} exe(\delta_j)$$
$$\text{then } \rho_{\delta(i)} = \rho_{\delta(j)} - 1$$

(3) 若两个及以上线程出现循环触发时,则任务初始释放时将上述线程合并为新的同源线程 δ_k,即

$$\exists \delta_i,\delta_j \in \varepsilon,\ \text{if } sus(\delta_i) \xleftarrow{\zeta} sus(\delta_j)$$
$$\text{then } \delta_i \cup \delta_j \Rightarrow \delta_k,\delta_k \in \varepsilon$$

在组件调度运行期间,针对 ε,执行‘run-to-completion’的运行机制,任务不具备动态优先级;针对 δ,根据线程相对截止期限 $(r + p_\delta - t)$(t 为当前时刻)与尚未完成的执行时间 $(e_\delta - x_\delta)$ 之差来调整组件优先级,即

$$\Delta t = |(r + p_\delta - t) - (e_\delta - x_\delta)|。$$

遵循“最晚截止时间最后(latest-deadline-last,LDL)”的原则设置组件动态优先级:即离截止期限最远的组件对象具有最低优先级,如表1(右)所示。

2.2.3 基于优先级的系统调度算法

为减少系统资源消耗并满足任务实时性需求,针对 HEROS 的‘$\varepsilon-\delta-\zeta$’架构,提出基于‘$\varepsilon-\delta$’的两级优先级调度策略:针对 ε,采用基于‘非抢占式的优先级’调度机制;针对 δ,采用基于‘优先级的抢占式’调度机制。

HEROS 支持4种条件激活调度:(1) 定时器中断服务程序(ISR)周期性地进行调度检测;(2) 当前线程 δ_{cur} 被阻止:δ_{cur} 被高优先级线程 $\delta_{highest}$ 抢占或 δ_{cur} 执行条件未满足;(3) δ_{cur} 被终止:δ_{cur} 执行结束或执行超时;(4) 任务被周期性外部事件 ζ_{out} 或被 I/O 中断产生的突发外部事件 ζ_{out} 所触发激活。

HEROS 调度策略如表2所示:针对当前任务

表2 HEROS 系统调度策略

系统调度算法
if $\vartheta_\delta(cur)$ == TERMINATE) //终止当前线程
Kill (δ_{cur}); $\delta_{cur} = \delta_{nxt}$;
else if ($\vartheta_\delta(cur)$ == SUSPEND) //阻止当前线程
Restore_context (δ_{cur}); $\delta_{cur} = \delta_{nxt}$;
else //无调度操作
Return;
end if
if (δ_{cur} == NULL) //当前任务的所有线程执行完毕
Kill (ε_{cur}); $\varepsilon_{cur} = \varepsilon_{nxt}$; $\delta_{cur} \leftarrow \delta_{highest}$ of ε_{cur};
end if //执行新的线程
Load_context (δ_{cur}); $\vartheta_\delta(cur)$ = RUNNING;

ε_{cur},采用基于‘抢占式的优先级’机制调度 ε_{cur} 线程集合;若当前任务的所有线程执行完毕或被终止,采用基于‘非抢占式的优先级’机制调度任务集。

2.2.4 调度实时性分析

HEROS 将突发性任务 ε_s 视为周期 p_s 为相对截止期限 D_s 的周期性任务 ε_p。因此,HEROS 实时性问题的关键在于分析 ε_p 的时间约束。按照 EDF 法则,假定系统由 N 个周期性任务 ε_p 组成,则 HEROS 实时调度的充要条件是

$$\sum_{i=1}^{N}(e_i/p_i) \leqslant 1,\quad \forall \varepsilon_i \in \Gamma, D_{\varepsilon(i)} = p_{\varepsilon(i)} \tag{4}$$

假定在 ε_{cur} 执行期间,因外部事件 ζ_{out} 激活某突发性任务 ε_s,按照 HEROS‘任务不可抢占’的特性,ε_s 将在 ε_{cur} 执行结束之后获得 CPU 资源。因此,ε_s 满足实时调度的必要条件是:ε_s 绝对截止时限 d_s 小于 ε_{cur} 绝对截止时限 d_{cur} 与 ε_s 执行时间 e_s 之和,即

$$d_s < d_{cur} + e_s \tag{5}$$

考虑到实际应用中 ε_s 是由突发事件所触发,通常需要被紧急处理。若式(5)不成立,为使 ε_s 满足实时调度,提出‘$\varepsilon-\delta$ 跃迁’机制,打破任务与线程的严格界定,允许紧急任务 ε_s 抢占当前任务 ε_{cur}。描述如下:

定义 ε_s 与 ε_p 的线程优先级集合 C_s 与 C_p:

$$C_s = \{\rho(\delta_\varepsilon(i) \mid 1 \leqslant i \leqslant M_s, \delta_\varepsilon(i) \in \varepsilon_s\}$$
$$C_p = \{\rho(\delta_\varepsilon(j) \mid 1 \leqslant j \leqslant M_p, \delta_\varepsilon(j) \in \varepsilon_{cur}\}$$

设定 $\forall \delta_\varepsilon(i) \in \varepsilon_s$,令 $\delta_\varepsilon(i) \in \varepsilon_{cur}$ 且满足 $\rho(\delta_\varepsilon(i) > \text{MAX}(C_p)$。

按照设定,原属于 ε_s 的线程并入 ε_{cur},且相较于 ε_{cur} 的线程具有更高优先级。按照任务调度机制,满足调度条件(2),原属于 ε_s 的线程因此抢占执行线程 δ_{cur},立即获得 CPU 资源。此时,由于 ε_{cur} 加入了原属于 ε_s 的线程组,任务执行时间 e_{cur} 增加。同时,为保持任务实时性,任务周期 p_{cur} 需保持不变,此时,实时调度充要条件是:

$$(e_1 + e_s)/p_1 + \sum_{i=2}^{N}(e_i/p_i) \leqslant 1 \tag{6}$$

2.2.5 调度示例

为说明上述调度机制,定义如下 HEROS 实例:在初始释放时刻 $t = 0$,释放 $\Gamma := \{\varepsilon_1, \varepsilon_2; \varepsilon_1 \rightarrow \varepsilon_2\} = \{<\delta_{11}, \delta_{12}>, <\delta_{21}, \delta_{22}>\}$,任务 ε_i 的释放时刻 r_i、执行时间 e_i 以及任务周期 p_i 如表3定义。

表 3 HEROS 实例调度示例

ε_i	r_i	e_i	p_i	δ
ε_1	0	4	10	δ_{11}, δ_{12}
ε_2	0	2	10	δ_{21}, δ_{22}
ε_a	1	1	2	δ_a
ε_b	5	2	20	δ_b

此时,式(4)得到满足,即 $e_1/p_1 + e_2/p_2 = 0.6 < 1$。因此,HEROS 实例具备实时调度可行性。按优先级分配算法,得到 $\rho_{\varepsilon(1)} > \rho_{\varepsilon(2)}$,执行‘$\varepsilon_1 \rightarrow \varepsilon_2$’次序。HEROS 在一个系统周期内的调度状况如图 1 所示。

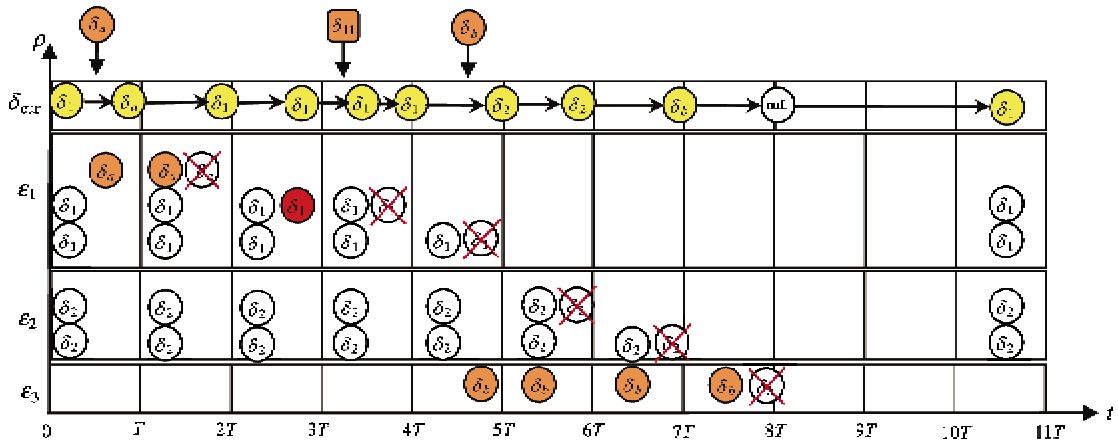

图 1 HEROS 调度策略示意图

突发性任务 ε_a 在 $r_a = 1$ 被释放,$e_a = 1$,需要在 2 个时间隙内($D_a = p_a = 2$)执行完毕。按照基于事件驱动的任务调度机制,执行‘$\varepsilon_1 \rightarrow \varepsilon_2 \rightarrow \varepsilon_a$’调度顺序。因此,$\varepsilon_a$ 至少需等待 $e_1 + e_2 = 6$ 个时间隙,显然无法满足相对截止时间($D_a = 2$)的时间约束。为此,采用‘$\varepsilon - \delta$ 跃迁’机制:将 ε_a 线程 δ_a 分配给当前任务 ε_1,且为 δ_a 分配最高优先级。此时,HEROS 满足式(6),即 $(e_1 + e_a)/p_1 + e_2/p_2 = 0.7 < 1$。因此,$\delta_a$ 将抢占 δ_{11} 优先获得 CPU 资源。

突发性任务 ε_b 在 $r_b = 5$ 被释放,$e_b = 2$,需在 20 个时间隙内($D_b = p_b = 20$)处理完毕。此时,HEROS 由三个任务组成,即 $\Gamma := \{\varepsilon_1, \varepsilon_2, \varepsilon_b;\ \varepsilon_1 \rightarrow \varepsilon_2 \rightarrow \varepsilon_b\} = \{<\delta_{11}, \delta_{12}>, <\delta_{21}, \delta_{22}>, <\delta_b>\}$,满足系统实时调度的充要条件,即 $e_1/p_1 + e_2/p_2 + e_b/p_b = 0.7 < 1$。

2.3 组件空间属性与系统通信策略

HEROS 遵循低资源消耗的设计原则,基于统一地址空间,支持系统内核和任务组件之间的内存共享,降低内存开销并减少数据拷贝。HEROS 通信策略基于并行程序语言 Linda 的“元组空间”(tuple space)思想[14],通过提供基于 tuple 的系统操作和 In/Out 元语,支持组件间通信。

传统嵌入式操作系统是通过共享内存或是信息传递方式达到系统通信与同步目的,在存储空间上或是时间上相关。在标准的 tuple 空间中,消息提取是依靠 tuple 内容匹配,进程之间空间上不相关;同时,由于进程间的异步通信,进程之间时间上不相关。基于标准 Linda 在 tuple 空间上的时间与空间不相关性,HEROS 采用基于元组空间的通信模式,支持进程间异步通信,具有可扩展性,能支持多进程并行设计模式,也适合于分布式多应用处理。

但是,标准的 Linda 概念不适合分布式硬实时并行处理,当进程或是处理器达到一定数目上限时,进程间通信的时间开销将不再确定,会随着进程和处理器数目的变化而迅速改变[15]。为了克服上述问题,HEROS 简化了 tuple 空间并实现两个轻量级系统元语(Out 和 In),以取代传统的基于消息的内存匹配和 tuple 提取方法,见表 4。

表 4 HEROS 系统元语:In/Out

$Out(\kappa, msg)$	$In(\kappa, msg)$

tuple (κ) $\leftarrow msg$;
$\exists \delta_i$, tuple (κ) $\in \delta_i$;
$\vartheta_\delta(i)$ = READY;
if ($\rho_{\delta(i)} > \rho_{\delta(cur)}$)
$\vartheta_\delta(cur)$ = SUSPEND;
Call schedule();
end if
	tuple (κ) $\leftarrow msg$;
	if (tuple (κ) = = NULL)
	$\vartheta_\delta(cur)$ = SUSPEND;
	Call schedule();
	end if

HEROS 采用基于元组标识符 κ 进行 tuple 匹配。元组标识符 κ 及其类型被指定给一个本地的、

共享的或是分布式的存储空间,此过程为静态配置。存储空间被映射至一个 tuple 表的地址队列上,构成元组空间。tuple 的匹配开销非常小且可以确定,并不随进程数目和处理器数目的变化而改变。

HEROS 组件共享一个由无序元组组成的元组空间。元组空间是系统组件(任务 ε、线程 δ、I/O 设备)间的数据交换区域,标识符 κ 提供内存接入保护。通过获得 κ,组件能够在元组 tuple(κ)内存单元上执行读或者写操作。元组内存单元是一个循环存储块,支持并行执行对 tuple(κ)的单次读和多次写操作。为避免内存溢出,组件读操作可一次读取多条消息(*msg*)。

HEROS 是一个基于 tuple 的事件驱动系统,事件消息(*msg*)必须与唯一 tuple(κ)关联。当 *msg* 自 I/O 设备或其它组件到达相关联的 tuple 时,相应事件(ζ)被触发,系统将激活组件调度并唤醒在 tuple(κ)上被暂停的任务 ε。图 2 显示了 HEROS 组件通信与同步工作模式。

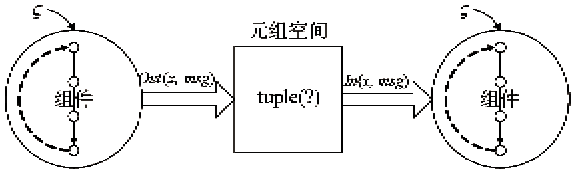

图 2 HEROS 组件通信与同步模式

3 性能评估

目前,HEROS 系统已经移植到远程心电实时看护系统(STAR)的智能无线心电图传感器节点[16]。系统主要耗能部件为 ARM7 架构的 AT91SAM7S256 微处理器,Chipcon CC2430Zigbee 模块,以及工作电压为 3.6V 的锂电池(1800mAH)。分析平台硬件与应用需求,HEROS 采用实时多任务运行模式,远程心电数据监控任务 ε_{ECG} 由数据采集线程 δ_S、预处理线程 δ_P 以及通信线程 δ_T 所构成,即 $\Gamma: = \{\varepsilon_{ECG}\} = \{<\delta_S, \delta_P, \delta_T>\}$。

本文评测 HEROS 系统内存和能量消耗,计算了系统元语与系统反应时间,并与 TinyOS 系统进行性能对比。

3.1 资源消耗分析

HEROS 是针对资源受限的感知节点,面向复杂的无线感知应用,具有严格的资源消耗约束,主要体现在内存、CPU 以及能量资源上。表 5 反映 HEROS 实例的内存资源与能量消耗情况。

与其它实时系统相比,HEROS 实现具有相对较小的内存需求(<5kB),能够被配置工作在不同模式(事件驱动、实时多任务以及混杂)。为减少能耗,感知节点的主要耗能模块(微处理器、无线收发模块)都支持低耗工作模式。当 HEROS 系统处于空闲状态(未有组件运行或是未有数据传输)时,HEROS 可以自配置节点的主要耗能模块处于低耗状态,以延长节点的生存时间。对于大多数感知应用,比如环境数据采集类与安全监测类感知应用,传感器数据采样频率很低(<1MHz),HEROS 在多数时间处于空闲状态,HEROS 系统能够被设置运行在低耗模式。表 5 显示,HEROS 实例节点在正常状态下工作超过 16h,在低耗模式下运行超过 1 年。

表 5 HEROS 的内存消耗与能耗评估

内存消耗(byte)		能量消耗(锂电池,3.6V,1800mAh)			
代码	数据			正常模式	低电模式
3572	1272	AT91SAM7S256(48MHz)		<50mA	<60μA
		Chipcon CC2430(ZigBee)	Send	<25mA	
			Receive	<27mA	
			Sleep	<0.9μA	
			Standby	<0.6μA	
		生存时间		>16 h	>1 year

3.2 系统实时性分析

3.2.1 系统元语分析

HEROS 系统元语执行组件通信与同步操作。一方面,对于静态配置的 HEROS 系统,其系统元语(In/Out)的执行操作是确定的;另一方面,HEROS 系统元语所传递的消息大小是确定且预置的。因此,In/Out 元语的执行时间是确定与可预测的。根据执行条件的不同,系统元语的执行时间(e_{in} 与 e_{out})在最大值与最小值的区间内,即 $e \in [e_{min}, e_{max}]$。表6给出了HEROS系统元语执行时间的评

表 6 HEROS 系统元语的性能评估

系统元语		指令周期(cycle)	执行时间(μs)(48MHz)	执行条件
In	Max ($n=1$)	149+46·n=195	3.101+0.957·n=4.058	tuple(x)→*msg*;
	Min	95	1.977	∵ tuple(x) is NULL ∴ Call *schedule*();
Out	Max ($n=1$)	193+32·n=225	4.016+0.666·n=4.682	tuple(x)←*msg*; ∵ $\rho_{\delta(ready)} > \rho_{\delta(cur)}$; ∴ Call *schedule*();
	Min ($n=1$)	104+32·n=136	2.164+0.666·n=2.83	tuple(x)←*msg*;

估,其中 n 表示传递消息的字节数。当 N = 1,系统元语的执行时间(μs)为:$e_{in} \in [1.98, 4.06]$,$e_{out} \in [2.83, 4.68]$。由于元语操作是操作系统最频繁的行为,短暂而确定的元语执行时间有助于确保系统行为的实时性。

3.2.2 系统时延分析

对于实时系统,系统调度策略以及中断处理机制必须高效且时间可预测。基于‘$\varepsilon-\delta-\zeta$’架构,以及组件可中断、任务可抢占的系统特性,HEROS 采用 4 类时延指标评估系统的调度与中断处理机制:

(1)‘$\varepsilon-\varepsilon$’时延:由 Schedule 调度产生,包含任务内最高优先级线程 $\delta_{highest}$ 的‘冷启动’动作,调用 Cold _ startup,没有上下文切换操作。

(2)‘$\delta-\delta$’时延:由 Schedule 调度产生,包含有保存当前任务上下文现场动作 Restore _ context,以及启动下一个执行线程(若为首次启动,执行‘冷启动’Cold _ startup;若为重新激活,执行‘热启动’Warm _ startup,并恢复上下文现场 Load _ context)。

(3)‘δ – ISR’时延:由系统从被中断线程的最后一条指令执行到中断服务程序(ISR)的第一条指令间的时间间隔来度量,反映系统对外部中断事件的响应速度;系统操作需保存部分通用寄存器。若中断发生时系统处于中断禁止的临界区,将导致最大时延产生。

(4)‘ISR – δ’时延:由系统从 ISR 最后一条指令到下一个被调度进程的第一条指令的时间间隔来度量,反映了从中断回到线程的时间开销;系统操作需恢复部分通用寄存器。若中断发生引发系统调度,将引发‘$\delta-\delta$’调度。

表 7 显示了 HEROS 系统时延的评测:‘$\varepsilon-\varepsilon$’调度时延为确定值(μs),即 $e_{\varepsilon-\varepsilon}$ = 1.87;‘$\delta-\delta$’调度时延因线程冷或热启动,满足 $e_{\delta-\delta} \in [3.27, 3.48]$;‘$\delta$ – ISR’中断响应时延因考虑到中断禁止的临界区问题,满足 $e_{\delta-ISR} \in [0.60, 5.14]$;‘ISR – δ’异常返回时延因考虑到中断引发的系统调度问题,满足 $e_{ISR-\delta} \in [0.67, 3.67]$。表 7 显示 HEROS 系统具有确定的调度时延和中断时延,符合实时性需求,且系统时延不受系统组件数目的影响。

表 7 HEROS 系统时延评估

系统时延		指令周期(cycle)	执行时间(μs)(48MHz)	系统操作
$\varepsilon-\varepsilon$	Avg	90	1.873	Cold _ startup
$\delta-\delta$	Min	157	3.269	Restore _ context(δ); Cold _ startup
	Max	167	3.477	Restore _ context(δ); Warm _ startup(Load _ context)
$\delta-ISR$	Min	29	0.604	Restore _ context(δ)_ Part;
	Max	247	5.140	Restore _ context(δ)_ Part; Max(‖ … ‖)
$ISR-\delta$	Min	32	0.666	Load _ context(δ)_ Part
	Max	176	3.662	Load _ context(δ)_ Part; Restore _ context(δ); Warm _ startup(Load _ context)

3.3 性能比较

表 8 从资源消耗以及响应时延方面对 HEROS 与 TinyOS 进行了性能对比分析。TinyOS 在 ATmega128 微处理器上(4MHz)测试,HEROS 在 AT91SAM7S256 微处理器上(48MHz)测试。由于两个系统在不同平台上进行测试,本文仅比较两个系统的内存消耗以及三种主要的系统行为:任务调度、上下文切换以及硬件中断(包括硬件部分与软件部分)。

结果显示,针对主要的系统行为,HEROS 与 TinyOS 具有大致相同的计算资源消耗。此外,HEROS 以更多的内存资源消耗为代价,支持了事件驱动与实时多任务两种运行模式,提升了系统的扩展性和兼容性,适用于不同的硬件平台和应用请求。

表 8 HEROS 与 TinyOS 性能对比

主要操作	HEROS (AT91SAM7S256)		TinyOS (ATmega128)	
	指令周期(cycle)	执行时间(μs)	指令周期(cycle)	执行时间(μs)
任务调度	43	0.895	46	11.5
上下文切换	56	1.165	51	12.75
硬件中断(hw)	5	0.104	9	2.25
硬件中断(sw)	61	1.269	71	17.75
内存消耗(byte) 代码	3572		432	
内存消耗(byte) 数据	1272		46	

4 结 论

HEROS 是一个针对资源受限的感知节点,面向复杂的无线感知应用,基于事件驱动与实时多任务的操作系统微内核。HEROS 遵循资源相关性以及环境自适应性的设计原则,为减少系统资源消耗并满足任务实时性需求,采用‘任务-线程-动作’的系统架构,并据此采用一种基于‘任务-线程’的两级优先级调度策略:针对任务,采用基于‘非抢占式的优先级’调度机制;针对线程,采用基于‘优先级的抢

占式'调度机制;提供'任务—线程跃迁'机制以确保实时性。HEROS 支持一个以元组为基础的系统通信机制:简化的元组空间以及轻量级组件通信元语(Out 和 In),实现系统组件与 I/O 设备之间的通信与同步操作。根据感知任务的不同,HEROS 可以被选配为事件驱动、实时多任务以及混杂三种不同工作模式。HEROS 能运行的平台从超低资源的设备到复杂的分布式系统,能支持从简单单任务应用到实时多任务应用。

HEROS 未来具有两种不同的发展趋势:面向通用嵌入式平台系统或针对 WSN 节点系统。一方面,为了实现一个实时通用的操作系统,HEROS 应该是具有容忍软、硬件错误(物理破坏除外)和连续不断正常工作能力的容错系统,具有在多个处理器上并行处理能力的分布式系统。另一方面,与 TinyOS 相比,HEROS 占用更多的系统资源(CPU 和内存),因此,必须提高 HEROS 系统性能、降低资源需求以适应无线感知设备。通过使用 meta 语言描述系统原子级别动作,从而使 HEROS 系统行为由汇编语言编写的动作集合所组成,系统元语、函数以及应用可以由多个动作组合而成。因此可以根据不同的应用需求配置和优化 HEROS,降低资源需求,提升系统性能。

参考文献

[1] Zhou H Y, Hou K M, de Vaulx C. SDREAM: A super-small distributed rEAl-time microkernel dedicated to wireless sensors. *International Journal of Pervasive Computing and Communications*, 2006, 2(4):398-410

[2] Levis P, Madden S, Polastre J, et al. TinyOS: An operating system for sensor networks. *Ambient Intelligence (Part II)*, 2005, 35: 115-148

[3] Han C C, Kumar R, Shea R, et al. A dynamic operating system for sensor nodes. In: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, New York, USA, 2005. 163-176

[4] Abrach H, Bhatti S, Carlson J, et al. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks & Applications (MONET)*, 2005,10(4): 50-59

[5] Dunkels A, Grövall B, Voigt T. Contiki: a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, Tampa, USA, 2004.455-462

[6] Dulman S, Havinga P. Operating System Fundamentals for the EYES Distributed Sensor Network. Progress Report, Utrecht, Netherlands, 2002. 1-4

[7] Blumenthal J, Handy M, Golatowski F, et al. Wireless sensor networks: new challenges in software engineering. *Journal of Software*, 2007,18(5):1218-1231

[8] Raatikainen K E E. Operating system issues in wireless Ad-hoc networks. In: Proceedings of the Keynote Speech in International Workshop on Wireless Ad-Hoc Networks, London, UK, 2005. 170-174

[9] 张朋,陈明,陈亚萍等. WSN 操作系统关键技术研究. 计算机应用研究,2007, 24(10):23-25

[10] Hong S, Kim T H. SenOS: State-driven operating system architecture for dynamic sensor node reconfigurability. In: Proceedings of International Conference on Ubiquitous Computing, Seoul, Korea, 2003. 201-203

[11] Duffy C, Roedig U, Herbert J, et al. An experimental comparison of event driven and multi-threaded sensor node operating systems. In: Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications Workshops, Washington, DC, USA, 2007. 267-271

[12] Li W F, Wang R H, Sun L J. Study on communication model between heterogeneous operating systems for a wireless sensor network middleware. *Chinese Journal of Electronics*, 2007,16(3): 543-546

[13] Barr R, Bicket J C, Dantas D S, et al. On the need for system-level support for ad hoc and Sensor Networks. *ACM Operating System Review*, 2002, 36(2):1-5

[14] Gelernter A D. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 1985,7(1): 80-112

[15] Park Y H, Hou K M. Embedded performance analysis tool for a distributed hard real-time microkernel: H-LINDA. In: Proceedings of the 3rd International Conference on Principles of Distributed Systems, Hanoi, Vietnam, 1999.67-84

[16] Zhou H Y, Hou K M, de Vaulx C, et al. Real-time Cardiac Arrhythmias Monitoring for Pervasive Healthcare. Wireless Body Area Networks: Technology, Implementation and Applications. Singapore: Pan Stanford Press, 2011.41-73

Study and design of a promiscuous mode-based operating system for wireless sensor networks

Zhou Haiying*, Hou Kun-mean**, Zuo Decheng*, Li Jian*, Christophe de Vaulx**, Zhou Peng*
(*School of Computer Science & Technology, Harbin Institute of Technology, Harbin 150001)
(**Laboratory LIMOS CNRS 6153, University of Blaise Pascal, Clermont-Ferrand, France 63000)

Abstract

Based on the analysis of the characteristics of limited resources and task diversity of wireless sensor networks (WSN), a dedicated operating system micro-kernel based on a promiscuous mode for wireless sensor networks was presensed according to the design principles of low resource consumption and task self-adaptation, and based on the kernel, a hybrid embedded real-time operating system (HEROS) suitable for source limited complex networks was designed and implemented. The operating system micro-kernel has the features below. In system architecture, adopting 'task-thread-action' modular design fashion; in operation mode, combining the event-driven and real-time multitasking; in communication mechanism, adopting the lightweight tuple-based In/Out primitives and resources reuse technologies. According to the environmental surveillance tasks, this micro-kernel can be self-configured to run in the modes of event-driven, real-time multi-tasking and mixture. The experiential results show that this kernel meets the requirements of low-resource consumption and real-time scheduling, and supports a diversity of WSN applications from simple single task to complex real-time multi-tasking systems.

Key words: wireless sensor networks (WSN), operating system micro-kernel, event-driven, real-time multi-tasking, tuple