Journal of Computer Applications
计算机应用,2014,34(5):1408 - 1412
ISSN 1001-9081
CODEN JYIIDU
2014-05-10
http://www.joca.cn

文章编号:1001 - 9081(2014)05-1408-05
doi:10.11772/j.issn.1001-9081.2014.05.1408

面向嵌入式的协程与脚本化机制

邹昌伟[1*],王 林[2]
(1.福建师范大学 软件学院,福州 350108; 2.福建省地震局 地震学会,福州 350003)
(*通信作者电子邮箱 zouchangwei@fjnu.edu.cn)

摘 要:针对CortexM3微控制器(MCU)对传统51单片机的部分替代所带来的系统复杂度的增加问题,提出了一种在无操作系统支持的嵌入式平台上实现并发控制的方法。首先基于上下文无关文法形式化地定义了控制流程的脚本语言,并实现相应的脚本解释器;然后指出多线程机制是实现多脚本并发执行的充分条件而非必要条件,通过在MCU自带的定时器中断处理函数中进行并发脚本控制流程的切换,实现了一个能用于嵌入式平台并发编程的协程机制。实验结果表明,该机制能避免对商业多线程库的依赖,降低产品研发成本,在代码可读性方面也有较大提高,使代码烧写次数减少58%左右。在无操作系统和有Linux操作系统支持的嵌入式平台上的分别应用,表明该机制有较好的可移植性和实用性。

关键词:协程;脚本化机制;嵌入式系统;上下文无关文法;多线程
中图分类号:TP311.52 **文献标志码**:A

Coroutine and scripted mechanism for embedded system

ZOU Changwei[1*], WANG Lin[2]
(1. *Faculty of Software, Fujian Normal University, Fuzhou Fujian* 350108, *China*;
2. *Seismological Society, Earthquake Administration of Fujian Province, Fuzhou Fujian* 350003, *China*)

Abstract: Due to the increase of system complexity arising from the partial substitution of the traditional 51 Micro Controller Unit (MCU) with Cortex M3, a method was proposed for concurrent control on embedded system without operating system. First of all, a script language and its corresponding interpreter for concurrent control were implemented via context-free grammar formally; further an proposition was pointed out that multithreading was a sufficient but not necessary condition for concurrent scripts; meanwhile, a coroutine mechanism for concurrent programming on embedded system was constructed by switching script contexts in the interrupt service routine of MCU timer. The experimental results show that the proposed mechanism avoids the dependency on commercial multithreading libraries, which is helpful to decrease the cost of product, promotes the readability of source code and causes a flash programming frequency drop by about 58%. The respective applications on systems with and without operating system demonstrate that this mechanism is portable and practical.

Key words: coroutine; scripted mechanism; embedded system; context-free grammar; multithreading

0 引言

随着ARM公司Cortex M3核在工业界的流行,传统的由51系列单片机占主导地位的无需操作系统支持的一些嵌入式应用领域,也越来越多地被STM32为代表的基于Cortex M3核的微控制器(Micro Controller Unit, MCU)所替代。以51单片机为核心构建的嵌入式系统,限于flash的容量,其对应C语言源代码的行数少则几百行,较多地也一般在几千行这一规模,因代码量相对较少,其控制系统的编码任务经常由一名程序员独立完成,即便软件架构设计得不太合理,修改起来也相对容易。而以STM32为MCU构建的嵌入式系统的C语言源代码经常达到几万行。因产品开发周期的约束,这一规模的软件系统通常需要由3至5名程序员配合完成,这也对软件架构的合理性和可维护性提出了更高要求。

代码量的大幅增加通常也意味着相应系统和应用复杂度的激增,这很自然地带来了对多线程的需求[1]。而要在单片机平台上实现多线程机制,移植μC/OS (Micro Control Operating System)操作系统一直是这一领域的首选。在μC/OS操作系统中实现的多任务机制中,各任务并没有独立的地址空间,实际上仅相当于传统操作系统理论中处于同一进程空间里的多线程[2-3]。在低端的嵌入式领域,这样的多线程机制已经基本满足大部分应用的需要。尽管源代码公开,但μC/OS是一个商业软件,需要支付一定的费用才能在产品中使用,而低端的嵌入式应用对成本控制相当敏感,这也在某种程度上限制了μC/OS的应用。文献[4-5]侧重从硬件结构去讨论多线程机制的实现。文献[6-9]主要讨论了确定性多线程调度算法。但这些都需要较复杂的软硬件设施的支持,而适用于无操作系统支持的32位单片机领域的控制流程并发机制在这些文献中较少被涉及。文献[10]在Contiki OS系统中引入名为Protothreads的coroutine机制,该机制基于标准C来实现,不必定制编译器和C预处理器,而是利用switch语句的case分支来跳入if和while等控制结构中,进而在单个栈上实现协程的并发,避免了传统多线程机制中

收稿日期:2013-11-18;**修回日期**:2013-12-15。 **基金项目**:福建省中青年教师教育科研项目科技B类资助项目(JB13027)。
作者简介:邹昌伟(1981 -),男,福建龙岩人,讲师,硕士,主要研究方向:嵌入式系统、信息安全、二进制代码分析; 王林(1983 -),女,福建福州人,工程师,硕士,主要研究方向:地理信息系统。

多个栈所带来的内存开销,满足了内存受限领域对并发编程的需要。虽然 Protothreads 机制中引入的 PROCESS_BEGIN、PT_BEGIN 和 LC_RESUME 等宏对应用级程序员隐藏了其协程并发的实现机制,但利用 switch 语句来跳入 if 和 while 的编程方法并不太符合通常的结构化程序设计风格,一定程度上牺牲了代码的可读性。在 Cortex M3 这样的定时器资源很丰富的 32 位 MCU 上,完全可以利用定时器来实现协程的切换,而不必使用有违结构化程序设计的 switch 跳转手段,由此可增加协程代码的可读性。基于此,本文利用嵌入式 MCU 自带的定时器,提出了一个能用于嵌入式领域的协程机制。

在嵌入式控制系统的开发过程中,需要对各种控制命令与参数进行不断的调整,以期达到更好的控制效果,甚至在产品上市后,技术服务人员去做产品维护时,也需要对控制流程进行微调。而如果能把这些控制流程用简洁的脚本语言来描述,则意味着可以不用重新烧写代码,只需要通过图形用户接口(Graphical User Interface, GUI)界面或者串口等改变控制脚本,就可以改变系统的控制流程,这给嵌入式产品的开发和维护都能带来极大的方便。因此,本文在上下文无关文法的基础上实现了一个脚本语言,用于嵌入式控制流程的脚本化。

协程与脚本化机制的结合使用,为相关嵌入式系统的开发提供了一个较好的软件架构,有助于降低产品的开发成本,提高产品的竞争力。

1　控制流程的脚本化

在嵌入式领域,经常会遇到多个控制流程并发执行的情况,比如温度的控制和水路的控制可能都有其各自独立的控制流程。这里以 S 代表控制流程中的每一步 Step,每步具体的动作可以是阀控,调泵的脉冲宽度调制(Pulse Width Modulation, PWM)值,延时若干毫秒或者轮询直到某个条件满足。并发执行的 N 个控制流程可以抽象为以下形式:

控制流程 1: S_{11};S_{12};S_{13};S_{14};S_{15}; …

控制流程 2: S_{21};S_{22};S_{23};S_{24};S_{25}; …

⋮

控制流程 N: S_{n1};S_{n2};S_{n3};S_{n4};S_{n5}; …

具体动作中的参数,如延时的时间和 PWM 的值等在开发过程中要不断进行调优。如果把这部分参数硬编码到代码中,则参数的每次变更都要重新修改代码,然后再对 MCU Flash 重新烧写。把这些控制流程脚本化,能方便系统的修改与维护。因此,需要定义一套简单易用且功能完备的脚本语言来描述这些控制流程。结构化程序设计的经典理论已证明顺序结构、分支结构和循环结构可构成任意程序。同理,功能完备的脚本语言也需要这三种基本的控制结构。可用上下文无关文法形式化地定义脚本语言的语法[11],如下文法所示:

```
one_script_thread→ statement
statement → basicStatement | ifStatement | whileStatement |
  statementSeq
ifStatment → if_name arguments statement eif_name;
  |if_name arguments statement else_name statement eif_name;
whileStatement → while_name arguments statement ewhile_name;
basicStatement→ wait_num; | action_name arguments; |;
statemenSeq → statement statementSeq | statement
arguments → argument arguments | ε
```

其中:one_script_thread 为文法的开始符号,代表可由若干个动作构成的一个控制流程;statement 代表脚本中的语句,对应控制流程中一个或多个动作;语句 statement 又可以进一步分为基本语句 basicStatement、条件语句 ifStatement、循环语句 whileStatment 和语句序列 statementSeq;ifStatement 中的 name 代表脚本语言的编写者给 if 语句取的唯一的名字,可以是由字母或数字构成的任意字符串,用于区别脚本中不同的 if 语句,同时该 name 也作为脚本解释器[12-13]选择回调函数的依据。同一条 if 语句中的 if_name、else_name 和 eif_name 中的 name 必须一致,一致性的检查可在检测输入的脚本是否合法时进行。词法分析时 if_name 当成 if_和 name 这两个 token。这样的处理虽然在语法结构上有冗余,但可简化和加快对脚本的语法分析,不需要产生任何中间代码,也不必记录语句的上下文信息,只要作简单的字符串匹配即可找到 if 语句的不同分支和结束位置。循环语句 whileStatement 中的 name 作类似处理,ewhile_name 代表名为 name 的 while 语句的结束。基本语句 basicStatement 由延时语句 wait_num、动作语句 action_name 和空语句构成,wait_num 中的 num 代表延时的具体时间,action_name 中的 name 代表脚本编写者给特定动作取的名字。arguments 表示若干个由字符或数字构成的动作参数 argument。ε 代表空串,即允许不带任何参数。

文法中的 if_name、while_name 和 action_name 中的 name 所代表的具体条件或动作由脚本解释器通过回调函数的形式提供给程序员进行实现,wait_num 的延时操作则需要协程调度机制与脚本解释器配合实现。脚本解释器所需的数据结构与算法如下所示:

```
//用于描述脚本中一条 statement 对应的回调信息
struct call_back_info{
  int type;          //语句类型, 如 IF, WHILE, ACTION, 或 WAIT
  const char * name;        //脚本中的 if, while 或 action 的名称
  void * arg;               //脚本中的 if, while 或 action 的参数
  CALL_BACK func;                          //回调函数的地址
};
//由若干条 statement 构成的控制流程的脚本上下文
struct script_context {
  call_back_info * call_backs;
                  //指向由多个 call_back_info 对象构成的数组
  const char * cur_pos;
                  //脚本的当前执行位置, 相当于程序计数器
  const char * from;                       //脚本的开始位置
  int wait_num;                    //还未执行完的延时时间
}
//脚本解释器的算法
int script_interpreter( const char * script, script_context * context) {
  while( 当前 token 不是脚本结束符 EOF) {
    if ( 当前 token 是 IF_ 或者 WHILE_) {
                                        // if 或 while 语句的开始
      取下一个 token 作为该 if 或 while 语句的 name
      如果有附带参数 arguments, 则保存 arguments 到脚本上下
        文 context 中
      从脚本上下文 context 中取出该 if 或 while 语句对应的回调
        函数 judge
      if ( judge( arguments) 为真) {
        跳转到该 if 或 while 语句的真出口
      } else{
        跳转到该 if 或 while 语句的假出口
      }
    } else if( 当前 token 是 ELSE_) {
                          //说明已经执行完 if 语句的真分支
```

```
    取下一个 token 作为 if 语句的 name
    跳转到 eif_name 处;
  }else if( 当前 token 是 EIF_) {                    //if 语句的结束
    跳过 eif_之后的 name;
  }else if( 当前 token 是 EWHILE_) {          //while 语句的结束
    取下一个 token 作为该 while 语句的 name
    //需要再次判断 while 语句的条件是否成立
    跳转到脚本中的 while_name 处
  }else if( 当前 token 是 WAIT_) {                    //延时操作
    取下一个 token 作为延时的参数
    在脚本上下文 context 中记录当前的执行位置和延时时间
    //在协程机制中完成延时, 返回 NOT_FINISHED, 代表脚
    //本未执行完
    return NOT_FINISHED;
  }else if( 当前 token 是 ACTION_) {                  //动作
    取下一个 token 作为该动作的 name
    取后续的 token 作为该动作的 arguments
    从脚本上下文 context 中取出该动作对应的回调函数
      action_func
    执行函数 action_func( arguments) ;
  }else if( 当前 token 是 SEMICOLON) {        //分号, 即空语句
    取下一个 token;
    continue;
  }
 }
 //返回 FINISHED, 代表当前脚本的所有控制流程都已经执行完
 return FINISHED;
}
```

2 协程机制

若有并发运行的温度控制和水箱控制流程如下:1)如果温度不到 70℃则开加热器加热 1 s, 然后再关加热器, 延时 2 s后循环判断;2)水箱如果未满, 则判断净化后的超纯水电阻值是否大于 16 MΩ(代表产水合格), 如果是则开阀 1 进水 5 s, 若不合格则关阀 1, 隔 5 s 后再判断。由前文的文法, 可以写出以下两个脚本分别描述这两个相互独立的控制流程。

温度控制流程 temper_control:

```
while_blow_degree 70
  action_heating on;
  wait_1000;
  action_heating off;
  wait_2000;
ewhile_blow_degree;
```

水箱控制流程 tank_control:

```
while_tank_not_full
  if _water_ready 16
    action_valve on 1;
  else_water_ready
    action_valve off 1;
  eif_water_ready;
  wait_5000;
ewhile_tank_not_full;
```

这是两个需要并发执行的 while 循环, 若在单线程的执行环境下, 要么温度控制先执行, 要么水箱控制先执行, 很难做到两者的并发。直观地, 引入多线程机制可使这两个控制流程能在各自独立的线程上下文中并发运行。在上述两个控制流程中, 开启加热模块、关闭加热模块、开阀和关阀等控制动作在 STM32 上一般可在微秒级时间内完成, 而大量的时间是耗在延时上, 用以满足嵌入式系统对控制时序的要求。此处对多线程机制的直观需求实际上源于各控制流程都需要独立地延时一段时间, 延时结束后, 还要能回到各自控制流程延时操作之后的位置开始执行。实际上, 在嵌入式系统中, 控制的焦点通常是时序的控制。而在单 MCU 上并发执行的多控制流程需要共享 MCU 的时间片, 所以此时各控制流程中一定是存在或多或少的主动或者被动的延时状态。主动的延时是由脚本编写者显式地添加的,如上述水箱控制流程中的 wait_5000;而被动的延时是因为共享 MCU 而隐式地存在的。各控制流程需要保存的上下文信息至少要包括脚本中的当前执行位置和当前主动的延时时间。

本文称这样的为实现脚本并发执行而保存的上下文信息为脚本上下文, 即前文的 struct script_context 所表述的信息。操作系统理论中的多线程机制满足了保存脚本上下文的需要,但多线程机制是实现多脚本并发执行的充分条件而非必要条件。只要为每个控制流程分配一个 script_context 对象, 就可以使各控制流程有独立的脚本上下文,问题的关键在于如何实现脚本上下文的切换。这主要涉及到脚本上下文切换的时机问题。嵌入式系统控制流程中的实际控制动作多由设置通用输入输出(General Purpose Input/Output, GPIO)口的高低电平或者对外设寄存器的操作来进行, 相对于为确保时序而引入的延时操作, 这些控制动作本身所耗时间实际上都不长, 所以本文选择在控制流程执行延时操作时进行脚本上下文的切换。可用以下 coroutine_thread 结构体来描述具体的一个控制流程:

```
struct coroutine_thread{
  THREAD_FUNC func;                              //协程的函数地址
  void * arg;              //协程的函数参数, 一般是脚本上下文
  int status;                  //协程的状态: 有效/失效等状态
}
```

前文中的温度控制流程就可描述为以下代码:

```
int temper _control( void * arg) {                    //温度控制
  script_context * context = ( script_context * ) arg;
  //调用脚本解释器对控制流程解释执行。
  script_interpreter( 当前执行位置 context -> cur_pos,
    脚本上下文 context) ;
}
```

可把函数地址 temper_control 和与其相关的脚本上下文对象 context 存于一个 coroutine_thread 对象中。协程机制内部会构建一个容器, 用于存放并发控制流程对应的多个 coroutine_thread 对象。至此,脚本上下文的切换问题就聚焦到如何实现并发控制流程的延时操作上。现代的 MCU 片内都有丰富的定时器资源,可从中任选一个定时器用来实现延时操作。这个定时器可配置成每隔若干毫秒溢出一次, 这个值通常设为 1 ms 或者 10 ms。不妨设这个时间为 SYS_TICK 毫秒。每隔 SYS_TICK 毫秒该定时器的中断处理函数 sys_tick_timer_isr()就会被执行一次。在该中断处理函数中实现了协程的调度机制。

```
//协程调度机制的算法:
void sys_tick_timer_isr( ) {
  for( 对容器中的每一个有效的 coroutine_thread 对象 pthrd) {
    if ( pthrd 对应脚本上下文中的延时时间不为 0) {
      //此时该控制流程处于延时操作中, 主动放弃 MCU
      延时时间 -= SYS_TICK;
```

```
    }else{
      //调用脚本解释器，从保存的断点开始继续执行脚本
      status = pthrd ->func(pthrd ->arg);
      if(status == FINISHED){
        //该控制流程的脚本已经全部执行完毕
        标志该协程 pthrd 的状态为失效，代表其生命周期已结束
      }
    }
  }
}
//开启新的协程
void start_coroutine_thread(){
  设置新的 coroutine_thread 对象，初始化其脚本上下文和函数
    地址等信息;
  关 MCU 中断;
  往容器中加入该 coroutine_thread 对象，置其状态为有效，以便
    参与调度;
  开 MCU 中断;
}
//结束协程
void stop_coroutine_thread(pthrd){
  关 MCU 中断;
  置该对象的状态位为无效，不再参与调度;
  开 MCU 中断;
}
```

由算法可见,各控制流程是在 MCU 的定时器中断处理函数中执行的。多个控制流程在宏观上并发,而在微观上是串行的。之所以称本文提出的机制为协程机制是因为:1)只要这些控制流程中有延时操作，本文的机制确实可让这些控制流程从表面上看起来像是在多线程中并发执行; 2) 而若从是否具有独立的 CPU 上下文来看,各控制流程实际上都是处于 MCU 的中断上下文，并没有自己独立的线程上下文,这与操作系统理论中的线程还是不太一样。最明显的例证是在真正意义上的多线程机制中，可以在各线程中分别执行"while(1);"这样的死循环操作,这些线程的时间片用完时,会被调度机制剥夺 CPU,换言之,对被剥夺 CPU 的线程而言,此处的死循环实际上也是存在延时的,不过这种延时是以被动形式隐式存在的。而本文实现的协程机制中,若有一个控制流程执行形如"while(1);"这样的不带延时操作的死循环，则整个 MCU 会在定时器中断处理函数中陷入死循环。要避免这个问题,脚本编写者可在此类死循环中主动地加入少量延时操作,或者在检测输入脚本合法性时由脚本化机制隐式地添加少量延时操作。

从抢占与否的角度来看，上述算法是非抢占式的，即只有当前控制流程主动通过延时操作放弃当前 MCU 时，系统才会切换到下一个控制流程中。延时操作是以定时器的溢出周期 SYS_TICK 为最小刻度，该参数值的选取应与实际应用相吻合。而控制流程的脚本化意味着这些脚本可在代码中以字符串的形式存在，脚本的当前执行位置也是以字符串地址的形式存在，脚本上下文切换时只要保存这个字符串地址即可，不必保存 MCU 的 CPU 上下文，比如程序计数寄存器。通常需要特定的 MCU 指令集的支持才能保存 CPU 上下文。本文实现的协程和脚本化机制只在设置定时器的中断向量和中断开关操作这两个方面跟具体的 MCU 相关，其余的大部分模块都可跨平台使用。这样,与在新的 MCU 平台上移植 μC/OS 等系统相比，本文提出的协程机制在移植上会更加简单易用。而在 Linux 和 Windows 平台上，只要利用这两个操作系统提供的软定时器来模拟嵌入式 MCU 提供的硬件定时器，用操作系统提供的互斥体操作来模拟 MCU 的中断开关操作，本文提出的协程和脚本化机制也可很容易地移植到这些平台上。

3　实际应用及分析

利用本文提出的协程与脚本化机制，在 STM32F103(频率 72 MHz)平台上完成了连续喷墨(Continuous Ink Jet，CIJ)喷码机的设计与实现。协程机制的引入，满足了 CIJ 喷码机这样复杂的实际应用对并发控制流程的需要，不需要购置嵌入式的商业多线程库，从而降低了产品成本，提高了产品的竞争力。同时，在产品的研发过程中，控制流程的脚本化也给调试带来了极大的方便，尤其是到了研发的中后期，整体控制流程已经相对稳定，此时脚本化带来的最明显好处就是代码烧写的次数明显减少，虽然代码烧写次数跟程序员编程习惯和具体产品密切相关,但是初步的统计显示,该次数可由之前的人均每日 12 次左右的烧写次数降低到人均每日 5 次左右,大约降低了 58% 左右。原因在于只需要通过 GUI 界面或者串口修改脚本，就可以改变系统的控制流程，缩短了调试时间，提升了研发和测试的效率。之后,又在以 Omap3530(频率 600 MHz)为 CPU 的 Linux 平台上完成了自来水净化系统的设计和实现，利用 Linux 操作系统提供的 signal 机制来充当软定时器,signal 处理函数充当中断处理函数的角色,pthread_mutex_lock 和 pthread_mutex_unlock 来模拟中断开关操作,在做很少代码改动的情况下,就能把已在 STM32F103 平台上用标准 C 实现的协程与脚本化机制移植到 Linux 平台上。Linux 平台上已有 pthread 等线程库，移植本文机制到 Linux 平台的主要出发点是把控制流程脚本化。在有文件系统支撑的操作系统上，这些脚本很自然地会以文件的形式存在。

控制流程脚本化，意味着嵌入式产品的最终用户可任意调整系统的控制流程，在带来灵活性和方便性的同时也给系统的安全性带来一定挑战。通过给终端用户设置不同的权限和密码，可在一定程度上缓和安全的问题。

4　结语

本文提出了一个用于嵌入式领域的协程与脚本化机制，该机制设计的初衷是针对无操作系统支持的低端 32 位单片机应用领域，即 ARM 公司 M 系列芯片所定位的领域。经实际产品的检验，该机制实现了降低产品成本、提高研发效率和增加系统可维护性的初始目标。而通过在 Linux 操作系统支持的高端应用上移植该机制，也表明了该机制有较好的通用性。

脚本化机制与协程机制是松耦合的，脚本化机制完全可以在真正多线程机制的支持下运行。本文实现的协程机制尚未支持不同的优先级。非抢占的调度也不适合对实时性要求很高的场合。所设计脚本语言中也没有支持 break 和 continue。脚本解释器的编写和脚本上下文对象的生成也是手工编码，下一步的工作是考虑利用 LEX 和 YACC 等自动生成器来简化词法和语法分析,以减少机械化的工作量。

参考文献：

[1] STALLINGS W. Operating systems: internals and design principles [M]. Upper Saddle River: Prentice Hall, 2004.

[2] LI Y, CUI X, LI X, *et al.* Hardware implementation of task management of μC/OS-Ⅱ[J]. Journal of Computer Applications, 2010, 30(5): 1386 - 1389. (李岩, 崔晓英, 李贤尧, 等. μC/OS-Ⅱ任务管理的硬件实现[J]. 计算机应用, 2010, 30(5): 1386 - 1389.)

[3] ZHOU B, WANG S, QIU W, *et al.* SHUM-UCOS: a real-time operation system for reconfigurable systems using uniform multi-task model [J]. Chinese Journal of Computers, 2006, 29(2): 208 - 218. (周博, 王石记, 邱卫东, 等. SHUM-UCOS: 基于统一多任务模型可重构系统的实时操作系统[J]. 计算机学报, 2006, 29(2): 208 - 218.)

[4] DENG Y, SUN Y, WAN J. Research and design of multiple thread mechanism in Matrix DSP [J]. Computer Science, 2013, 40(4): 51 - 54. (邓宇, 孙永节, 万江华. Matrix DSP 中多线程机制的研究与设计[J]. 计算机科学, 2013, 40(4): 51 - 54.)

[5] YIN Z, ZHAO H, ZHANG W, *et al.* Implementation of a hardware scheduled multithread processor for embedded system [J]. Journal of Northeastern University: Natural Science, 2006, 27(9): 1386 - 1389. (尹震宇, 赵海, 张文波, 等. 一种嵌入式硬件多线程处理器的研究[J]. 东北大学学报: 自然科学版, 2006, 27(9): 1386 - 1389.)

[6] CUI H, WU J, GALLAGHER J, *et al.* Efficient deterministic multithreading through schedule relaxation [C]// SOSP '11: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. New York: ACM, 2011: 337 - 351.

[7] BERGAN T, ANDERSON O, DEVIETTI J, *et al.* CoreDet: a compiler and runtime system for deterministic multithreaded execution [J]. ACM SIGARCH Computer Architecture News, 2010, 38(1): 53 - 64.

[8] MA C, YIN J, JIANG L, *et al.* Deterministic multithreaded scheduling based on long parallel distance priority [J]. Journal of Chinese Computer Systems, 2012, 33(10): 2177 - 2181. (马超, 尹杰, 江凌波, 等. 基于长并行距离优先的确定性多线程调度[J]. 小型微型计算机系统, 2012, 33(10): 2177 - 2181.)

[9] LIU T, CURTSINGER C, BERGER E D. Dthreads: efficient deterministic multithreading [C]// SOSP '11: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. New York: ACM, 2011: 327 - 336.

[10] DUNKELS A, SCHMIDT O, VOIGT T, *et al.* Protothreads: simplifying event-driven programming of memory-constrained embedded systems [C]// Proceedings of the 4th International Conference on Embedded Networked Sensor Systems. New York: ACM, 2006: 29 - 42.

[11] AHO A V, LAM M S, SETHI R, *et al.* Compilers: principles, techniques, and tools [M]. Upper Saddle River: Pearson Education, Inc, 2006.

[12] GUAN S, ZHANG Z, XU L, *et al.* Implementation of automatic generating scripting language interfaces with C/C++ code [J]. Computer Engineering, 2005, 31(14): 102 - 104. (官尚元, 张芝萍, 徐立锋, 等. C/C++ 代码自动生成脚本语言接口的实现[J]. 计算机工程, 2005, 31(14): 102 - 104.)

[13] SUN J, TAN X, ZHAO H, *et al.* Research and implementation of ball games scenario interpreter [J]. Journal of Computer Applications, 2010, 30(3): 612 - 614. (孙晶, 谭效辉, 赵会群, 等. 球类比赛的脚本解释器的研究与实现[J]. 计算机应用, 2010, 30(3): 612 - 614.)

(上接第1399页)

参考文献：

[1] KENNEDY J, EBERHART R. Particle swarm optimization [C]// Proceedings of the 1995 IEEE International Conference on Neural Networks. Piscataway: IEEE Press, 1995, 4: 1942 - 1948.

[2] SHI Y, EBERHART R. A modified particle swarm optimization [C]// Proceedings of the 1998 IEEE World Congress on Computational Intelligence Evolutionary Computation. Piscataway: IEEE Press, 1998: 69 - 73.

[3] LIU B. Particle swarm optimization and its engineering application [M]. Beijing: Electronic Industry Press, 2010: 48 - 50. (刘波. 粒子群优化算法及其工程应用[M]. 北京: 电子工业出版社, 2010: 48 - 50.)

[4] FANG Z, TONG G, XU X. Particle swarm optimized particle filter [J]. Control and Decision, 2007, 22(3): 273 - 277. (方正, 佟国峰, 徐心和. 粒子群优化粒子滤波方法[J]. 控制与决策, 2007, 22(3): 273 - 277.)

[5] ZHANG W. Hybrid optimization algorithm of ant colony and particle swarm with application [D]. Tianjin: Tianjin University, 2007: 22 - 40. (张维存. 蚁群粒子群混合优化算法及应用[D] 天津: 天津大学, 2007: 22 - 40.)

[6] JIA D, ZHANG J. Niche particle swarm optimization combined with chaotic mutation [J]. Control and Decision, 2007, 22(1): 117 - 120. (贾东立, 张家树. 基于混沌变异的小生境粒子群算法[J]. 控制与决策, 2007, 22(1): 117 - 120.)

[7] LIU K, TAN Y, HE X. Particle swarm optimization based learning algorithm for process neural networks [J]. Acta Scientiarum Naturalium Universitatis Pekinensis, 2011, 47(2): 238 - 244. (刘坤, 谭营, 何新贵. 基于粒子群优化的过程神经网络学习算法[J]. 北京大学学报: 自然科学版, 2011, 47(2): 238 - 244.)

[8] QIN Y. Inertial navigation [M]. Beijing: Science Press, 2006: 355 - 360. (秦永元. 惯性导航[M]. 北京: 科学出版社, 2006: 355 - 360.)

[9] GAO S, LI W, SANG C. SINA/SAR integrated navigation system using quaternion model [J]. Journal of Chinese Inertial Technology, 2010, 18(1): 63 - 69. (高社生, 李伟, 桑春萌. 基于四元数的 SINS/SAR 组合导航系统[J]. 中国惯性技术学报, 2010, 18(1): 63 - 69.)

[10] CHANG M. Study on the integrated navigation of SINS and global positioning system [D]. Chongqin: Chongqin University, 2004: 42 - 46. (常明飞. GPS/SINS 组合导航系统研究[D]. 重庆: 重庆大学, 2004: 42 - 46.)

[11] LIU J, ZENG Q, ZHAO W, *et al.* Theory and applications of navigation system [M]. Xi'an: Northwestern Polytechnical University Press, 2010: 350 - 354. (刘建业, 曾庆化, 赵伟, 等. 导航系统理论与应用[M]. 西安: 西北工业大学出版社, 2010: 350 - 354.)

[12] PAN S, WANG S, SUN J. Test data generation based on *K*-means clustering and particle swarm optimization [J]. Journal of Computer Applications, 2012, 32(4): 1165 - 1167. (潘烁, 王曙燕, 孙家泽. 基于 *K*-均值聚类粒子群优化算法的组合测试数据生成[J]. 计算机应用, 2012, 32(4): 1165 - 1167.)

[13] WU L, ZHAO H, WANG T, *et al.* New particle swarm optimization algorithm with global-local best minimum [J]. Journal of Computer Applications, 2009, 29(12): 3270 - 3272. (吴琳丽, 赵海娜, 汪涛, 等. 新的全局—局部最优最小值粒子群优化算法[J]. 计算机应用, 2009, 29(12): 3270 - 3272.)