嵌入式实时操作系统中的内存管理技术

王金德 20721156

2008-04-28

摘 要

实时计算正在成为越来越重要的原则。嵌入式操作系统常常是实时系统。为保证嵌入式实时操作系统中数据的安全性和可靠性，相关的内存管理技术研究必不可少。

本文针对嵌入式实时操作系统的特性，系统地分析了实时操作系统中内存管理的需求和特点。在此基础上，本文对当前一系列有助于提高系统安全性和可靠性的内存管理技术进行了研究，主要包括动态内存泄漏检测及回收、内存容错及冗余技术和内存保护技术。

本文对这些技术探讨的同时，对这些技术能解决的问题和优缺点作了总结，为以后的相关研究提供一定依据。

关键词：嵌入式，实时系统，内存管理

目 录

第 1 章 引言

1.1 相关概念

随着多媒体网络技术发展及电子产品智能化，嵌入式操作系统以其稳定性、伸缩性、可扩展性等优良特性，在嵌入式产品中得到了广泛的应用。多数嵌入式产品往往对实时性有着一定的需求，而事实上，很多嵌入式操作系统本身都是实时系统。所谓的实时系统，是指必须在一个给定的时间期限之前完成计算任务的系统。

实时系统根据其时限要求和对超时错误码的容忍程度不同，分为硬实时系统和软实时系统。硬实时系统具有强实时约束，它对作业的运行有一个严格的时间限制，任何超出时限的错误都会导致灾难性后果。软实时系统具有弱实时约束，它对计算任务有时限要求，但一般的超时并不会引起严重的后果，随着延迟的作业增多，系统整体性能可能有所下降[1]。如图1.1①。

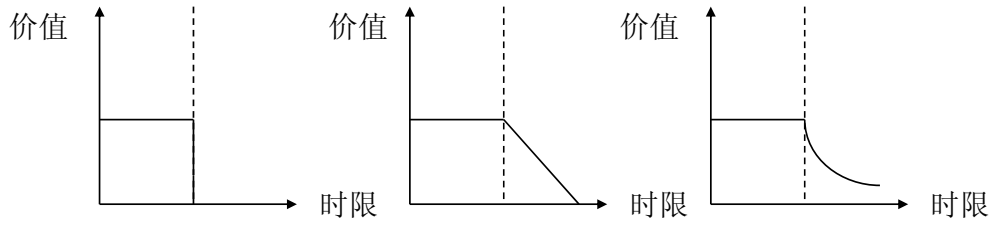

图 1.1 实时系统的不同实时性特点

目前世界范围内嵌入式实时操作系统产品非常多，应用十分广泛，其中著名的有Nucleus Plus、VxWorks、QNX、pSOS、LynxOS、RT-Linux、IxCLinux、IXC/OS-II、eCo等。而国内也有一些产品，如科银京成的DeltaOS、中科院的EEOS、凯思集团的HopenOS等，尽管技术含量相对较低。

对于嵌入式实时操作系统来说，最重要的是必须能够满足在一个事先定义好的时间限制中对外部或内部的事件进行响应和处理。相应地，它对于内存管理也有着特殊的要求[2]：

①. 该图来自于浙大嵌入式系统硕士课程实时系统课件，陈天洲教授。

响应时限 内存分配和释放的最坏响应时间是提前设定的，并且与具体的应用无关。

快速性 对于实时系统来说，内存管理仅仅保证硬实时约束是不够的，还必须保证快速性，保证简单快速的分配和释放内存。这决定了实时系统中不能采用像通用操作系统那样复杂而完善的内存分配策略。

可靠性 系统对于内存的请求必须得到满足。对于实时应用来说，内存请求的无法满足，可能会带来灾难性的后果。

高效性 在嵌入式系统中，内存是珍贵的资源，必须高效地加以利用，以做到浪费的内存尽可能地小。

另外，内存管理中的内存分配，主要可分为静态分配和动态分配。静态分配是指在编译或链接时将程序所需的内存空间进行分配，程序运行过程中能且只能使用这些内存。动态分配是指内存在程序运行时根据需要向系统申请后获得。相比较而言，静态分配注重效率和安全，动态分配则偏重于灵活性。

1.2 相关研究和意义

由于嵌入式实时操作系统实时的特点，它决定了内存管理单元的响应时限、快速性、高效性和高效性这四个特性。从内存分配角度来说，对于实时性和可靠性要求极高的系统（硬实时系统），不允许延时或者分配失效，必须采用静态内存分配。但是，仅仅采用静态分配，系统便失去了灵活性。虽然动态内存分配会导致响应和执行时间不确定、内存碎片等问题，但是它的实现机制灵活，给程序实现带来极大的方便，有的应用环境中动态内存分配甚至是必不可少的。

正是因为动态分配的灵活性，国内外对动态内存分配的研究也相对较多。论文[3]对嵌入式实时操作系统的动态内存分配进行了研究，提出一种新的分配机制，称之为TLSF（Two Level Segregated Fit）。TLSF是专门为实时系统设计的动态内存分配机制，它考虑实时系统时间和空间两个维度的约束。TLSF的时间复杂度是O(1)，并且保证了较好的空间复杂度。论文[2]进一步对TLSF内存分配机制进行了研究，把它与已知的嵌入式实时操作系统上的各种内存动态分配机制进行了比较，并且得出，TLSF的空间复杂度也是已知的分配机制中最好的。论文[4]更是设定了一个衡量框架，系统地分析和比较了各种动态内存分配机制的优势和劣势，主要集中于时间复杂度和内存碎片的角度。

这些论文着重讨论动态内存分配的同时，也讨论了实时系统安全性和可靠性相关的一些内存分配技术。从目前的国内外研究现状来看，这些技术一般从以下三个方面着手：

- 动态内存泄漏检测及回收；
- 内存容错及冗余技术；
- 内存保护技术。

另外，论文[5–7]讨论了内存泄漏检测及回收相关内容，而这些内容可以使用在特定的嵌入式实时系统中，解决这方面的问题。论文[8,9]讨论了一些容错及冗余技术。论文[10]讨论了内存保护相关的技术，主要是基于MMU的内存保护。

本文根据嵌入式实时操作系统的特点，及对内存管理的特殊要求，着重针对于一系列有助于提高系统安全性和可靠性的内存管理技术进行了深入的研究，主要包括动态内存泄漏检测及回收、内存容错及冗余技术和内存保护技术。本文对这些技术研究的同时，得出结论，为以后的相关研究提供一定依据。

第 2 章　内存管理技术

2.1　内存泄漏检测及回收

内存泄漏是指程序堆中已经动态分配的内存，由于某种原因未得到释放或无法释放，造成系统内存浪费，最终可能导致一些严重后果[7]。内存泄漏是隐蔽的，并且不断积累，与其他内存非法访问错误相比，更难检测。内存泄漏通常不会产生直接可观察的错误症状，而是逐渐积累，降低系统性能，极端的情况下可能使系统崩溃。特别对于嵌入式系统来说，内存泄漏往往使得系统失去使用价值。

目前与动态内存错误检测相关的方法主要有动态方法和静态方法两类。

- 动态方法需要在程序运行过程中进行潜在错误的检测，如Purify①就是一个动态的单元测试工具。Purify可以检测到程序中存在的内存泄漏问题，但对于其他的动态内存错误，如空指针的引用、动态内存的重复释放等却无能为力。Purify需要在程序的运行中进行内存泄漏的检测，由于程序中分支结构的存在，程序的一次运行不能覆盖所有的程序分支，从而在一定程度上降低了检错效率。动态方法的一个缺点是常常会显著增加系统运行时的负荷。
- 静态方法通过对程序结构的分析来检测程序中隐藏的动态内存错误，如Lint②、Splint等都是通过在源程序中添加注释语句来检测应用程序中存在的潜在错误。这些工具的检错效果与应用程序中的注释量有关，并且它们经常会给出错误的信息，影响错误检测工作的顺利进行。

在嵌入式应用环境中，资源通常是非常有限的。特别对于嵌入式实时操作系统来说，性能和安全是至关重要的。利用运行时的动态内存泄漏检测方法开销很大，所以不适用于在实际的嵌入式产品。但是该方法可以使用在嵌入式系统的测试阶段，检测内存泄漏错误。静态方法则适用于嵌入式环境，但由于其自身的缺点，能力有限。

①. IBM提供的工具，URL：http://www-306.ibm.com/software/awdtools/purify/
②. 静态检测的工具，URL：http://docs.sun.com/source/806-3567/lint.html

对于动态内存泄漏检测方法，它常常是基于对动态内存跟踪[6]来实现的。对动态内存进行跟踪通常有三种方法：

- 重写编译器。这种方法很费时并且代价很大，所以极少被采用。
- 在程序执行或者链接的过程中直接插入目标代码。这种方法在不破坏程序逻辑和符号表的前提下，往已经编译好的目标代码中插入动态内存跟踪指令。这种方法的不足之处在于其适用性不强，在不同的指令集甚至不同的操作系统之间无法方便的移植。
- 包装源代码。这种方法包括解析、分析及把原来的代码转化成新的代码。它比前面两种办法方便高效，但对源代码的改变，难以保证不影响原来系统功能，需要更多的测试代价。

为了回收泄漏内存，常常需要一个垃圾回收机制。由于考虑到嵌入式实时系统的性能，不同的具体应用，往往实现不同的回收泄漏内存的机制。这些机制为了保证性能，并不需要非常复杂和全面。一般来说，为了能够回收泄漏内存，系统必须对任务申请的内存块进行跟踪。

2.2　内存容错及冗余技术

容错技术是指系统对错误的容忍技术，指处于工作状态中的系统中一个或多个部分发生错误时，系统能自动检测与诊断，并能采取相应的措施保证系统正常运行的技术[9]。容错技术能够达到对错误的容忍，但并非无视错误的存在。它首先要能自适应地检测并诊断系统的错误，然后采取对错误控制或处理的策略。因此容错技术包括以下两部分：

- 系统错误检测与诊断。当系统产生错误时能自动发现系统错误，并确定出错误的类型、位置、原因等。故障检测与诊断的成功与否直接影响系统的容错能力和可靠性，它是自适应运行的。
- 系统错误控制与决策。它针对错误的类型、位置、原因等信息采取相应的容错处理，在检测与诊断出系统的错误后立即决策出处理错误的方案并采取行动。目前常采用的处理方案是通过冗余资源来置换错误的单元，使系统继续正常工作。

简单地说，冗余就是指多余的资源。当系统中的一部分出现错误时，可以由冗余的部分顶替错误的部分工作，以保证系统在规定的时间内完成任务[9]。冗余

主要分为硬件冗余、软件冗余、信息冗余和时间冗余：

硬件冗余 指系统中某个或几个关键单元除了工作所需的基本单元外，另外设置一个或者一个以上的冗余单元。这些冗余单元可以与工作单元同时工作，也可以处于等待状态。一旦工作单元出现错误就可以顶替错误单元继续工作。硬件冗余是系统容错技术的基础。

软件冗余 通常针对计算机系统而言，将关键的软件部件复制多份保存在存储器。

信息冗余 指利用各种传感器或多种系统之间存在的已知函数关系来产生冗余信息，从而实现检测和识别错误。这种冗余多用于动态容错技术中的错误检测。

时间冗余 指重复执行某段程序的方法来检测错误或使系统从错误中恢复工作。

冗余设计是提高系统安全性与可靠性的一种方式，当采用其它方法不能满足系统可靠性要求时，可以考虑采用冗余设计。不过冗余设计需要更多的资源。

由于嵌入式实时应用常用于工控、航天等领域中，可能会面临恶劣的工作环境。比如在航天应用中，系统可能会受到宇宙射线等电磁干扰，导致内存信息发生翻转，破坏数据的一致性。对于一些关键的数据，这样的翻转可能会导致灾难性的后果，使系统发生不可预测的行为，更严重的可能导致系统崩溃。因此，在嵌入式实时操作系统中，对关键数据的保护显得尤为重要。这种保护常可通过内存冗余分配和内存冗余编码功能来实现。

- 内存冗余分配是针对系统中的关键数据，在内存中分配若干块冗余内存，以达到容错目的。当内存中关键数据因为某些原因发生错误时，系统可以通过一定方法，从冗余块中恢复关键数据。在这方面，还需要制定相应的比对策略和同步技术。
- 内存冗余编码是指对系统中关键数据增加校验码，通过校验码来对内存关键数据进行检错和纠错。常用的编码如海明码，可以满足内存冗余编码的需求。

2.3　内存保护技术

在嵌入式实时系统内存管理中，要实现内存的保护，常常依赖于内存管理单元（MMU）。MMU负责将内存的逻辑地址转换为物理地址，保护内存不会被非

法访问。本节所讨论的正是基于MMU的内存页面保护技术。如图2.1为高速缓存的MMU存储器系统①。

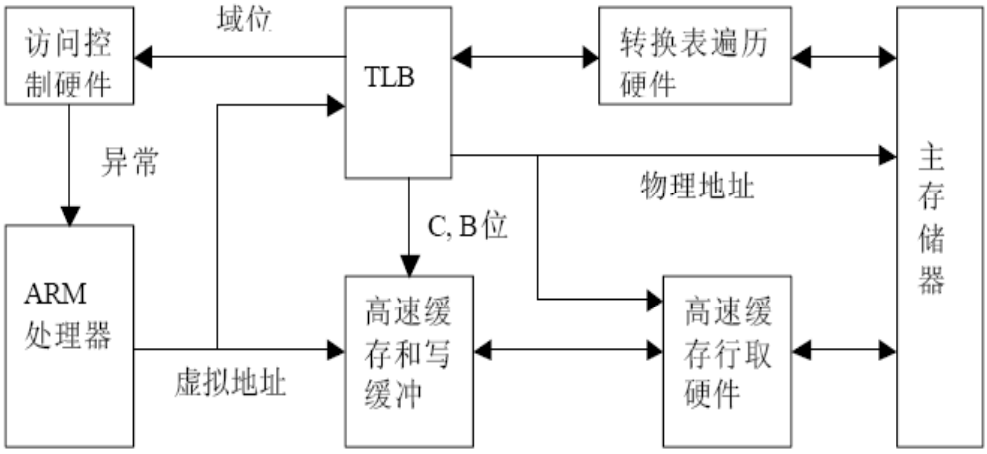

图 2.1　高速缓存的MMU存储器系统

在嵌入式实时系统内存管理中，对内存的保护主要有：

- 任务堆栈段的保护。任务的堆栈段通常存放在一个全局区域内，如果不采取任何措施的话，可以被其他任务访问改写。这种保护可以通过MMU支持，任务通过改变内存页面属性实现。
- 中断向量表的保护。在嵌入式系统运行中的内核模式下，应用程序可能对中断向量表非法改写，这也是容易发生的一个错误。这可以设置中断向量表所在内存页面属性为不可写。非法访问将产生一个TLB错误。
- 代码段的保护。如果程序代码段被意外修改，可能会导致致命的错误。这极易因为内存溢出等原因发生，也可能被人为地利用修改。这同样可通过设置代码段页面不可写实现。
- 共享数据的保护。任务之间经常需要以在保护状态下共享数据。需要这种保护的例子很多，比如经典的生产者－消费者模型。实现这种保护的方法也很多，比如信号量、监视器等。

①. 该图来自于浙大嵌入式系统硕士课程内存管理课件，陈天洲教授。

第 3 章　总结与展望

嵌入式操作系统得到了很广泛的应用，它们本身往往对于实时性有一定要求，多数都是实时系统。实时系统根据其时限要求和对超时错误码的容忍程度不同，分为硬实时系统和软实时系统。嵌入式实时操作系统对于内存管理有相应的要求，主要包括响应时限、快速性、可靠性和高效性。

本文对内存分配中的动态分配和静态分配进行了比较讨论，并着重探讨了与实时系统安全性和可靠性相关的一些内存分配技术，主要包括动态内存泄漏检测及回收、内存容错及冗余技术和内存保护技术。这些技术针对于不同的问题设计，在一定程度上解决各自问题的同时，也存在着一些缺陷。

由于嵌入式实时系统内存管理涉及诸多技术，而这些技术很多都处于不断成熟和完善阶段，所以，今后还有很多问题有待研究，这些研究包括：

- 静态和动态内存分配算法的研究。静态内存分配算法常用于硬实时系统中。而对于动态内存分配算法，由于嵌入式实时操作系统中实时的要求，在利用动态的灵活性的同时，仍然存在很多问题未解决。
- 系统整体性能的保证。由于加入了一些额外的安全性和可靠性相关的内存管理技术，这些技术需要消耗一定的系统资源，并且对系统的性能有一定影响，所以如何保证原先的系统性能是一个挑战。
- 新的内存管理技术。随着嵌入式环境和需求的不断发展，包括硬件和软件环境的成熟，势必需要研究新的内存管理技术。这些内存管理技术可能针对于新的问题，也可能是对原有问题解决方案的改善。

参考文献

[1] Kopetz H. Event-triggered versus time-triggered real-time systems. In: Proceedings of Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, London, UK: Springer-Verlag, 1991. 87–101

[2] Crespo A, Ripoll I, and Masmano M. Dynamic memory management for embedded real-time systems. In: Proceedings of IFIP International Federation for Information Processing, From Model-Driven Design to Resource Management for Distributed Embedded Systems. Springer Boston, 2006. 195-204

[3] Masmano M, Ripoll I, Crespo A, et al. Tlsf: A new dynamic memory allocator for real-time systems. In: Proceedings of ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems, Washington, DC, USA: IEEE Computer Society, 2004. 79–86

[4] Masmano M, Ripoll I, and Crespo A. A comparison of memory allocators for real-time applications. In: Proceedings of JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, New York, NY, USA: ACM, 2006. 68–76

[5] Xie Y and Aiken A. Context- and path-sensitive memory leak detection. SIGSOFT Softw. Eng. Notes, 2005, 30(5):115–125

[6] Feng Q, Shan L, and Yuanyuan Z. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In: Proceedings of Proceedings - International Symposium on High-Performance Computer Architecture, Piscataway, NJ 08855-1331, United States: Institute of Electrical and Electronics Engineers Computer Society, 2005. IEEE Computer Society, TCCA, Intel, AMD, IBM. 291-302

[7] Maebe J, Ronsse M, and Bosschere K D. Precise detection of memory leaks. Proceedings of the Second International Workshop on Dynamic Analysis, 2004, pages 25–31

[8] Ganai M K, Gupta A, and Ashar P. Verification of embedded memory systems using efficient memory modeling. In: Proceedings of DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA: IEEE Computer Society, 2005. 1096–1101

[9] Huang R F, Li J F, Yeh J C, et al. A simulator for evaluating redundancy analysis algorithms of repairable embedded memories. In: Proceedings of MTDT '02: Proceedings of the The 2002 IEEE International Workshop on Memory Technology, Design and Testing, Washington, DC, USA: IEEE Computer Society, 2002. 68

[10] Miller F W. Simple memory protection for embedded operating system kernels. In: Proceedings of Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2002. 299–308