嵌入式实时系统动态内存分配管理器的设计与实现

重庆大学硕士学位论文

（学术学位）

学生姓名：吴文峰

指导教师：李　斌　副教授

专　　业：控制科学与工程

学科门类：工　学

重庆大学自动化学院

二〇一三年四月

Design and Realization of A Dynamic Memory Manager for Embedded Real-time Systems

A Thesis Submitted to Chongqing University

in Partial Fulfillment of the Requirement for the

Master's Degree of Engineering

By

Wu Wenfeng

Supervised by Associate Prof. Li Bin

Specialty: Control Science and Engineering

College of Automation of Chongqing

University, Chongqing, China

April 2013

摘　要

随着物联网技术的发展，嵌入式实时操作系统得到越来越广泛的应用。嵌入式实时操作系统对动态内存分配的实时性、碎片率、可靠性有更高的要求，因此，动态内存分配已成为嵌入式实时操作系统的一个重要研究内容。现有内存管理一般通过牺牲内存利用率来满足实时性要求，造成了内存资源浪费。此外，由于嵌入式系统处理器一般没有内存管理单元，实时操作系统的内存管理无法对内存进行保护，易造成内存越界访问，降低系统的可靠性。为此，改善内存利用率和内存安全保护功能对于提供动态内存分配性能具有重要意义。

本文针对上述问题，在嵌入式实时系统内存管理算法 TLSF(Two-level Segregated Fit)基础上，针对其内存分配和释放时的不足，以及在内存保护方面的缺陷，研究了在不降低系统实时性的前提下降低内存碎片率的方法，通过加入二级位图算法，提出新的数据结构，添加内存越界访问保护机制，完成了一种新的嵌入式实时系统动态内存管理器 ERMM(Embedded Real-time Memory Manager)的设计。主要工作包括：

1）分析了常用嵌入式实时操作系统的动态内存管理算法，对不同的动态内存分配算法(DSA)的实时性及内存碎片率进行了讨论，根据嵌入式实时系统的对实时性及内存碎片率的要求，提出了以 TLSF 算法为基础的嵌入式实时系统动态内存管理器内存管理方案。

2）通过对 TLSF 算法进行详细分析，提出了一个新的嵌入式实时系统动态内存分配管理器(ERMM)。针对 TLSF 算法在小内存分配时效率较低的缺点，提出新的数据结构，对不同大小的内存请求采取不同的分配策略。对小内存请求，采取二级位图算法加快查找速度，并通过以空间换时间的策略降低分配时间，减少外部内存碎片；对大内存请求，采用改进 TLSF 算法：对其内存分配的“取下限”切割策略，提出“精确切割”的改进策略，降低内部内存碎片率；对其内存释放时空闲内存块“立即合并”策略，提出“延时合并”的改进策略，提高实时性；在可靠性方面，采取内存越界保护等措施提高内存管理的安全性。

3）将 TLSF 算法及 ERMM 在 μC/OS-Ⅲ系统上进行实现，在内存碎片率及实时性方面对这两种方法进行实验对比，并对 ERMM 的内存安全性进行测试。实验结果表明，ERMM 动态内存分配管理器具有较好的实时性、较高的可靠性及较低的内存碎片率，可有效提高嵌入式实时系统内存管理的性能。

关键词：动态内存分配，　TLSF 算法，　内存管理器，　嵌入式实时系统

ABSTRACT

With the development of Internet of Things technology, embedded real-time operating systems are more widely used. Embedded real-time operating system for real-time dynamic memory allocation, fragmentation, reliability have higher requirements, therefore, the dynamic memory allocation has become embedded real-time operating system is an important research content. Existing memory management by sacrificing memory utilization to meet real-time requirements, resulting in a waste of memory resources. In addition, embedded system processor generally do not have a memory management unit, memory management, real-time operating system can not protect the memory, could easily lead to cross-border access memory, reduce the reliability of the system. To this end, improve memory utilization and memory security features for dynamic memory allocation performance is important.

To address the problem on the basis of embedded real-time system memory management algorithms TLSF (Two-level Segregated Fit) for less than its memory allocation and release, as well as defects in the memory protection, research without reducing the system real-time under the premise of reducing memory fragmentation rate, by adding two bitmap algorithm proposed new data structure, to add memory cross-border access to protection mechanisms, completed a new embedded real-time systems dynamic memory management ERMM (Embedded Real- time Memory Manager) design. The main work includes:

1) analysis of the dynamic memory management algorithm commonly used in embedded real-time operating system, real-time and memory fragmentation rate of different dynamic memory allocation algorithm (DSA) are discussed, based on the embedded real-time systems for real-time and memory fragmentation rate requirements, the proposed algorithm TLSF-based embedded real-time systems dynamic memory management, memory management program.

2) conduct a detailed analysis the TLSF algorithm, proposed a new embedded real-time systems dynamic memory allocation manager (Ermm). For TLSF algorithm shortcomings of low efficiency in small memory allocation proposed new data structure, the different sizes of memory requests to take a different allocation strategies. Small memory requests, to accelerate the search speed take two bitmaps algorithm to reduce the allocation of time and space for time strategy to reduce external memory

fragmentation; large memory request, improved TLSF algorithm: take the lower limit of its memory allocation " cutting strategy proposed precision-cut improvement strategies to reduce the rate of internal memory fragmentation; release its memory free memory block Now Merge strategy proposed delay Merge improvement strategies to improve real-time; reliability , to take the memory cross-border conservation measures to improve the security of memory management.

3) TLSF algorithm and ERMM of OS-III system implementation in the memory fragmentation rate and real-time experimental comparison of these two approaches, and the memory security ERMM test. Experimental results show that the the Ermm dynamic memory allocation manager has better real-time, high reliability, and low memory fragmentation, which can effectively improve the performance of the memory management of embedded real-time systems.

Keywords: Dynamic Memory Allocation, TLSF Algorithm, Memory Manager, Real-time System

目　　录

1 绪　论

1.1 研究背景

1.1.1 嵌入式实时操作系统内存管理

嵌入式系统以低功耗、高可靠性、功能强大、性价比高、实时性强、支持多任务、网络功能丰富、占用空间小、效率高、软硬件选择灵活及相关接口可定制等特点广泛应用于工业控制领域[1]。作为嵌入式系统的一部分，嵌入式实时操作系统的正常运行至关重要。资源受限条件下系统的正常运行对操作系统设计提出了更高的要求，一方面要满足应用的实时性约束；另一方面，又要提高系统的灵活性与资源利用率，增强抵御未知风险的能力[2]。

嵌入式实时操作系统(Real-Time Embedded Operating System)能在确定的时间内执行相应操作，并对外界的异步事件做出响应，它是嵌入式系统的重要组成部分。不同于一般操作系统，它不仅对计算结果的准确有要求，而且对产生正确结果的时间也有要求。嵌入式实时操作系统最重要和最本质的特征是实时性和可预测性。算法要控制最坏情况下运行时间的有界性，以确保系统能够及时响应，这在嵌入式系统中更加重要。实时性是指系统要保障实时任务的时间约束，控制最坏情况下运行时间的有界性，以确保系统能够及时响应；可预测性是指系统要保证实时任务的执行时间是可预测的，处理一个重要实时任务的等待时间必须有个上限，要保证实时系统保证在规定的时限内完成实时任务。

嵌入式实时系统在功能上能够支持多任务模型下的任务管理、任务间同步通信、中断时钟管理、内存管理等。它具备较高实时性、较好的响应能力以及较强的可预测性；并且，它又具有体积小、相关接口可定制、软硬件选择灵活等特点[3]。实时操作系统可分为软实时系统和硬实时系统，硬实时系统要求在规定的时间内必须完成某些操作；而软实时系统则只要根据系统任务的优先级，尽快地完成相应操作即可。

嵌入式实时系统中，内存管理具有重要的意义，它是操作系统的一个重要模块，包括内存的分配和释放。内存管理的好坏直接关系着嵌入式系统的可靠性与稳定性。

1.1.2 动态内存分配概述

从时间分配上，内存分配的分配策略可分为静态内存分配和动态内存分配。静态内存分配的内存空间在编译或链接时就已根据实际需要分配好，静态分配是在栈(stack)上进行的，采用这种内存分配方式，分配内存的大小一般在编译时就能够确定；动态内存分配是指系统在运行时根据实际需要动态地进行内存分配[4]，也

叫基于堆的内存分配，它根据应用程序的内存需求提供内存空间，程序员可以根据具体问题的具体需要来分配合适大小的内存[5]。

静态内存分配是在代码编译或链接阶段进行的，分配速度快，但一旦分配内存大小就确定了，也无法更改，并且分配的内存会一直被占用，而无法释放，这会造成内存空间的浪费。而且，在实际应用中，内存空间分配的不够大，可能会引起内存越界访问，产生不可预料的后果。

动态内存又被称为 DSA(Dynamic Storage Algorithm)，它是在代码执行的过程中动态地分配或回收内存，代码需求空间是在运行中由实际需求动态决定，开发者能够根据实际需求获取内存分配，这使得开发者能更灵活地对内存进行使用和开发，提高了灵活性。在一些实际应用中动态内存分配是必须的[6]。比如，嵌入式系统中的网络协议栈，网络数据的大小不是固定的，这就要求系统要支持动态内存分配。若采用静态内存分配，协议栈在不同嵌入式平台间的移植就会变得困难，而且使得系统功能的灵活调整变得困难[7]。

下面从从实时性、空间效率、稳定性、灵活性等四个方面对静态内存分配和动态内存分配进行比较。

①实时性。静态内存分配由于申请简单，直接使用其地址即可，而且不需要释放，时间效率较高，而动态内存分配的时间效率依赖于具体的动态内存管理算法。

②空间效率。静态内存分配的内存大小往往是最坏情况的大小，这往往会造成空间的浪费，而且这些被分配的内存不能被重复利用。动态内存分配的内存大小由应用程序在运行时动态决定的，故其空间效率相对较高。

③稳定性。静态内存分配的稳定性较高，由于在编译、链接过之后，内存大小和地址已经确定，不会出现申请失败的情况。动态内存分配的稳定性相对较低，它依赖于动态内存管理算法的内部和外部碎片率。在一些情况下，可能系统总的空闲内存大小超过需要申请的内存大小，但由于内存碎片的大量存在，导致无法申请到足够大的连续内存，从而导致内存分配失败。

④灵活性。静态内存在编译链接之后就确定了具体的内存分配，复用性差，不够灵活。而动态内存分配，内存使用完并释放给系统之后，可以再次被其他应用程序利用，复用性较高，更为灵活。

动态内存分配中存储空间是按需分配，并且被释放的内存块可以再次参与分配，因此动态内存分配内存资源利用率比静态内存分配高。静态内存分配是在栈上进行的，在栈上进行大量静态分配是很危险的，若栈溢出时，多出的数据会覆盖掉相邻堆栈的数据，甚至会破坏内核造成系统崩溃。动态分配可避免这一状况，大块内存的分配必须采取动态内存分配。因此，为了在实时嵌入式系统开发中更

好地使用动态内存分配策略，越来越多的研究人员开始关注动态内存分配的实时性及内存碎片问题。

1.1.3 国内外实时系统动态内存管理研究现状

频繁的动态内存分配操作会造成大量内存碎片的产生，影响系统性能，所以在不同的系统中，对于动态内存管理，开发了许多不同的内存管理算法，典型的实时系统动态内存管理算法有[8]：

①顺序查找算法：包括首次适应算法(First-Fit)、最好适应算法(Best-Fit)、下一次适应算法(Next-Fit)等。First-Fit 算法的主要思想是：从空闲内存块链表头部向后遍历链表，直到发现第一个合适的空闲内存块。Best-Fit 算法是从空闲内存块链表中找最小的满足要求的空闲内存块。Next-Fit 算法则把最后一次满足要求的内存块的位置记录下来，有新的分配请求时，从记录的位置向后搜索，直到找到合适的内存块。

②伙伴算法：算法对空闲内存块不断进行对半切分，直到切出来的内存块刚好满足分配需求。内存分配时首先要计算最小的满足需求的块是多大，如果在对应的链表中能找到这样的块，就直接分配，否则就需要搜索更大的块，将它分为 2 个相等的子块，不断地重复这个过程，直到产生的块是最小能满足要求。

③Half-Fit 算法[9]：算法维护了 $\log_2 N$(N 表示可分配内存大小)个链表，每个链表里面内存块大小范围为$[2^i,2^{i+1})$。内存分配时，通过索引值获取对应的链表，将链表里的第一个空闲内存块，根据实际大小分配一部分内存给用户，剩余的内存加入到对应的链表里[10]。

④位示图算法：本算法通过位图中的位表示对应内存块的使用情况。位图通常是一个数组类型的数据结构。它将所有内存块的空闲状态信息都保存在一个 32 位的内存中，降低了时间损耗，提高了空闲内存块的查找效率。它基本思想是使用一组标志位来跟踪内存的使用情况，内存区域被分割成若干个小块，每一个内存块是否已经分配的状态由一个标志位来反映[11]。通常情况下，用“0”表示一个空闲内存块；用“1”表示一个已分配的内存块。

⑤TLSF 算法：TLSF(Two-level Segregated Fit)算法即二级间隔表动态内存分配算法。算法通过二级索引将空闲内存块组织起来。第 1 级索引按 2 的幂次将内存分成多个区域；第 2 级索引在第 1 级索引划分的内存区域里再次进行线性分割。第 2 级索引的每一位都对应一条空闲内存块链表[12]。

顺序查找算法的时间复杂度是 O(N)，实时性不好，空闲内存块通过链表进行链接，内存的分配时间与链表的节点个数成正相关，分配时间随着链表节点数目的增多而增加。伙伴算法的实时性很好，但内存分配时，对空闲内存块不断进行对半切分导致内存碎片率也很高，严重的时候内存碎片率可能达到 50%。位示图

算法是位运算，速度快，时间是常数级的，但只适合小内存的分配请求。Half-fit 算法在空闲内存块的查找中使用了位图算法，它的时间复杂度为 O(1)，具有较好的实时性。TLSF(Two-level Segregated Fit)算法的时间复杂度也为 O(1), 相比 Half-fit 算法，TLSF 算法具有更少的内存碎片率[13]。TLSF 算法通过二级位图能够快速索引到大小合适的空闲内存块，算法时间复杂度为 O(1)，但其二级索引每次查表、清除、设置表会耗费不少时间。TLSF 算法内存分配时的“取下限”内存分割策略及内存释放时的“立即合并”策略增加了内存碎片率，降低了分配效率[2]。另外，它对于过小的对象和过大的对象的处理不是很理想，会产生较多的碎片[5]。

1.1.4 实时操作系统中动态内存分配方案

常用的实时操作系统有 VxWorks、μC/OS-Ⅲ、RT_Thread、eCos(embedded configurable operating system)等操作系统，不同的操作系统采用不同的动态内存分配方案。

VxWorks 是美国 Wind River 公司推出的可在嵌入式系统上运行的高性能、高可靠性、可裁减的嵌入式实时操作系统，它适用于目前所有流行 CPU 平台[14]。它在嵌入式领域中有着广泛的应用，具有很高的市场占有率。VxWorks 具有抢占式调度、中断延迟小、系统内核可剪裁等特点。VxWorks 的内存管理以一个双向链表管理系统空闲块，采用 First-Fit 算法分配内存，释放的内存被合并成更大的空闲块。空闲内存块按地址大小递增排列，若动态内存分配请求的内存大小为 SIZE，则从链表头开始查找合适大小的空闲内存块，直到找到大于等于 SIZE 的空闲内存块为止。返回该空闲块的地址，完成动态内存分配[15]。

μC/OS-Ⅲ系统是在 μC/OS-Ⅱ系统上进行的改进版本，是一种基于优先级的抢占式多任务实时操作系统，它具有任务管理、时间管理、实时内核、任务间通信同步（信号量，邮箱，消息，队列）和内存管理等功能。μC/OS-Ⅲ内存管理的方式是把系统的内存空间分成若干内存分区，每个分区又被划分成整数个类型一致大小相同的内存块，不同分区的内存块大小是不同的。系统根据不同的需求，从大小不同的内存分区中分配最合适大小的内存块。内存使用完毕后，再把内存块释放回原来的内存分区。

RT_Thread 是一款由国内 RT-Thread 工作室开发的开源实时操作系统，它具有文件系统，网络协议栈，图形界面组件等。RT-Thread 操作系统有两种动态内存管理方法—小堆内存管理和 SLAB 内存管理，系统根据内存请求大小的不同采取相应的管理方法。小堆内存管理是将堆内空闲内存块用链表链接在一起，若有内存分配请求，则从这些链表中找出合适的内存块。SLAB 内存管理方法最主要应用于大内存的分配管理。

eCos 系统是由 Red Hat 公司开发的可裁剪的小型实时操作系统

RTOS(Real-Time operating system)。它的最低编译核心可小至 10K 的级别，适合用于微小型系统。eCos 系统根据不同内存需求采用不同的动态内存管理方法：一种以固定大小的内存块为单位进行内存分配，通过位图算法对内存块进行管理；另一种是根据内存请求的尺寸大小采用 Best-Fit 算法进行内存分配，它使用链表对内存块进行管理[16]。

1.1.5 实时系统动态内存分配方案存在的问题

嵌入式系统内存资源相对紧张，因此嵌入式实时操作系统的内存管理主要研究方向为提高内存分配的实时性，并同时降低内存碎片率。现在的实时操作系统在保证算法实时性的基础上采用了不同的内存切割与内存合并技术来保证减少内存碎片率，内存资源的利用率。μC/OS-Ⅲ通过分配和释放固定大小的内存块保证了动态内存分配的实时性，但是这会导致内部碎片率较高。VxWorks 内存管理采用 First-Fit 算法进行内存管理，实时性不是很好。RT_Thread 采用了小堆内存分配方法和 SLAB 内存分配方法，小堆内存是通过链表链接起来，实时性也不是很好。eCos 系统根据不同内存需求采用不同的内存管理方法，小内存分配固定大小的内存块，提高了内存分配的实时性，大内存分配是通过查找整个链表来实现的，实时性不高。

从实时性、内存碎片率这两方面考虑，TLSF(Two-level Segregated Fit)算法是最合适的动态内存分配算法[16]。为了提高实时性，TLSF 通过双级位图快速索引到大小合适空闲内存块，算法复杂度 O(1)。为了减少内存碎片，TLSF 算法在内存分配和释放时采用了切割与合并技术，当算法返回的空闲内存块大小大于所请求的内存大小时，将空闲内存块切割为两部分，一部分用于分配，另外一部分作为新的空闲块插入到空闲链表中；在内存释放时，判断相邻物理空间上的内存块是否空闲块，若都为空闲内存块，则两个空闲块合并成一个更大的空闲块。TLSF 算法的切割与合并策略降低了内存碎片率，但是内存分配时 TLSF 采用“取下限”内存分割策略使得切割后的内存块仍然存在内部内存碎片[2]，在内存释放时的采用“立即合并”策略增加了内存释放的时间，降低了内存的实时性，降低了分配效率。另外，TLSF 算法对于过小的对象和过大的对象的处理不是很理想，会产生较多的碎片[5]。最后 TLSF 算法对内存没有保护，容易导致内存访问越界的产生。

目前许多处理器都提供了内存管理单元(MMU)，它可将内存的逻辑地址转换为物理地址，它可以保护内存不会被非法访问[17]。但目前大多数嵌入式系统没有内存管理单元，用户程序、内核处于同一个地址空间，若内核的空间被非法访问，将会对整个系统造成不可估量的破坏。

1.2 课题的提出及研究意义

在远程监控系统中，嵌入式实时操作系统的动态内存分配得到了广泛的应用。由于需要传输的数据量大且数据大小不确定，动态内存分配在网络数据传输中起到了重要的作用。实时系统最重要的特点就是实时性，然而嵌入式系统内存资源有限，嵌入式实时系统的动态内存管理在满足系统实时性的前提下，往往增加了内存碎片，降低了内存利用率，造成了内存资源浪费。甚至由于内存碎片率过高，导致一些内存请求无法满足，使网络监控的数据传输无法完成。由于嵌入式系统处理器一般没有内存管理单元(MMU)，嵌入式实时系统无法使用虚拟内存管理对内存进行保护，若内存空间被错误修改，可能会导致任务无法运行，甚至会导致系统的崩溃，这降低了嵌入式实时系统内存管理的可靠性与安全性。

目前，嵌入式实时系统采取了一系列的动态内存算法在保证实时性的前提下降低内存碎片率，在这些算法中，TLSF(Two-level Segregated Fit)算法是比较优秀的算法，它在实时性及内存碎片率方面都有较好的表现，但 TLSF 算法在对于过小的对象和过大的对象的处理时会产生较多的碎片；在内存分配时 TLSF 算法的“取下限”内存分割策略及内存释放时的“立即合并”策略增加了内存碎片率，降低了分配效率；另外，TLSF 算法没有考虑内存保护，降低了算法的安全性。

嵌入式系统内存资源相对紧张，因此，在不降低系统实时性的前提下提高内存碎片率具有重要意义。本课题以某企业远程监控系统建设为背景，提出一种新的实时系统动态内存分配管理器，在 TLSF 算法的基础上，提出新的数据结构，增加二级位图算法以增强对小对象的处理，改进 TLSF 算法的内存分配和内存释放策略，增加内存保护算法，以期达到在满足系统实时性的同时，降低内存碎片率，提高内存的安全性，增加系统的可靠性，该内存管理器能满足嵌入式实时系统对内存管理的要求。

1.3 主要研究内容

本文以实时嵌入式系统在设备远程监控中的应用为背景，分析常用实时系统内存管理算法的思想及其优缺点，在 TLSF(Two-level Segregated Fit)算法的基础上，提出一种内存碎片率低、实时性好并适用于嵌入式实时系统的内存管理器 ERMM(Embedded Real-time Memory Manager)，主要在提高内存分配实时性、减少内存碎片、提高内存利用率及内存安全性等方面加以研究，分析其优越性和不足，并将这种算法在嵌入式实时系统 μC/OS-Ⅲ上实现。具体工作包括实时系统动态内存管理设计需求分析、实时系统动态内存管理分析、提出了在内存碎片率、实时性及安全性都较好的内存管理器 ERMM，并将 ERMM 在 μC/OS-Ⅲ系统上进行了实现，本文的主要研究内容如下：

1．实时系统动态内存管理分析及设计要求的提出

在阐述嵌入式系统特点及嵌入式实时系统的基础上，分析典型的动态内存管理算法及其在不同实时操作系统的动态内存分配方案，指出不同算法存在的问题，并针对实时性及碎片率，提出了在 TLSF 算法进行改进的动态内存管理器设计方案。

2．ERMM 内存管理器的设计

通过分析不同内存管理算法优缺点，选择目前实时性、内存碎片率方面比较优秀的 TLSF 算法，针对 TLSF 算法在内存分配和释放时存在的问题，在该算法的基础上进行改进，提出了一种新的动态内存分配管理器 ERMM(Embedded Real-time Memory Manager)。具体的改进包括改进数据结构，增加二级位图算法以增强对小对象的处理，对不同大小内存分配采用不同的分配策略以提高实时性，添加边界标示技术提高算法的安全性和可靠行 ;内存分配采用“精确切割”降低内存碎片率，内存释放采用“延时合并”策略提高实时性。

3．ERMM 在 μC/OS-Ⅲ系统上的实现

首先在 VC 环境下把 μC/OS-Ⅲ系统移植到 Intel 奔腾系列处理器上，并在 μC/OS-Ⅲ系统上通过修改 μC/OS-Ⅲ系统内存管理的相关文件实现 TLSF 算法，并在此基础上，修改 TLSF 算法的数据结构，重新实现内存分配及释放函数，在 μC/OS-Ⅲ系统上实现 ERMM 内存管理器。

4. ERMM 内存管理器实时性、内存碎片率、内存安全性检测实验

进行 ERMM 内存管理器实时性、内存碎片率及内存安全性检测实验，并和 TLSF 算法进行对比分析。

1.4 论文结构

论文共分为五章，各章节内容如下：

第一章，绪论。本章介绍了本文的研究背景，课题的提出及研究意义和主要研究内容；

第二章，嵌入式实时系统内存分配管理的研究。本章主要阐述嵌入式实时系统对内存管理的要求，分析不同嵌入式实时系统内存管理的方法，对典型的内存管理算法进行比较，指出这些内存管理算法存在的问题。

第三章，提出一种新的内存分配管理器 ERMM(Embedded Real-time Memory Manager)，并对它的实现方法及在 TLSF(Two-level Segregated Fit)算法上的改进之处加以详细阐述。本章首先对 TLSF 算法从重要参数、数据结构及内存分配和回收三个方面进行详细阐述。接着重点分析了 ERMM 在 TLSF 算法上的改进之处，以及新的数据结构在实时性及内存碎片率方面所作的改进，然后介绍了 ERMM 小内

存和大内存的分配和回收步骤，ERMM 的内存越界保护方法。最后，对 ERMM 动态内存分配管理器进行理论分析，从时间复杂度、内存碎片率及内存安全性三个方面说明 ERMM 能够提高实时性、降低内存碎片率、提高内存安全性。

第四章，将 ERMM 动态内存管理器及 TLSF 算法在 μC/OS-Ⅲ系统实现。本章首先介绍了 μC/OS-Ⅲ系统，阐述了 μC/OS-Ⅲ系统内存管理方案，将该系统在 VC 环境下移植到了 X86 平台下，并在该系统上实现了 TLSF 动态内存管理算法。最后，在此基础上，对 TLSF 算法加以改进，在 μC/OS-Ⅲ系统上实现了 ERMM 动态内存管理器。

第五章，在 μC/OS-Ⅲ系统上对 TLSF 算法及 ERMM 内存管理器进行测试，对这两种算法进行内存分配响应时间、释放响应时间、内碎片率及内存越界访问保护的实验，根据实验结果，对 ERMM 及 TLSF 算法进行比较。

第六章，总结与展望。本章主要对本文的工作内容进行总结，并针对其中的不足，对下一步工作进行展望。

1.5 本章小结

本章主要介绍了课题的研究背景，课题的提出及研究意义，本文的主要研究内容和文章结构。

2 嵌入式实时系统内存管理

2.1 引言

嵌入式系统的内存资源有限，嵌入式实时操作系统的内存管理方式不同于桌面操作系统。同时，嵌入式实时系统对实时性要求较高，嵌入式实时系统的内存管理也不同于非实时操作系统。

本文先简要介绍嵌入式系统及其内存管理，然后重点介绍了不同嵌入式实时系统的动态内存管理方案，最后介绍常见的内存管理算法。

2.2 嵌入式系统简介

目前，随着生产规模和业务量的不断扩大，企业对终端设备信息化管理方面的需求也逐渐增多，越来越多的嵌入式系统被广泛应用[18]。

对于嵌入式系统，目前还没有一个统一的定义。英国电器工程师协会(U.K. Institution of Electrical Engines)将嵌入式系统定义为控制、监视或辅助设备、机器或用于工厂运作的设备[19]。一般来说，嵌入式系统是一种专用的计算机系统，作为装置或设备的一部分，它主要完成信息交互、数据计算与信号控制等功能[20]，它具有体积小，结构紧凑，软硬件可定制，并有丰富的人机交互接口，具有较高的性价比和智能性。其软硬件体积小，结构紧凑，并且为用户提供人机交互接口，使设备及应用系统具有较高的智能和性价比。

嵌入式系统由 I/O 设备、中央处理器、存储器等基本单元组成，但和普通计算机系统不同的是，由于嵌入式系统应用于特定的应用环境，它还具有硬件高度集成、软件定制灵活、软件开发方便、硬件资源有限等特点[19]。

与通用计算机操作系统相比，嵌入式系统有较高的实时性，同时，嵌入式系统可以进行多任务处理，对重要任务可以实时响应。嵌入式操作系统的多任务处理能力，以及配合界面美观的用户图形系统，使其可实现复杂的应用功能，也使得嵌入式操作系统得到越来越广泛的应用。

2.3 嵌入式系统内存管理

桌面操作系统 Linux、Windows 等的内存管理方式和嵌入式系统不同，它们大多采用段页式虚拟内存管理，内存空间和程序被分成大小相同的页面，内存分配和程序运行只需要操作相应页面即可[21]。由于桌面系统的微处理器带有内存管理单元(MMU)，MMU 的作用是将逻辑地址映射为物理地址，防止内存越界访问，从而对内存进行保护。大多数嵌入式系统的微处理器没有内存管理单元，无法使用虚拟内存管理技术，只能采用实地址管理模式，直接访问实际的物理地址。嵌入

式实时操作系统因此也没有对内存进行保护，系统和应用程序共享一个运行空间，若应用程序发生内存越界访问，很可能对内核空间的数据造成破坏，导致系统发生不可预料的异常。任务之间也可能相互破坏数据或任务代码，使得任务工作异常，甚至崩溃[22]。

因此，内存管理是嵌入式实时系统很重要的一个方面它对嵌入式实时系统的稳定运行有着至关重要的意义。近年来对嵌入式系统安全及可靠性的要求也越来越高，但在嵌入式实时系统动态内存分配中，内存碎片，内存泄露都对嵌入式实时系统的安全可靠运行带来很大的危害。内存如何分配和释放，才能保证内存碎片少，保证实时系统的实时性好，这些都是内存管理所要考虑的问题。

实时系统对动态内存管理的需求如下：

1）有界的反应时间（Bounded response time）。实时系统的内存管理需要控制最坏情况下运行时间的有界性，以保证系统的响应时间，满足嵌入式实时系统的实时性。

2）快速的反应时间（Fast response time）。保证实时系统动态内存分配的快速性，使动态内存算法的执行时间尽量的短，提高嵌入式实时系统的快速反应性。

3）可靠性（Reliability）。动态内存管理要保证内存请求总是被满足并被高效使用。尤其是一些重要任务的内存请求必须得到满足，除非系统内存资源耗尽。动态内存算法应该尽可能减少耗尽内存池的可能，通过减少内存碎片，减少内存的浪费[23]。

内存的申请策略可以分为[24]：首次适应(First-fit)、最佳适应(Good-fit)、下一次适应(Next-fit)、最差适应(Worst-fit)、准确适应(Exact-fit)等。

内存的释放策略有：空闲内存块立刻合并、延迟合并以及不合并策略。

内存管理的数据结构有：链表（包括顺序与多重链表）、位图、索引结构及混合机制等[25]。

2.3.1 内存碎片

内存碎片是指内存空间没有被有效地利用，虽然有空闲内存块，但却无法用于满足任务的内存请求[26]。程序执行一段时间以后，随着内存块的使用和释放，许多正在使用或已经释放的存储块将混杂在一起，未被使用的内存分裂成许多小块内存，形成这些内存碎片，这些小的空闲内存块无法被使用，使实际可使用内存空间减少，造成内存资源浪费，降低了应用程序的性能。优秀的内存分配算法必须要将内存碎片维持在一定范围内，内存碎片率是衡量内存分配算法的一个重要的参数[27]。

内存碎片分为外部碎片和内部碎片。

内部碎片：内部碎片就是已经被分配给应用程序，但没有用到的那部分存储

空间；它是处于区域内部或页面内部的存储块。应用程序并不使用这些存储块，当应用程序占有这块存储块时，系统无法使用它。直到该内存块被释放，或进程结束时，该存储块有可能被重新利用。

外部碎片：外部碎片是还没有被分配出去，但由于太小了无法被分配的内存空间。及时这些内存空间的总和可以满足当前内存申请大小的要求，但是这些内存空间的地址不连续，从而导致这些内存空间无法被分配。这些外部内存碎片一般可以通过紧缩的方法进行合并，但紧缩消耗很多时间，嵌入式实时系统一般不对内存进行紧缩。嵌入式系统是将相邻物理地址的空闲内存块进行合并以降低内存碎片[28]。

大量的内存碎片将带来三个后果：

1）降低内存使用效率。虽然有大量的空闲内存，但分配一个并不是很大的连续内存空间却不可得。如果内存中最大的空闲内存块为 90 字节，即使空闲内存的总量超过 1000 字节，仍然无法满足 100 字节的内存分配请求。

2）如果使用任意长度分配法，大量碎片会使内存管理的开销增大，大量的内存碎片使空闲内存链表的结点数量增加，延长了搜索和管理链表所需的时间，分配和释放内存所花费的时间变得很长且不确定。

3）造成伪内存丢失，伪内存丢失指的是该内存虽然仍然当作空闲内存被操作系统登记在案，但却永远不可能被使用。假如动态申请的内存块最小尺寸为 100 字节，如果内存区中存在一个 90 字节的内存碎片，只要这个碎片的相邻内存块不释放与本块合并，那这 90 字节内存永远无法使用。

2.3.2 内存泄露

动态内存分配的另一个可能的后果就是内存泄漏，也叫内存丢失，内存丢失是一块已经分配但不再被任何人使用且永远不可能被释放的内存块，属于最严重的软件事故之一。如果一个内存块不再有指针指向它，这块内存就丢失了。内存丢失将直接使系统可用内存减少以致内存被逐步耗尽，对于要求高可靠性且长时间连续运行的嵌入式系统，程序员需要确保绝对没有内存丢失。内存丢失是软件使用 malloc 和 free 族函数不当的 bug 造成的，一个没有 bug 的程序是不会产生内存丢失的。程序员虽然难于控制内存碎片的产生，但可以消除内存泄漏，但是消除这些 bug 的难度非常大。避免内存丢失是每一个程序员都要特别注意的问题，经验丰富的程序员只要看到程序里出现 malloc（包括 calloc，realloc）时，就要象狼一般警觉，要避免内存丢失，就必须成对地使用 malloc 和 free 函数，内存使用完后及时释放。而且 malloc 和 free 要在同一层调用，尽量不要在函数里面调用 malloc，而在函数外面调用 free。更不要在这个线程调用 malloc，在另一个线程中调用 free，造成内存碎片的错误主要有两种，一是使用完毕后没有释放，另一种情

况 malloc 后是内存指针被改变。

内存碎片和内存丢失都可能会导致内存枯竭，而且问题是慢慢积累发生的，内存资源越丰富，积累的时间就越长。这个问题功能测试往往发现不了，而是在软件运行一定时间后才表现出来，排查这类问题是非常困难的。

有些操作系统会进行内存碎片整理，使小块内存聚合而成为一块完整的大块内存。但实时系统决不会这样做，内存是一种临界资源，整理内存时必需禁止多任务调度，此时，即使有高优先级的任务就绪，也没有办法运行，而且整理内存所花费的时间是不确定的，实时系统不可能花费很长且不确定的时间来整理内存。另一个原因是，整理内存就要移动内存，这样将迫使应用程序必需以间接指针来引用动态分配的内存，而且内存的读写都必需使用原子操作，这样势必大大降低程序的执行效率，这在实时系统中也是不允许的。

2.3.3 嵌入式实时系统内存管理要求

衡量嵌入式实时系统动态内存分配算法的优劣，主要从三个方面进行考察[29]：

1）实时性。算法要控制最坏情况下运行时间的有界性，以保证系统的响应时间，这在嵌入式实时系统中更加重要。

2）内存碎片率。内存碎片分为内部碎片和外部碎片，碎片会影响系统的性能，并导致内存浪费，降低内存的利用率。嵌入式实时系统动态内存分配算法要尽可能降低内存碎片率。

3）可靠性。一方面是嵌入式实时系统必须满足一些重要任务的内存分配请求，否则，可能会使系统产生不可预料的后果。另一方面是对内存的保护，防止内存越界产生。嵌入式实时系统应用环境复杂，对可靠性的要求很高。

2.4 不同嵌入式实时系统内存管理概述

常用的嵌入式实时系统有 μC/OS-Ⅲ、RT_Thread、VxWorks、RTEMS 等系统，为了保证嵌入式实时系统的实时性以及内存碎片率及内存利用率，这些嵌入式实时系统采用了不同的内存管理方法。下面对每个内存管理方法进行详细分析。

2.4.1 μC/OS-Ⅲ的内存管理

μC/OS-Ⅲ系统把连续的内存空间分成若干内存分区，每个分区又被划分成整数个内存块，同一内存分区里的内存块的类型一致、大小相同，不同分区划分后的内存块大小是不同的。系统根据应用程序不同大小的内存分配请求，首先查找合适大小的内存分区，再从该内存分区中查找空闲内存块。若该内存分区中存在空闲内存块，则将该空闲内存块整个分配出去，否则，继续查找含有更大内存块的内存分区，直到找到空闲内存块或搜索完所有内存分区为止。内存释放时，将内存块重新释放到原来的内存分区。通过这样的内存管理算法，解决了内存碎片

的问题，提高了系统性能[30]。

μC/OS-Ⅲ内存管理直接操作物理内存，查找速度快，提高了系统实时性。内存分配时，将整个内存块进行分配，不会产生外部碎片。

μC/OS-Ⅲ内存管理的缺点：

1) 空闲内存块尺寸是在编译时确定的，每个内存分区中，内存的起始地址、块个数是创建内存分区时设置的，创建后不能再修改。这造成了内存浪费，也限制了系统的灵活性。
2) 内存分区的同一分区的内存块大小是相同的，系统为了满足实际应用中不同大小的内存请求，需要建立多个内存分区，增大了维护难度[31]。
3) 动态内存分配失败后，μC/OS-Ⅲ没有提供后续处理措施，这降低了系统的可靠性与稳定性。
4) 当内存分区中没有合适的空间分配时，系统需要找一块比较大的内存块分配或重新建立一个内存分区，新建分区会降低系统的实时性，找一个比较大的内存块分配会引起内存碎片的增多。
5) μC/OS-Ⅲ的内存管理没有提供存取控制、审计跟踪等安全性措施无法满足越来越紧迫的安全性要求[32]。
6) μC/OS-Ⅲ的内存管理内存空间泄漏，也没有特别好的解决方法。

2.4.2 RT_Thread 的内存管理

RT-Thread 操作系统在内存管理上，根据上层应用及系统资源的不同，有针对性的提供了数种内存分配管理算法：静态分区内存管理及动态内存管理。动态内存管理又根据可用内存多少划分为两种情况，一种是针对小内存块的分配管理，一种是针对大内存块的分配管理。小内存块的分配管理的分配管理是用小堆内存管理算法，大内存块的分配管理是用 SLAB 内存管理算法。小堆内存管理算法主要针对系统资源比较少的系统；而 SLAB 内存管理模块则主要是在系统资源比较丰富时，提供了一种近似的内存池管理算法 RT-Thread 操作系统根据不同内存需求提供了两种动态内存管理方法：小堆内存管理方法和 SLAB 内存管理方法；两种方法在实际系统中只能采用其中一种，其中小堆内存管理主要用于系统资源比较少一般小于 2M 内存空间的系统，而 SLAB 内存管理方法则用于系统资源比较丰富的系统中[33]。两种内存管理模块在系统运行时只能选择其中之一（或者完全不使用动态堆内存管理器），两种动态内存管理模块 API 形式完全相同。

小内存管理算法将空闲内存块通过链表连接在一起，有内存分配请求时，算法会顺序搜索链表，查找到有可用内存的时候，会从中分割出一块作为分配的内存，剩余的再添加到链表中。内存释放则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

SLAB 内存管理算法是 Matthew Dillon 在 DragonFly BSD 中实现的 SLAB 分配器基础上针对嵌入式系统优化过的内存分配算法。算法会根据对象的大小分成多个内存区，每个内存区的大小在 32k 到 128k 字节之间。内存区的大小会由算法在堆初始化时根据堆的大小自动调整。每个内存区中的内存块大小是固定的，分配相同大小内存块的内存区会通过链表链接在一起，不同的内存区链表通过一个数组统一管理。

2.4.3 VxWorks 的内存管理

VxWorks 系统的内存管理采用简单的“首次适应，立即聚合”方法，它采用用户程序、内核处于同一个地址空间的内存管理策略，通过链表管理内存块。系统中只有 1 个系统内存分区，内存分配时，调用 malloc 函数从系统分区中进行分配。空闲内存块按地址大小递增排列，分配时采用 First-Fit 算法，若应用程序内存请求大小为 SIZE，算法从空闲内存块链表头节点开始查找，直至找到内存块大小大于或等于 SIZE 的空闲内存块为止，返回该空闲内存块的指针，完成内存分配。内存释放时，调用 free 函数，首先判断空闲内存块相邻物理地址是否存在空闲内存块，若存在，则将空闲内存块合并，修改内存块块头的长度信息及空闲状态信息，然后将该内存块插入到链表中[34]。

作为一个嵌入式操作系统，VxWorks 的内存管理既不分段也不分页[34]，通常，VxWorks 内核和应用程序对内存的操作和管理是基于内存分区的。内存分区包含分区自身的描述信息和一个或多个内存池，描述信息是一个结构体，内存池是一块连续的内存区域，内存池被分成多个内存块[35]。

2.4.4 RTEMS 的内存管理

RTEMS(Real Time Executive Multiprocessor Systems)是微内核抢占式实时嵌入式操作系统，它是个开源的实时嵌入式操作系统，具有实时性高、稳定性强、良好的可移植性和可裁剪性等优点，最早用于美国的国防系统，早期的名称为实时导弹系统，后来改名为实时军用系统。目前无论是在军工还是在民用领域都有着广泛的应用[36]。RTEMS 系统的动态内存管理算法是首次适应算法。

2.4.5 eCos 的内存管理

eCos 是 Red hat 公司开发的一个可裁剪的嵌入式实时操作系统。eCos 系统支持大部分主流的 CPU，目前已经成功移植到主流的 CPU 的体系结构上。eCos 系统的内存管理可以根据实际需求选择合适的分配算法[7]。

eCos 系统有两种内存管理方式，都是通过内存池来实现的。一种方式是使用固定块大小的内存池，内存池被分为大小相同的内存块，使用位图标识内存块的状态，分配固定大小的内存块。另一种方式是用链表将不同大小的内存块链接在一起，根据实际的内存请求大小分配合适大小的内存块[37]。

它提供两种动态内存分配算法：最好适合算法(Best-Fit)和 DougLea 算法。

最好适应算法的实时性不好, 时间花费随着链表节点的增多而增大, 但 Best-Fit 算法的优点就是会产生较少的外部碎片，具有良好的空间性能；

DougLea 算法是比较优秀的内存管理算法[38]。它将所有的内存块放在固定大小的 128 个容量仓里，不同大小容量仓的内存块存储方式不同，小于 512B 的容量仓里内存块大小从 16B 开始，以公差为 8B 递增，最高大小为 512B；大于 512B 的容量仓，内存块按大小递增排列[39]。

内存分配时，根据不同大小的内存请求采取不同的内存分配策略，DougLea 算法先判断容量仓中是否有合适的空闲内存块，若没有，则检测最近使用的内存块是否有足够的剩余空间。算法对大块内存（512B）的分配, 采用最佳适应算法[39]。

2.5 典型的内存管理算法

典型的内存管理算法有顺序查找算法、伙伴算法、Half-fit 算法、TLSF 算法等，下面对这些算法进行一下简单的介绍。

2.5.1 顺序查找算法

顺序查找算法，包括最先适合(First-Fit)、最好适合(Best-Fit)、下一次适合(Next-Fit)等算法，这些算法都是通过链表将空闲内存块连接起来。First-Fit 算法从块链表头部开始遍历整个链表，直到发现首个满足要求的空闲内存块。Best-Fit 算法从块链表中找最小的满足需求的空闲内存块。Next-Fit 算法会记录最近一次满足要求的空闲内存块的位置，处理下一个分配请求时，直接从该位置向后搜索，从而让每个块都有响应机会[7]。

最先适合算法实现比较简单，从空闲内存块链表头部开始遍历整个链表，直到发现首个满足要求的空闲内存块，若找到的内存块比请求的大，就分割这个内存块，并将剩余的插入到空闲链表中。经过一段时间后，链表前面的内存块会越来越小，从而导致每次搜索链表的范围越来越大，内存分配时间也越来越长且无法预测，大内存块要比小内存块分配花费时间更长，内存分配的搜索时间和已分配的内存块个数成 O(N)的关系，内存分配实时性得不到保证[40]。另外，内存块的不断地拆分使内存碎片率越来越高。

最好适合算法内存分配需要搜索整个链表，查找最合适的空闲内存块。搜索链表花费的时间和链表长度成正相关，内存分配效率较低。同时，为了解决它的碎片问题，那就是每次提交给请求者的内存都会大于等于它的请求值.不过这样会导致内存浪费。

2.5.2 伙伴算法

伙伴算法(Buddy)用位图和链表管理空闲内存块，算法按照 2 的幂次方对内存

进行内存分配和释放，所有的空闲内存块通过多条链表组织起来，每个链表中分别包含 2^0 到 2^9 个页面大小的内存块若干，每个块的起始物理地址是该块大小的整数倍[41]，内存分配和释放速度较快。内存分配时，伙伴系统利用位图搜索空闲内存块。若没有搜索到大小合适的空闲内存块，则将一个大内存块分成相等的两块，若分割后的内存块依旧太大，则继续等分，直到分割后的内存块大小最小满足要求[42]。。内存释放时，相邻物理地址的空闲内存块会进行合并，以得到更大的空闲块，并减少了内存碎片。为了便于内存合并，内存块通常按照地址顺序存放[43]。内存释放时，相比按内存大小分配的算法，伙伴算法只需要搜索互为伙伴的空闲内存块，然后进行合并，具有更快地合并速度。

伙伴算法的内存利用率较低，它在内存分配时，会按 2 的幂次方对空闲内存进行分割，在极端的情况下，内存资源会被浪费近一半。这种内存管理方式造成了内存空间的极大浪费，甚至会因此导致后续的内存请求无法得到满足，对应用程序的正常运行造成严重影响。

2.5.3 Half-Fit 算法

该算法维护了 $\log_2 N$ 个链表(N 表示可分配内存大小)，每个链表里的内存块大小范围在 2^i 到 2^i+1，通过间隔表的方式组织空闲内存块[43]。对每个空闲内存块，若其大小为 size，则总能找到整数 i，使得 size 在$[2^i,2^{i+1}-1]$,同一个 i 所对应的空闲块以链表的方式链接在一起，这些链表再以递增的方式以链表的形式组织起来。有动态内存分配请求时，首先找到对应的链表，确定空闲内存块对应的链表索引值 i，返回索引值为 i+1 的空闲内存块对应链表的第一个空闲块的地址。若请求内存块的大小小于空闲内存块，并满足内存的切割条件，则对该空闲内存块进行切割，剩余的空闲内存块返回给链表。内存释放时，查看该空闲内存块物理地址相邻的内存块，若该内存块是空闲的，则采用一定的算法与该内存块进行合并。算法用位图来确定索引值 i，最坏执行时间是确定的，算法具有较好的实时性。

2.5.4 TLSF 算法

TLSF(Two-level Segregated Fit)是一种二级隔离适应算法，使用位图(Bitmaps)与分组空闲链表(Segregated List)相结合的方式对内存进行管理。TLSF 采用最好适合(Good-fit)算法，它通过使用位图与分组空闲链表相结合的方式达到最佳匹配的效果，这样减少了内存碎片。位图的使用增加了算法的查找速度，提高了算法的实时性。

TLSF 实现过程如下：定义一级索引最大值 MAX_FLI(小于 31)，与二级索引最大值 MAX_SLI(一般为 8)，MAX_SLI 的值是 2 的幂次方。通过定义全局变量的方式申请内存区，内存区的创建使用初始化函数 init_memory_pool()。动态内存申请时使用 tlsf_malloc()函数，动态内存释放时使用 tlsf_free()函数，若重新申请则使用

realloc()函数[44]。

2.6 典型内存算法存在的问题

顺序查找算法这种算法的实现较简单，现在大多数的实时操作系统也是采用了顺序查找算法中的最先适合(First-Fit)算法，但算法对内存块的查找都是通过搜索链表来实现的，内存块的查找时间无法保证，从而导致算法的实时性不好。算法随着内存碎片的增多，空闲块链表的长度逐渐增大，申请和释放所用的时间也将增大[45]。

伙伴算法是一种比较常见的算法，在 Linux 系统中得到广泛的应用，该算法按照 2 的幂次方对内存进行分配，综合利用了位图和链表的方式，具有很高的分配速度和回收速度，但是由于内存分配时，对内存块的分割是按照 2 的幂次来进行的，导致内存利用率不高，产生较多的内存碎片，嵌入式内存资源相对比较紧张，伙伴算法不适合嵌入式实时系统。

Half-Fit 算法是基于离散表匹配的分配算法，使用位图(Bitmaps)与分组空闲链表(Segregated List)相结合的方式对内存进行管理，算法通过位图来查找空闲内存块，使得查找速度很快，时间复杂度 O(l)，并且，使用位图来判断内存块的状态，需要的额外开销很小。算法用位图来确定索引值 i，最坏执行时间是确定的，算法具有较好的实时性；通过间隔表的方式组织空闲内存块，内存碎片率较低。

TLSF(Two-level Segregated Fit)算法使用位图与分组空闲链表相结合的方式对内存进行管理，使用位图提高了算法的执行速度，TLSF 在小内存分配时采用了 Good-Fit 算法，使得分配内存大小和请求内存大小尽可能达到匹配，虽然一定程度上减少了碎片率，但是这也降低了内存分配的实时性。TLSF 通过双级位图能够快速索引到大小合适空闲内存块，算法复杂度 O(1)，但其二级索引每次查表、清除、设置表耗费不少时间。TLSF 内存分配时的“取下限”内存分割策略及内存释放时的“立即合并”策略增加了内存碎片率，降低了分配效率。另外，它对于过小的对象和过大的对象的处理不是很理想，会产生较多的碎片。

2.7 本章小结

本章首先介绍了嵌入式系统的定义、特点和基本结构，详细阐述了内存碎片及内存泄露，并分析其危害，提出了嵌入式实时系统对内存管理的要求；然后分析了不同嵌入式实时系统的动态内存管理方案；最后介绍常见的内存管理算法，比较它们在实时性、内存碎片率方面的优劣，并指出这些内存算法存在的问题。

3 ERMM 内存分配管理器设计

3.1 引言

TLSF(Two-level Segregated Fit)算法在内存碎片率及实时性方面都有较好的表现，但它在安全性及内存分配和释放时都存在一些不足。本文针对 TLSF 算法的不足，提出一种新的动态内存分配管理器 ERMM(Embedded Real-time Memory Manager)。它将内存请求分为小内存请求和大内存请求；针对小内存请求，采用二级位图算法，通过以空间换时间的策略降低分配时间，减少外部内存碎片；针对大内存请求，在 TLSF 算法的基础上改变二级索引结构，通过“精确切割”及“合并阈值”策略提高内存利用率，降低内存碎片。

本章先对 TLSF 算法进行介绍，分析算法在实时性、内存碎片率及安全性方面的不足，详细叙述了动态内存管理器 ERMM 数据结构、算法改进方案及内存分配和回收具体方案和内存保护措施。最后对 ERMM 的时间复杂度、内存碎片率进行理论分析。

3.2 TLSF 算法介绍

TLSF(Two-level Segregated Fit)是一种二级隔离适应算法，使用位图(Bitmaps)与分组空闲链表(Segregated List)相结合的方式对内存池进行管理。从文献[46]中可以看出 TLSF 算法比 First-Fit、Half-Fit、伙伴算法在实时性及内存碎片率方面要好。因此本文在 TLSF 算法基础上对其进行改进，以提高算法的实时性及内存碎片率。

3.2.1 重要参数

TLSF(Two-level Segregated Fit) 结构特性主要依靠三个参数：

1）一级索引（FLI）；该索引标识一级链表长度，内存区共被划分成 REAL_FLI(一级索引个数)个大的内存块区间(一级内存块)，一级索引值为 fl 的内存区间区间为$[2^{fl},2^{fl+1})$。参数 FLI_OFFSET 定义了最小一级索引的内存大小[0 $2^{FLI\_OFFSET+1}$)，低于此内存的所用内存块在此区间分配。

2）二级索引（SLI）：二级索引将一级索引划分的内存区域按照线性再次划分。SLI 的值可人为设定，若 SLI=3，则，一级索引划分的内存区被划分成 8 段。SLI 的值越大，则将内存区划分得越细，但一般而言，SLI 不超过 5。若 SLI 的值超过 5，则 2 的 SLI 次幂超过 32，无法用位图表示。

3）最小块大小（MBS）：这个参数是定义最小块的大小。最小块的大小一般定义为 16Bytes。块大小也可根据用户的实际需求而更改。

3.2.2 数据结构

TLSF 用位图与分组空闲链表相结合的方式对内存池进行管理，空闲内存块用二级索引进行组织。TLSF 定义两个索引值：一级索引个数（REAL_FLI）及二级索引个数（MAX_SLI），一级索引将空闲内存区进行分割成个 REAL_FLI 个内存区间，索引值 i 对应的内存区间大小为$[2^i,2^{i+1})$，二级索引在此基础上，将一级索引划分的内存区间$[2^i,2^{i+1})$再线性分割成 MAX_SLI 个区间。二级索引的每个类别，都对应一条属于该类别尺寸范围内的空闲内存块链表，用一个二维数组存储所有的空闲内存块链表的头指针。二级索引划分的内存区间尺寸范围可参见图 3.1。第一级索引、第二级索引都使用位图指示该类别内存块的空闲状况，若有空闲内存块，则对应的位为 1，否则为 0。

当 n=27，MAX_SLI(二级索引数目)=8 时，TLSF 数据结构如图 3.1 所示。其中，fl 为一级索引值，sl 为二级索引值。FL_Bitmap 和 SL_Bitmap 描述索引表。

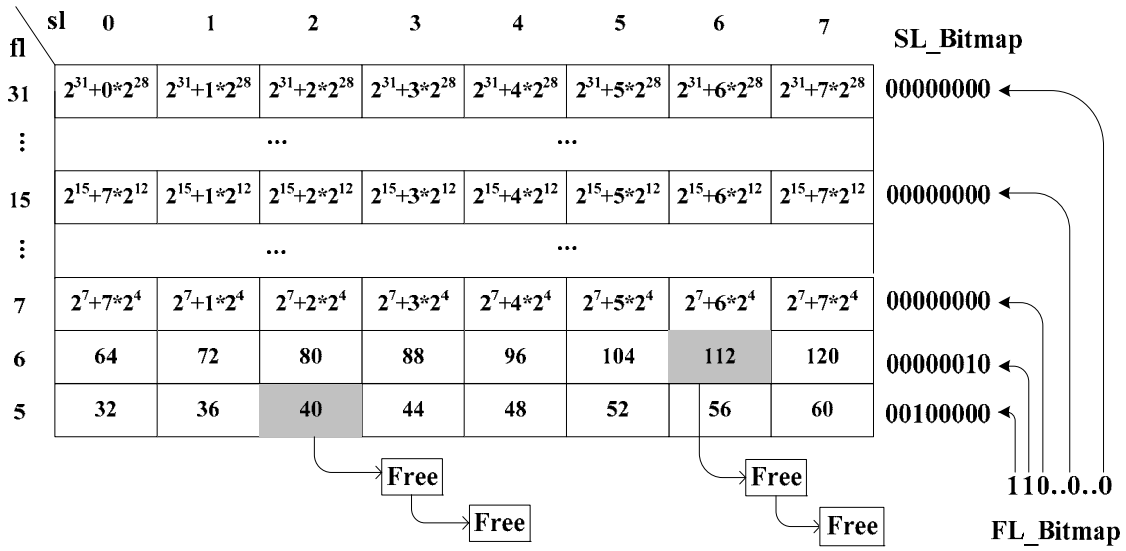

图 3.1 TLSF 算法数据结构图

Fig.3.1 TLSF algorithm data structure diagram

每个区间里都有空闲内存块和已使用的内存块，空闲内存块是待分配的内存块，它们通过链表链接在一起，已使用的内存块是已经分配出去的内存块，空闲块和已使用内存块的结构图如下图 3.2 及图 3.3 所示。由这两个图可以看出，空闲内存块和已使用内存块都是由两部分组成—头部信息及存储空间，图中阴影部分代表头部信息，其余部分为内存空间。头部信息包括内存块大小及其标志位(T 为最后一个物理块标志，其值为 1 时表示该内存块是最后一个内存块；F 为空闲内存块标志，其值为 1 表明该内存块是空闲内存块)、指向前一个物理内存块的指针、

指向后一个空闲内存块的指针、前一个空闲内存块的指针。

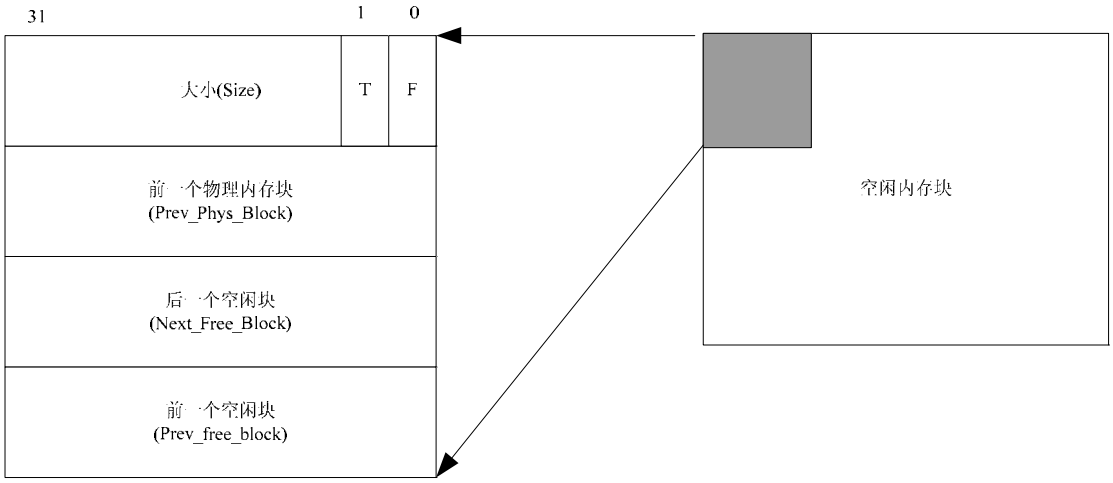

图 3.2 空闲块结构示意图

Fig.3.2 Free block structure diagram

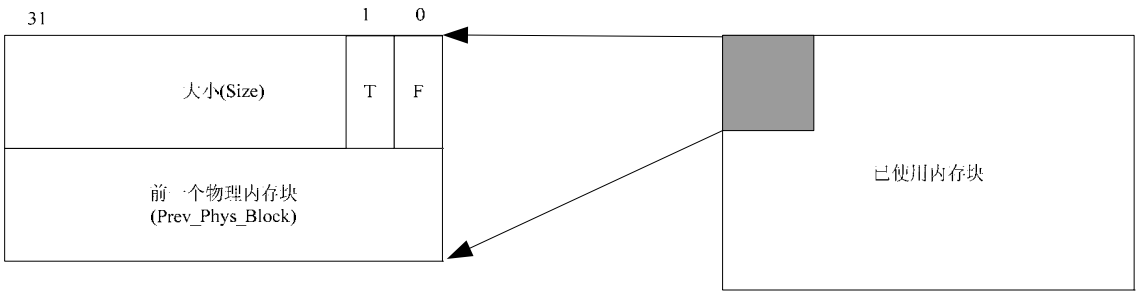

图 3.3 已使用内存块结构示意图

Fig.3.3 Used memory block structure diagram

已使用的内存块块头信息通过结构体 bhdr_t 来保存，如图 3.3 所示。该结构体保存前一个物理内存块首地址，以及内存块的大小及内存块的空闲状态信息。结构的具体实现如下：

```
typedef struct bhdr_struct {
    /*   指向前一个物理内存块*/
    struct bhdr_struct *prev_hdr;
    /*  内存块的大小*/
    size_t size;
    /*  联合用来保存内存块空闲状态信息  */
```

```
    union
    {
        struct free_ptr_struct free_ptr;
        u8_t buffer[1];
    } ptr;
} bhdr_t;
```

未使用的空闲内存块的块头信息通过结构体 bhdr_t 和结构体 free_ptr_t 来保存，如图 3.2 所示。其中 free_ptr_t 结构体用于保存前后空闲内存块的地址信息。结构的具体实现如下：

```
typedef struct free_ptr_struct
{
    /*指向前一个空闲内存块的指针*/
    struct bhdr_struct *prev;
    /*指向后一个空闲内存块的指针*/
    struct bhdr_struct *next;
} free_ptr_t;
```

每个内存分区的基本信息都存在结构体 tlsf_t 里, tlsf_t 结构体位于每个内存分区的头部。该结构体如下所示。

```
typedef struct TLSF_struct {
    /* TLSF 算法标识符/标志符，用于区别别的算法*/
    u32_t tlsf_signature;
    /*  用于链接所有的内存区（池），把不连续的内存区链接起来  */
     area_info_t *area_head;
    /*  位图，用于标志所在空闲链表中有无空闲块，有为 1,*/
    u32_t fl_bitmap;
    u32_t sl_bitmap[REAL_FLI];
    /*  空闲内存块链表，用于存储相应空闲内存块*/
    bhdr_t *matrix[REAL_FLI][MAX_SLI];
} tlsf_t;
```

该结构体使用成员变量 fl_bitmap、sl_bitmap 描述索引表，matrix 存储相应空闲内存块的地址，结构体通过这些变量来记录整个内存区中空闲内存块的使用情况。

内存分区信息对应的结构体为—结构体 area_info_t, 该结构体用来保存不同内存区的基本信息。链表可以通过该结构体链接各个不相邻内存区，结构的具体实

现如下：

```
typedef struct area_info_struct {
    /*指向内存分区的最后一块空闲内存块*/
    bhdr_t *end;
    /*指向下一个内存分区的首地址*/
    struct area_info_struct *next;
} area_info_t;
```

3.2.3 内存分配和回收

TLSF(Two-level Segregated Fit)算法用到的函数有：内存初始化函数 init_memory_pool(size_t mem_pool_size, void *mem_pool)、内存区销毁函数 destroy_memory_pool(void *mem_pool)、增加一个新内存区函数 add_new_area(void *area, size_t area_size, void *mem_pool)、动态内存分配函数 tlsf_malloc(size_t size)、动态内存释放函数 tlsf_free(void *ptr)、动态内存再分配函数 tlsf_realloc(void *ptr, size_t size)等。

TLSF 算法给用户提供了 3 个接口，分别为内存初始化、动态内存分配及动态内存释放。

①动态内存初始化：动态内存初始化的目的是申请一块大的内存区，并初始化 tlsf_t，调用函数 process_area()处理内存块，对申请到的内存区进行初始化。

②动态内存分配：调用 tlsf_malloc()函数进行动态内存分配，该函数的输入参数为动态内存申请的大小，函数根据内存大小，查找空闲内存块，若分配成功，则返回它的地址。

TLSF 动态内存分配的步骤如下：

内存分配的操作步骤为：

1) 查找一级二级索引值。通过 MAPPING_SEARCH(&size, &fl, &sl)函数查找到查找满足所需内存大小的一级与二级索引 fl 和 sl, 根据 fl 与 sl 的值得到适合的内存块的链表的表头，若链表为空闲，则将列表的头结点取下来，并将内存块头部信息的空闲状态标志位 F 置为 0。
2) 返回空闲内存块所在链表的表头地址。根据一级二级索引值，通过函数 FIND_SUITABLE_BLOCK(tlsf, &fl, &sl)找到合适的空闲内存块, 并返回空闲内存块所在链表表头的地址。
3) 调整位图。将空闲内存块对应位图的值清零。
4) 进行内存切割。比较空闲内存块大小 size_t 和实际内存请求大小 size，若 size_t 的值大于二级索引划分的内存块区间最小值，则将空闲内存块按照此最小值进行内存切割，并将切割后的新空闲块插入到相应子链表中并对

相关信息进行更新。

5) 返回空闲内存块地址。

③动态内存释放：通过 tlsf_free()函数进行内存释放，它将要释放的空闲内存块插入到原来的链表中，函数的参数只有一个—释放内存块的物理地址。动态内存释放的具体步骤为：

1) 查找相邻物理地址内存块。获取与要释放的空闲内存块相邻的内存块的物理地址，并判断相邻内存块的空闲状态。
2) 合并内存。若相邻内存块是空闲内存块，则进行合并，否则，不合并。
3) 插入。将空闲内存块插入到对应的链表中，更新链表的相关信息。
4) 调整位图。将空闲内存块对应位图置位。

3.3 ERMM 内存分配管理器

TLSF(Two-level Segregated Fit)算法虽然实时性能优秀，但实时性及内存碎片率均有进一步优化的余地。

在 TLSF 算法中，内存分配时，内存切割采用“取下限”策略，按照二级索引划分的内存块区间最小值进行内存切割，内存块使用完后仍可以插入到原先的内存分区里，下一次再重新申请类似大小的内存时就可再次从该内存区分配。虽然这样保证了实时性约束，能很好地利用新释放的内存块，但以空间换时间的方式造成大量的内部碎片，内存资源利用率较低。本文以精确切割作为内存块的分割条件，并设定切割门限，只有待分配的内存和空闲内存块差值大于切割门限时才进行精确分割，否则，按照内存块区间最小值进行切割。“精确切割”的内存方式减少了内存碎片，并最大限度地提高了内存利用率。

TLSF 算法在内存释放时，查找相邻物理内存块并判断其状态，如果内存块是空闲的，则立即进行内存合并。立即合并增大了空闲内存块的大小，但在释放内存时，导致开销增大，并且在处理近似大小内存重复分配时，没有利用内存重复分配这一点[47]，导致其性能下降，降低了实时性。本文定义了一个合并阈值，并计算当前系统内存碎片率，比较合并阈值与当前系统的内存碎片率，只有当内存碎片率大于合并阈值时才对相邻的空闲内存块进行合并。这种方式减少了内存释放时间，在处理近似内存大小重复分配时具有更好的实时性。

针对 TLSF 算法的上述缺点, 本文在其基础上提出了一个新的嵌入式实时系统动态内存分配管理器—ERMM(Embedded Real-time Memory Manager)，它将内存请求分为小内存请求和大内存请求，并对这两种内存请求提出了新的数据结构。针对小内存请求，采用二级位图算法，通过以空间换时间的策略降低分配时间，减少外部内存碎片；针对大内存请求，在 TLSF 算法的基础上改变二级索引结构，在

内存分配时采用“精确切割”, 在内存释放时采用“合并阈值”策略, 提高内存利用率, 降低内存碎片。并且，在空闲内存块中添加了内存越界保护，提高了内存管理器的安全性。

3.3.1 提出新的数据结构

在大多数系统中，小内存的分配请求远远多于大内存的请求[5]。本文对不同大小的内存请求采用不同策略，小内存请求采用二级位图算法，大内存请求采用改进后的 TLSF(Two-level Segregated Fit)算法。大小内存的划分可根据具体系统进行设定，定义变量 MEM_VAL，内存值小于 MEM_VAL 定义为小内存，大于 MEM_VAL 为大内存。本文定义 MEM_VAL 的值为 128B。

ERMM(Embedded Real-time Memory Manager)在系统中划分 2 个内存分区 MP1 及 MP2，MP1 为小内存分配区，MP2 为大内存分配区，每个内存分区都有它自己的内存控制块，它记录内存分区的相应信息。MP1 分区被划分为大小相等的内存块，响应小内存的分配请求。MP2 的内存空间按照改进后的 TLSF 算法数据结构进行划分，通过二级索引进行链接，响应大内存的分配请求。ERMM 整体时间复杂度为 O(1)。

①小内存分配区数据结构

小内存分配时，产生的内部内存碎片本身就很小，对小内存频繁地分割会降低分配效率。本文将内存分区 MP1 分割为大小相等的内存块，在小内存分配时，直接分配整个内存块，提高了内存分配效率，保证了实时性，同时消除了外部碎片。虽然直接分配整个内存块会产生一些内部碎片，但由于每份内存块产生的内部碎片都很小，内部碎片总量也不大。

内存控制块(Memory Control Block)记录内存的分配信息，它是一个数据结构，用 OS_MCB_Bitmap 表示，内存分配成功后被赋值。OS_MCB_Bitmap 有两个数据结构—OSMemGrp 和 OSMemTbl[]，它们描述内存分配表。内存分区 MP1 被分成相等的 1024 块，每 32 个内存块为一组，每个内存块有相应的内存序号，当 MEM_VAL 为 128B 时，数据结构如图 3.4 所示。

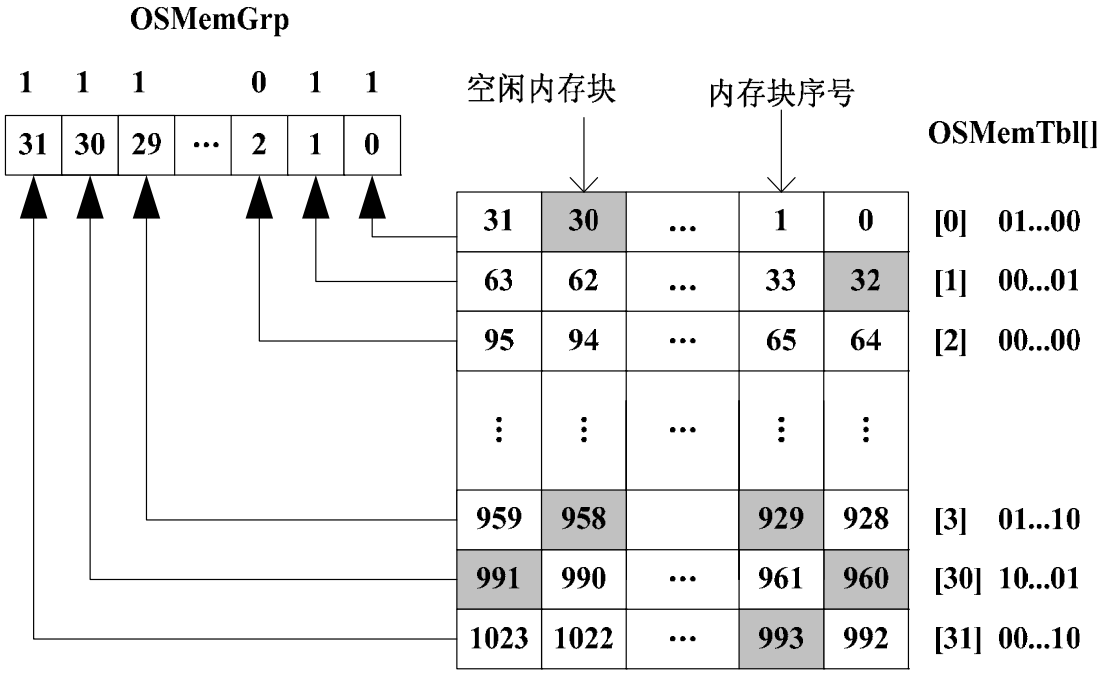

图 3.4 ERMM 小内存分配区数据结构图
Fig.3.4 ERMM memory allocation area data structure diagram

②大内存分配区数据结构

大内存分配的数据结构在 TLSF 算法基础上加以改进。在 TLSF 算法的数据结构中，小内存块和大内存块的数量是相同的，但在实际应用中，小内存的动态内存分配需求高于大内存的分配需求，小内存分配时若无合适的空闲内存块只有申请更大的空闲内存块，这降低了内存利用率并增大了内存碎片。

ERMM 在 TLSF 算法的基础上将内存分区 MP2 按照 2 的幂次方分割成不同大小的内存区间，每个内存区间有大小相同的内存块，且内存块的数量随内存块大小的增加而递减，较小内存块的数量高于较大内存块的数量，这充分利用了较小内存分配请求相对大内存请求要频繁的特点, 一定程度上克服了 TLSF 算法中小内存块数量不足只能分配大内存块导致内存碎片增加的缺点。

改进后的数据结构将空闲内存块通过二级索引组织起来，第一级索引将内存分区 MP2 分割成 24 个区间，第 0 个区间大小是 $24*[0,2^8-1)$，第一个区间是 $23*[2^8-1,2^8+1)$，其他区间的区间范围是 $22*[2^{start},2^{start+1})$，$21*[2^{start+1},2^{start+2})\ldots[2^{end-1},2^{end})$。第二级索引在第一级索引的基础上再进行线性分割，分割数目依次递减。当 start=9、end=30 时，MP2 分区状况如图 3.5 所示。

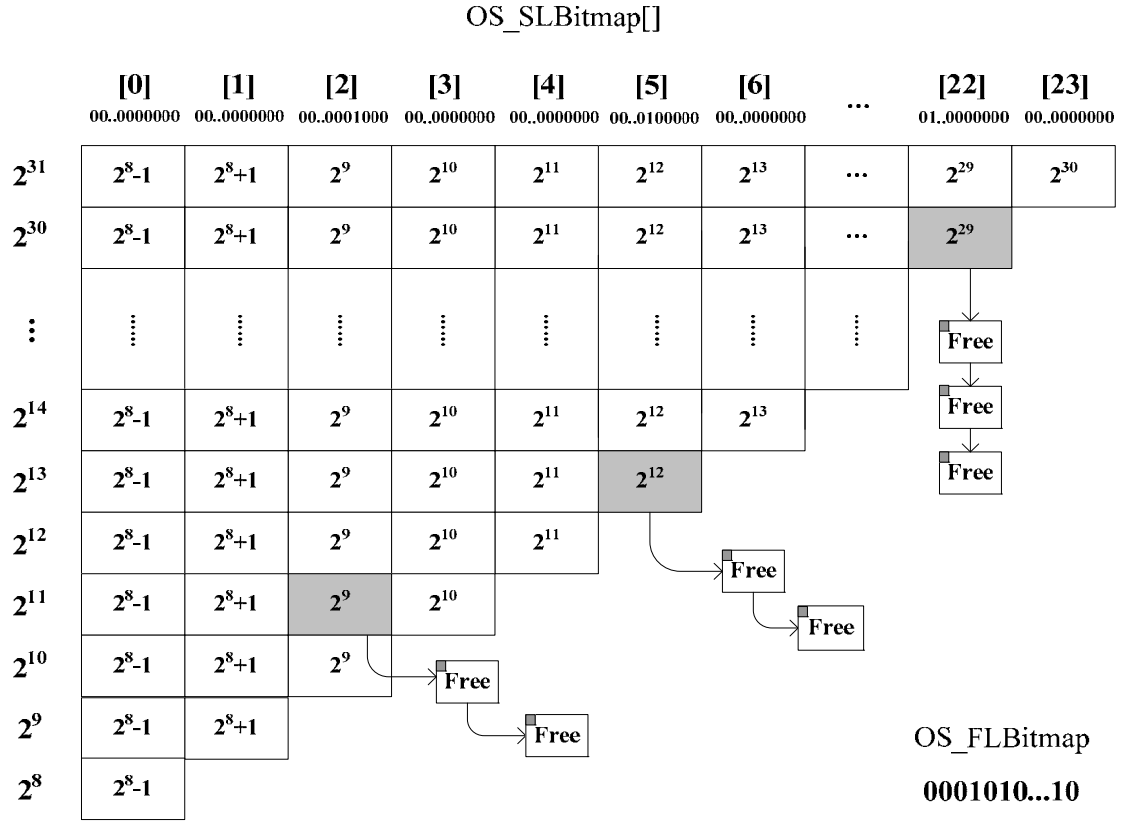

图 3.5 改进后的 TLSF 数据结构图

Fig.3.5 The improved TLSF data structure diagram

内存控制块记录内存分区 MP2 内的空闲内存块的分配情况及对应的链表指针。内存控制块的数据结构如下：

```
static struct ERMM_struct
{
    u32_t   OS_FLBitmap;
    u32_t   OS_SLBitmap[REAL_FLB];
    bhdr_t *matrix[REAL_FLB][MAX_SLB];
} ERMM_t;
```

OS_FLBitmap 及 OS_SLBitmap[REAL_FLB]描述二级位图, 每个内存块对应位图中的 1 位，位为 1 表示有空闲内存块，为 0 则表示没有，matrix[REAL_FLB][MAX_SLB]存储空闲链表头指针。

每个内存块的空闲状态标志都放入内存分配表中，OSMemGrp 中的每一位代表相应组中是否有空闲内存块，OSMemTbl[n]则记录第 n 组内空闲内存块分布情况。内存分配时，通过查找 OSMemGrp 及 OSMemTbl[]最低有效位，获取空闲内

存块的序号，根据该序号得到空闲内存块的地址。

③内存块添加边界标记

为了减少内存碎片，ERMM(Embedded Real-time Memory Manager)在空闲内存块和已分配的内存块头部和尾部都添加了存储内存块大小及空闲标志等信息，使内存块在释放时可以更快地合并。内存释放时，可以通过内存块的头部及尾部信息迅速判断相邻物理内存块的空闲状态及空闲内存块的大小，使得内存块的释放及合并在常数时间内完成。空闲内存块及已分配内存块的结构示意图如图 3.6、图 3.7 所示。

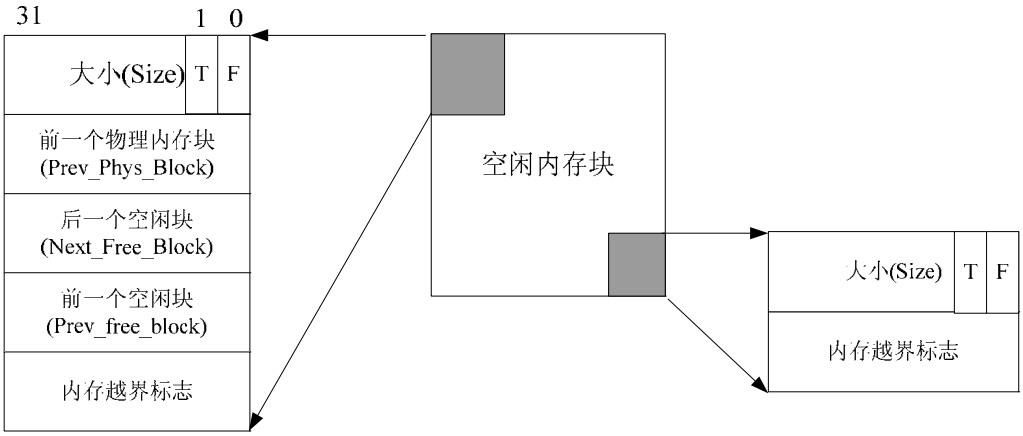

图 3.6 空闲块结构示意图

Fig.3.6 Free block structure diagram

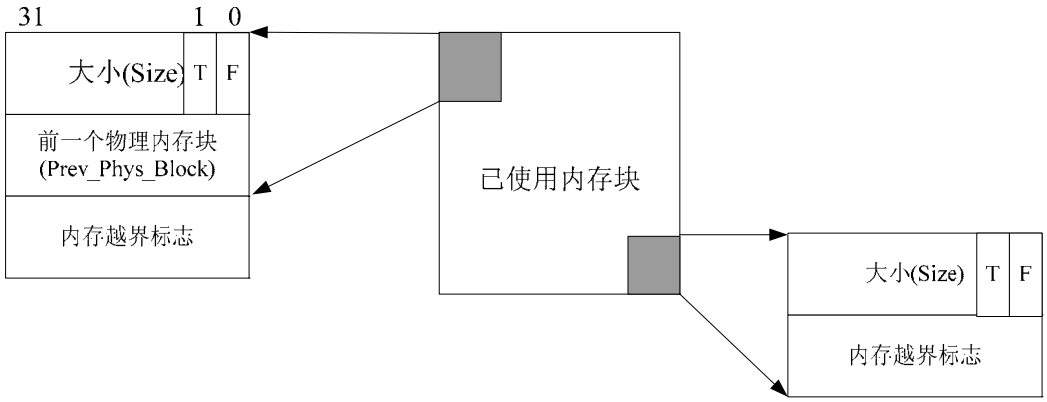

图 3.7 已使用内存块结构示意图

Fig.3.7 Used memory block structure diagram

为了防止内存越界访问，在空闲内存块和已分配的内存块头部和尾部都添加了一个 32 位的“越界保护内存”区域，该区域用于防止内存写越界并提供写越界检测。当有内存越界访问时，通过检测内存越界标志里的值是否被改变，来判断是否发生了内存越级访问，给出相应提示，并中止系统的运行。

3.3.2 算法改进

①小内存区的分配和释放

小内存的分配采用位图算法。相较于传统的位图算法，本文通过查找就绪表获取空闲内存块序号，直接进行地址分配。在查找空闲内存块时采取以空间换取时间的策略，提高了查找时间及分配速度。

内存分配表标识每个内存块的空闲状态，如上图 3.5 所示，OSMemGrp 中的每一位代表相应组中是否有空闲内存块，OSMemTbl[n]则记录第 n 组内空闲内存块分布情况。内存分配时，通过查找 OSMemGrp 及 OSMemTbl[]最低有效位，获取空闲内存块的序号，根据该序号得到空闲内存块的地址。内存分配后，内存分配表 OSMemTbl[]及 OSMemGrp 中的相应元素被清零。内存释放时，内存分配表中相应元素的相应位置位。

②大内存区内存分配

内存分配时，TLSF 算法中内存切割采用“取下限”策略，按照二级索引划分的内存块区间最小值进行内存切割，虽然这样保证了实时性约束，能很好地利用新释放的内存块，但以空间换时间的方式造成大量的内部碎片，内存资源利用率较低。ERMM 以精确切割作为内存块的分割条件，并设定切割门限，只有待分配的内存和空闲内存块差值大于切割门限时才进行精确分割，否则，按照内存块区间最小值进行切割。“精确切割”的内存方式减少了内存碎片，并最大限度地提高了内存利用率。

在内存释放时，TLSF 算法内存合并采用立即合并策略，TLSF 在内存释放时首先查看相邻物理内存块是否是空闲内存块，若是的话则立即进行合并。这种方式增大了空闲内存块的大小，但在释放内存时，导致开销增大，并且在处理近似大小内存重复分配时，没有考虑到这样一种情况：TLSF 算法在内存块被释放后，会查找相邻物理地址是否存在空闲内存块，若存在，则将和相邻的空闲内存块合并成一个更大的空闲内存块，虽然这样减少了内存碎片率，但是若后面申请的内存大小和之前释放内存大小一样，TLSF 算法则会将刚合并好的空闲内存中再次分割，分配和上次释放时相同大小的空闲内存块。这明显地导致了 TLSF 算法内存分配时没有利用内存重复分配这一点，做了没必要的内存合并及再切割操作，导致分配效率下降，降低了实时性。ERMM 则采用延时合并策略，定义了一个合并阈

值，并计算当前系统内存碎片率，只有当内存碎片率大于合并阈值时才对相邻的空闲内存块进行合并。这种方式减少了内存释放时间，在处理近似内存大小重复分配时具有更好的实时性。

另外，在内存释放时，ERMM 采用边界标示技术的使用也加快了相邻物理地址空闲内存块的合并。通过检测内存块块头信息中的内存块空闲状态标识，来判断该内存块的状态是已使用还是空闲。内存释放时，根据该空闲内存块状态快速将空闲内存块合并，以形成更大的空闲块，降低内存碎片率。

③内存保护

TLSF 算法没有对内存进行保护，内核系统和用户空间公用一个物理地址空间，很容易发生内存越界访问，造成不可预知的后果。ERMM 通过在内存区的底部设有内存越界保护内存区，在内存释放时通过检查内存保护区的数值来确定是否存在内存写越界。

3.3.3 小内存分配与回收

ERMM 收到用户的动态内存分配请求后，首先判断内存请求的大小，若该内存小于等于 MEM_VAL，并且 MP1 中有空闲内存块，则在 MP1 中用二级位图算法进行分配；若大于 MEM_VAL，在 MP2 中用改进后的 TLSF 算法进行分配。内存释放的过程则相反。

① 小内存分配过程

1）查找最低空闲内存块序号

查询内存分配表中 OSMemGrp 及对应的 OSMemTbl[]中的最低有效位，获取该内存块的序号。相应的程序如下：

```
y = ls_bit(OSMemGrp);
x = ls_bit(OSMemTbl[y]);
memBlkSN = (y << 5) + x;
```

其中，y 为 OSMemGrp 最低有效位位置，x 为 OSMemTbl[y]最低有效位位置，memBlkSN 为空闲内存块序号。ls_bit()用于查找最低有效位，其核心代码为：

```
/ /保留 i 最低有效位，其它位清零
x = i & -i;
a = x <= 0xffff ? (x <= 0xff ? 0 : 8) : (x <= 0xffffff ? 16 : 24);
//查找 i 的最低有效位置
low_bit = table[x >> a] + a;
```

table 表是一个大小为 256 的数组，它存储了索引值的最低有效值的位置，通过查表求取 i 的最低有效位的位置 low_bit，以空间换时间的方式提高了查找效率，上述算法时间复杂度为 O(1)。

```
/*数组[256] 索引值（下标）的最高有效值的位置，0 的最高有效位为-1*/
static const int table[] =
{
   -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
   5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
};
```

查找空闲内存块最低序号时，通过查 table 表求取整数 i 的最低有效位的位置，以空间换时间的方式提高了查找效率，时间复杂度为 O(1)。

在 CM3 内核中可以通过相应的指令来实现求最低有效位的位置：

```
ls_bit:
    CMP          R0, #0
    ITTE         NE
    RBITNE.W     R1, R0
    CLZNE.W      R0, R1
    MOVEQ        R0, #-1
    BX LR
```

2）调整 bitmap 位图

查找就绪表空闲内存块相对应的位，并将该位置位。核心代码为：

```
OSMemTbl[y] &= ~(0x01<<x);
OSMemGrp &= ~(!OSMemTbl[y]<<y);
```

3）返回所需大小的内存首地址

由于每个内存块的大小是固定的，空闲内存块的地址和空闲内存块序号成正相关。空闲内存块地址 = memBlkSN * 基址。

② 小内存释放过程

查找空闲块的分配位置，内存按块进行分配，内存释放只是将分配的内存块对应的内存分配表相应元素清零。

```
m = memBlkNum & 0x1F;
n = memBlkNum >>5;
```

OSMemTbl[m] |= 0x01<<m;

OSMemGrp |= 0x01<<n;

3.3.4 大内存分配与回收

大内存的分配和释放是通过改进后的 TLSF 算法在内存分区 MP2 中进行。内存分配时, 首先查找一级二级索引, 寻找合适的空闲内存块, 然后采用不同于 TLSF 算法的“精确切割”方式切割内存, 即将空闲链表里面的空闲内存块按照请求分配内存大小进行切割，剩余的部分再插入到链表中，最后把 bitmap 中相应的位清零。内存释放时，先判断相邻块中是否有空闲内存块，若有空闲内存块，不同于 TLSF 算法的立即合并相邻空闲块，而是根据内存的碎片率决定是否合并空闲块，计算当前系统的内存碎片率，并将该内存碎片率和合并阈值进行比较，只有当内存碎片率大于合并阈值时才进行内存碎片率的合并。

① 大内存分配过程

1）查找一级及二级索引

查找一级及二级索引相应的代码为：

```
*_fl = ms_bit (*_r) - FIRST_LOG2_INDEX;
//得到*_r 值的最高有效位（二进制）的位置
if (fl_bitmap | !(0x01<<*_fl)
{
    //查找二级索引
    *_sl = ms_bit(sl_bitmap[*_fl]);
}
```

其中，*_fl 是一级索引值，*_sl 是二级索引值，ms_bit 函数查找最高有效位的位置。FIRST_LOG2_INDEX 为一级索引起始值。

2）寻找合适的块

matrix 数组中存储了指向空闲内存块链表的指针, 通过一级和二级索引值得到空闲内存块地址 addr。

```
addr = ERMM_t->matrix[*_fl][*_sl];
```

3）切割内存

TLSF 算法中内存分割采用了“取下限”策略，即当请求分配的内存大小小于空闲链表中空闲内存块的大小时，按照二级索引划分的内存块区间最小值进行内存切割，虽然这样做能够保证实时性约束，即当下次有类似大小的动态内存分配请求时，可以快速将该空闲内存块分配出去。但利用空间换时间的方式造成大量的内部碎片，内存资源利用率较低。本文以精确分割作为内存块的分割条件,设定切割门限，当待分配的内存和空闲内存块差值大于切割门限进行精确分割，否则不

进行内存分割。

具体的实现是比较待分配的空闲内存块 size_t 和请求内存大小 size, 如果 size_t – size 的值大于切割门限，则对空闲内存块进行精确切割，否则不进行切割。这样既避免了“取下限”策略带来的内存碎片, 又很好地防止对额外的小内存进行切割产生的小内存碎片。

4）调整位图

根据一级与二级索引值得到内存块，将空闲内存块对应 bitmap 位图的相应位清零。

② 大内存释放过程

1）查找相邻空闲块是否有空闲内存块

首先查找待释放内存块物理空间附近是否有相邻空闲块，若有，则进入 2)，否则转到 3)中执行。

2）根据内存碎片率合并空闲内存块

按照文献[8]中定义, 以实际分配内存大小的最大值与请求内存大小的最大值的差计算当前系统的内存碎片率，表示为公式 3.1：

$$F = \frac{M_{max} - M_{real}}{M_{max}} \tag{3.1}$$

公式中，F 为碎片率，M_{real} 为实际分配内存的最大值，M_{max} 为理想情况下分配内存的最大值。该公式适用于计算外部碎片，内存碎片是大于等于 0 小于等于 1 的小数。

定义合并阈值 L，它是由经验决定的。当 F 小于 L 时暂不合并，当 F 大于 L 时，查找待释放内存块是否有相邻空闲内存块，若有则立即合并。

3）插入链表并调整位图

查找一级二级索引，通过该索引把释放的内存块插入相应链表的表头，再次形成一个完整的双向链表，并使相应 bitmap 标志位置位，完成动态内存的释放。

3.3.5 内存保护

嵌入式实时操作系统有保护模式和无保护模式两种内存空间模式[48]。

①保护模式

在保护模式下，CPU 利用对虚拟地址的支持为系统内存资源提供不同访问权限，操作系统和用户程序的权限不同，对资源的访问权限也不同，同时虚拟地址也进程资源共享，进程管理提供了有力支持。

保护模式分为系统/用户保护模式和私有保护模式[49]。在系统/用户保护模式中，通过页式保护机制来实现内存保护。系统模式模式是分配给内核系统的页面具有很高特权级，用户模式是分配给用户的页面具有一般的特权级。给不同的页

设置不同的访问权限，从而对内存进行保护。通过 MMU 完成虚拟地址到实际物理地址的转换，页式保护机制判断页面的权限，控制应用程序对页面的访问。私有保护模式下，任务运行在虚拟地址的内存空间中，操作系统通过任务的段描述符对任务进行管理[50]。

②无保护模式

在无保护模式中，代码段和数据段共享整个物理内存空间，内核空间和用户空间没有任何保护。段地址从 0 开始，到系统中实际物理地址的最高地址结束。这种情况下，操作系统无需要对地址空间和保护模式进行管理[49]。

在嵌入式实时系统中，处理器一般没有内存管理单元，无法使用虚拟内存保护，将内存的逻辑地址转换为物理地址，无保护内存空间模式更为常见。比如，在 VxWorks 嵌入式实时系统中，所有任务共用一个一个内存地址空间。程序直接访问物理地址，无需进行虚拟地址到物理地址的转换。系统和用户共享一个内存空间，嵌入式实时系统对内存没有保护，若应用程序发生内存越界访问，并修改了系统空间的内容，会产生不可预料的后悔，甚至会导致整个系统的崩溃[49]。

本文在充分考虑嵌入式设备的限制及其对内存管理模块的要求的基础上，结合嵌入式实时系统对内存管理的要求，提出了一套内存保护方案。通过内存加墙实现保护，在每一个内存块的头部及尾部添加 4 个字节，将该字节的值初始化为一个一般不会出现的特殊数值，本文定义该值为 0XFEEEEEEE，模块内存初始化时把池集内存区和动态内存区都初始化为 0XFEEEEEEE，内存释放时，判断该值是否发生改变，若该边界标志的内容已经变化，说明内存操作有内存越界行为，由于内存越级会产生严重问题，此时需要报警，并中止程序的运行。内存块的结构图，如图 3.8 所示。

图 3.8 内存块结构图

Fig.3.8 Memory block diagram

在内存块结构图中，内存块体才是真正被用户使用的内存区域。内存块的前后有一小块“越界保护内存”区域，该区域用于防止内存写越界并提供写越界检测。用户块后有一个指针域“内存块头指针”, 该内存块头指针除了保存内存块相应的信息外，还保存了指向内存块头的指针，用于内存释放。

3.4 ERMM 内存分配管理器理论分析

下面从时间复杂度和内存碎片率两个方面对 ERMM(Embedded Real-time Memory Manager)内存管理器的内存分配和释放过程进行分析。

①时间复杂度分析

尽管 ERMM 在内存分配的时候需要查找链表，但内存操作函数 malloc 和 free 在执行和查找内存操作中，都没有循环函数，时间复杂度为 O(1)，并且查找操作是通过位图实现的，这提高了内存查找的效率。

动态内存分配操作主要包括：内存大小判断、定位空闲内存块、获取内存块、根据内存切割策略对内存进行切割。针对小内存的分配请求，ERMM 利用位图查找空闲内存块，时间复杂度为 O(1)，并且由于小内存分配是直接分配固定大小的空闲内存块，无需对内存块进行切割，分配时间很快；针对大内存请求，ERMM 通过位图在常数时间内查找到一级二级索引，在最坏情况下，内存分配需要定位两次，并进行内存切割，但相关的操作函数中均没有循环，操作均可在常数时间内完成，最坏执行时间是有界的。

动态内存释放操作主要包括：查找相邻内存块、合并空闲内存块、插入空闲内存块。对小内存的释放，只有插入空闲内存块操作，插入空闲内存块只需获取空闲内存块的序号即可，时间复杂度为 O(1)，无需对空闲内存块进行合并；针对大内存的释放，TLSF(Two-level Segregated Fit)算法在内存释放时，查找相邻物理内存块并判断其状态，如果内存块是空闲的，则立即进行内存合并。这种方式增大了空闲内存块的大小，但在释放内存时，导致开销增大，并且在处理近似大小内存重复分配时，没有利用内存重复分配的特点，导致其性能下降，降低了实时性，ERMM 定义了合并阈值来，通过是否高于合并阈值来决定是否进行内存合并，这种方式减少了内存释放时间，在处理近似内存大小重复分配时具有更好的实时性。

②内存碎片率分析

内存碎片分为内部碎片和外部碎片。ERMM 将内存请求分为小内存请求和大内存请求, 针对不同的内存请求采取不同的分配策略, 对于小于 128B 的内存请求，直接分配大小为 128B 的内存块。由于无需对内存进行分割，同时使用了位图算法查找空闲内存块，加快了内存分配效率，提高了实时性。虽然分配整个内存块会

导致内部碎片的产生，但由于内存块本来体积就很小，内部碎片的总量也不是很多。针对大内存，ERMM 在 TLSF 算法的基础上，首先，修改了数据结构，结合在大多数系统中小内存的分配请求远远多于大内存的请求的特点[4], 提高较小内存块的数量, 一定程度上减少了 TLSF 算法中由于小内存已经分配完只好分配更大内存增加了内存碎片的问题；其次在内存分配时，采用“精确切割”的策略对内存进行切割，相比 TLSF 算法的“取下限”切割策略，减少了内存碎片。

3.5 本章小结

本章首先从 TLSF(Two-level Segregated Fit)算法的重要参数、数据结构及内存分配和回收方法上对 TLSF 算法进行阐述, 分析 TLSF 算法的内存分配和释放流程, 指出 TLSF 算法在实时性及碎片率方面的不足, 在此基础上提出新的动态内存管理器 ERMM，提出一个新的数据结构，使用位图算法进行小内存分配，在 TLSF 算法的基础上改进内存分配和释放策略以进行大内存的分配方案，并增加内存保护方案以提高内存管理的可靠性与安全性，最后从时间复杂度、内存碎片率上对 ERMM 内存管理器进行了理论分析。

4 ERMM 在 μC/OS-III系统上的实现

4.1 引言

μC/OS-III系统是一个新推出的嵌入式实时系统，它是 μC/OS-II 的升级版本，它已经应用在几乎所有的平台上。μC/OS-III系统现在已经成功地移植到 X86 体系的处理器上，但是移植过程是在 DOS 环境下进行的，本文则在 Windows 操作系统环境下对 μC/OS-III系统进行移植。并通过 VC++ 6.0 开发环境进行调试。

本文将嵌入式实时系统动态内存分配管理器 ERMM(Embedded Real-time Memory Manager)在 μC/OS-III系统上进行实现，由于 ERMM 是在 TLSF(Two-level Segregated Fit)算法的基础上加以改进，因此，本文首先介绍了 μC/OS-III系统的内存管理，然后将系统在 VC 环境下移植到 X86 体系的 CPU 上，在 VC 环境下将通过修改 μC/OS-III系统内存管理相关的数据结构及函数实现了 TLSF 算法。最后通过修改 TLSF 算法的相关数据结构及内存分配和释放函数，将 ERMM 内存管理器在 μC/OS-III系统进行实现。

4.2 μC/OS-III系统内存管理

4.2.1 μC/OS-III系统内存管理

μC/OS-III是一种开源的、可移植、固化、剪裁的占先式实时多任务内核，它具有执行效率高、占用空间小、实时性能优良和扩展性强等特点。μC/OS-III已经移植到了几乎所有知名的 CPU 上。

μC/OS-III系统将系统内存分成若干个不同大小的内存分区。每个内存分区再被分成整数个大小相同的内存块，如图 4.1 所示。

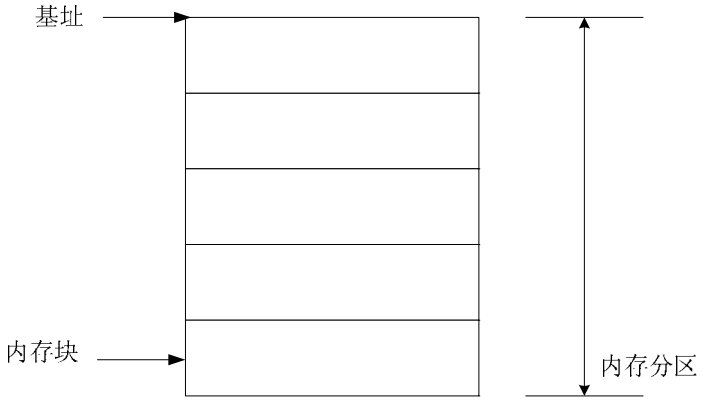

图 4.1 内存分区示意图

Fig.4.1 Memory partition schematic diagram

系统对内存分配函数 malloc()和内存释放函数 free()进行修改，对于不同大小的内存分配请求，malloc 函数分配固定大小的内存块，内存释放时，free 函数也释放固定大小的内存块。通过这种方式，保证了内存分配和释放的时间，满足系统实时性要求。

不同内存分区里的内存块大小时不同的，针对不同的内存请求，系统内存管理首先查找对应的内存分区，从合适的内存分区中分配内存块。内存释放时，待释放的内存块要重新放回它分配时所属于的内存分区。

μC/OS-Ⅲ使用“内存控制块”（MCB）来具体管理内存分区及分区中的内存块，分区与 MCB 有“一对一”的关系。

内存控制块的数据结构为：

```
typedef struct {
    /*指向内存分区起始地址的指针*/
    void *OSMemAddr;
    /*指向下一个内存控制块或空闲内存块的指针*/
    void *OSMemFreeList;
    /*内存分区中内存块的大小*/
    INT32U OSMemBlkSize;
    INT32U OSMemNBlks;
    INT32U OSMemNFree;
} OS_MEM;
```

空闲内存块通过链表进行链接。空闲内存块链表如图 4.2 所示。

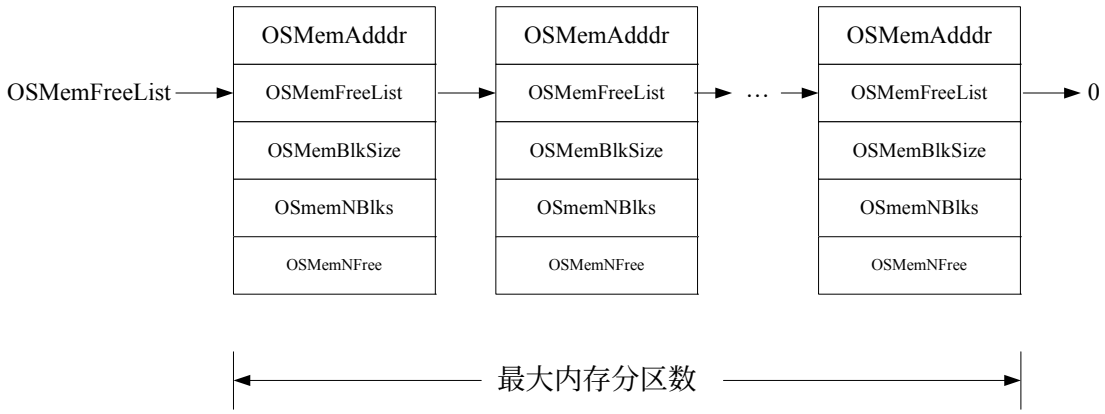

图 4.2 空闲内存控制块链表

Fig.4.2 List of free memory control blocks

有内存请求时，通过一级二级索引找到空闲内存块链表，把链表的第一个空闲内存块分配给应用程序，同时把空闲内存块链表的头指针 OSMemFreeList 指向下一个空闲内存块。内存释放时，将该内存块释放到相应的内存分区中，将该空闲内存块插入空闲内存块链表的头结点，并将头指针 OSMemFreeList 指向该内存块，该内存块的头部信息中的指向下一个空闲内存块的指针 Next_Free_Block 指向原来内存块空闲链表的第一个内存块[51]。

在使用一个内存分区之前，必须先建立该内存分区。这个操作可以通过调用 OSMemCreate()函数来完成[30]。

内存申请函数为 OSMemGet()，应用程序通过调用该函数从已建立的内存分区中申请一个内存块。该函数的输入参数是 OSMemCreate()函数返回的指向内存分区的指针[15]。

内存释放函数为 OSMemPut()，当内存块使用结束后，必须及时地将其释放，通过调用 OSMemPut()函数将内存块放回到相应的内存分区中[30]。

4.2.2 μC/OS-Ⅲ系统内存管理的不足

μC/OS-Ⅲ的动态内存管理模块直接操作物理内存，提高了系统实时性，由于是整块内存块分配，也不会产生外部内存碎片，但它提供的功能十分有限，存在着以下不足：

① 动态管理的内存块尺寸须在编译时已确定，每个内存分区的内存的起始地址、内存块数量是创建内存分区时设置的，创建后不能再修改，这限制了应用程序的灵活性，同时也造成内存浪费。

② 同一分区的内存块尺寸是相同的，而实际的内存请求大小千差万别。为了满足不同的内存请求，减少资源浪费，需建立不同大小的内存区，这增加了系统开销。

③ 未提供内存分区的释放和合并功能，μC/OS-Ⅲ的内存块虽然可以释放，但没有提供内存分区释放功能，造成这部分内存空间一旦被任务申请使用后就无法再次回收利用．这对于一些稍微复杂一点应用环境来说是一种比较苛刻的条件，可能会导致产品对内存需求增大，从而引起成本的增加。

④ μC/OS-Ⅲ的内存管理没有提供存取控制、审计跟踪等安全性措施。另外，它对内存空间泄漏也是无能为力的。

4.3 VC 环境下 μC/OS-Ⅲ系统移植

本文将 μC/OS-Ⅲ系统移植到 Intel 80x86 系列 CPU 上。移植过程中主要的步骤如下：

①获取 μC/OS-Ⅲ系统工作时钟

文献[53]中，μC/OS-Ⅲ移植到 X86 的处理器上，需要修改系统的 3 个文件：OS_CPU.H，OS_CPU_A.ASM 和 OS_CPU_C.C。在 DOS 环境下，系统时钟的获取是通过修改 DOS 下的硬件时钟中断来得到时钟的。由于 VC 可以嵌入汇编代码，VC 环境下将 μC/OS-Ⅲ系统的移植到 X86 处理器不需要修改汇编文件 OS_CPU_A.ASM，只修改 OS_CPU.H 和 OS_CPU_C.C 两个文件即可。

在 VC 环境下将 μC/OS-Ⅲ系统移植到 X86 处理器上，需要获取 μC/OS-Ⅲ系统运行时钟，但在 windows 的保护模式下不能像 DOS 下面那么容易获得时钟，在 DOS 环境下，通过函数调用就可以产生中断获取时钟。Windows 环境下获取时钟，修改中断需要修改相应的驱动程序，这使得移植变得困难。不过，在 VC 环境下可以通过使用 Windows 软件定时器产生模拟时钟。Windows 的软件定时器种类很多，下面简要介绍一下：

1）SetTimer()函数

SetTimer 是 windows 系统中的一个 API 函数，它用于创建或设置一个定时器。本文的应用程序是控制台应用程序，要使用 SetTimer()函数需要专门创建一个消息循环处理线程，这会使程序设计变得复杂，同时 SetTimer()函数的定时精度不高，它不适合作为 μC/OS-Ⅲ系统的定时器。

2）QueryPerformanceFrequency()和 QueryPerformanceCounter()函数

这两个函数定时精度很高，定时误差不超过 1 微秒[52]。但这两个函数主要用于时间计算，由于没有设置回调函数机制，如果用这两个函数来进行定时，需要创建一个新的线程，实现定时函数回调机制，在线程中计算时间进行时间定时。它的实现比 timeSetEvent()函数要更复杂，它也不适合作为 μC/OS-Ⅲ系统的定时器。

3）timeSetEvent()函数

该函数的定时精度较高，可以达到毫秒级，且应用比较简单，无需使用消息循环，应用程序可以在精确时间间隔内完成对事件的调用。进行事件处理时，通过调用该函数，在 LpTimeProc 回调函数中定义需要周期性执行的任务。调用这个函数后会新建一个线程，定时时间到后，在新的线程中调用回调函数。这种调用方式和外部中断调用非常类似，由于 timeSetEvent()函数的定时精度较高，应用比较简单，且能够完成类似中断的调用，因此，本文选择该函数来模拟产生时钟中断。

②模拟时钟中断的产生

在 VC 环境下，本文通过 timeSetEvent()函数来模拟产生 μC/OS-Ⅲ的时钟中断。但由于主线程和 timeSetEvent()调用回调函数是同时进行的，不能实现 μC/OS-Ⅲ的时钟中断。为了实现中断，必须使回调函数执行时，中止主线程，Windows 系统中通过调用 SuspendThread 函数挂起线程。回调函数执行完毕后再调用

ResumeThread 函数继续主线程的运行。挂起和继续执行操作在 μC/OS-Ⅲ系统中的时钟中断处理函数里执行。具体的执行过程如图 4.3 所示。

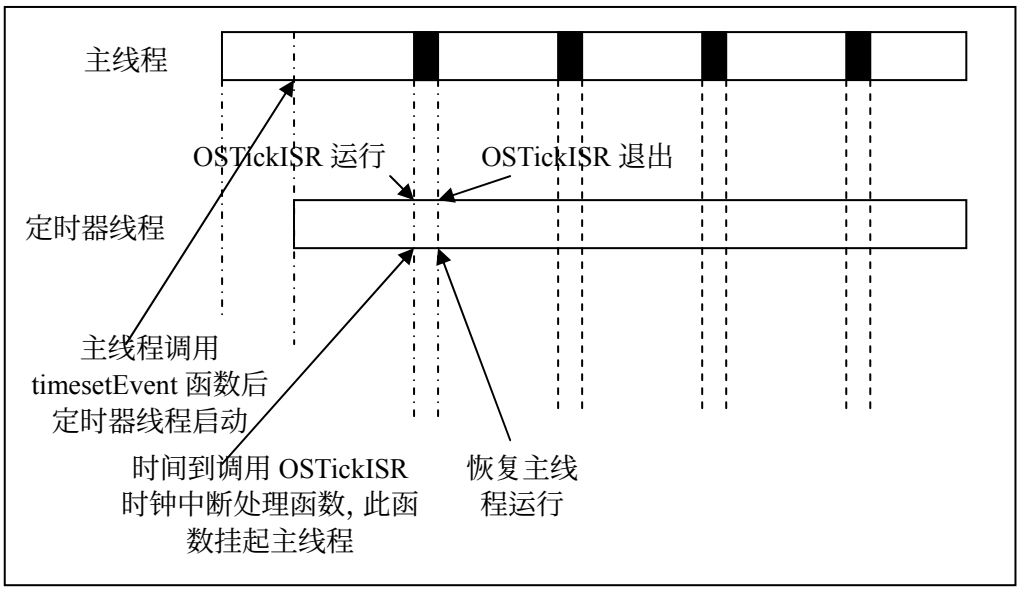

图 4.3 中断模拟过程

Fig.4.3 Interrupt simulation process

首先主线程开始运行，调用软件定时器 timeSetEvent()函数，新建一个定时器线程，定时时间到后，调用系统的中断处理函数，中断处理函数 OSTickISR 会挂起主线程，然后进行任务调度。

函数的执行程序为：

```
void main (void)
{
    //在初始化 μC/OS-Ⅲ系统前对 VC 环境进行初始化
      VCInit();
      //初始化 μC/OS-Ⅲ系统
      OSInit();
      //创建任务
      //开始多任务调度
      OSStart();
}
```

在上面的 main 函数中，在 μC/OS-Ⅲ系统的初始化 OSInit()前，添加 VC 环境初始化函数 VCInit(), 该初始化函数的功能是获取主线程的句柄, 初始化 VC 环境, 并设置任务切换标志位等功能。

```
// mainhandle：全局变量，主线程句柄
HANDLE mainhandle;
// Context：全局变量，主线程任务切换
CONTEXT Context;
```

VC 环境的初始化程序为：

```
void VCInit(void)
{
    HANDLE CurrentThreadHandle,CurrentCourseHandle;
    Context.ContextFlags = CONTEXT_CONTROL;
    //获取当前进程句柄
    CurrentThreadHandle = GetCurrentProcess();
    //获取当前线程伪句柄
    CurrentCourseHandle = GetCurrentThread();
    //通过伪句柄转换，获取线程真句柄
    DuplicateHandle(CurrentThreadHandle,          CurrentCourseHandle,
        CurrentThreadHandle, &mainhandle, 0, TRUE, 2);
}
```

主线程调用软件定时器 timeSetEvent()函数后，在回调函数线程中调用μC/OS-Ⅲ系统的中断处理函数 OSTickISR，OSTickISR 会被周期性调用。中断处理函数 ISR 程序为：

```
void CALLBACK OSTickISR(uint , uint , ulong , ulong , ulong )
{
    //判断中断是否可用
    if(!FlagEn)
    {
        //若中断被屏蔽则返回
      return;
    }
    //中断主线程运行
    SuspendThread(mainhandle);
    //得到主线程句柄
    GetThreadContext(mainhandle, &Context);
    OSIntNesting++;
    if (OSIntNesting == 1)
```

```
    {
        //保存当前 esp 值
        OSTCBCur->OSTCBStkPtr = (OS_STK *)Context.Esp;
    }
    OSTimeTick();
    OSIntExit();
    //模拟中断返回，继续执行主线程
    ResumeThread(mainhandle);
}
```

在中断处理程序中首先判断判断中断状态，它通过一个全局变量进行判断。若中断被屏蔽，则不挂起主线程，只有中断没有被屏蔽，才可挂起主线程，完成进一步的工作。

③任务切换

μC/OS-Ⅲ系统的任务切换，实际完成的是任务的上下文切换。

在其他处理器可以很容易判断任务的上下文，一般为处理器上的寄存器。在 VC 环境下，上下文环境是不带浮点运算的，因此任务的上下文和在 80x86 上移植的上下文很相似，不同的是段寄存器不用保存，它在同一个线程中是不会变的[48]。由于 80X86 没有 PUSHA 指令，移植的时候要用几条 PUSH 指令来代替。任务切换压栈时如图 4.4 所示

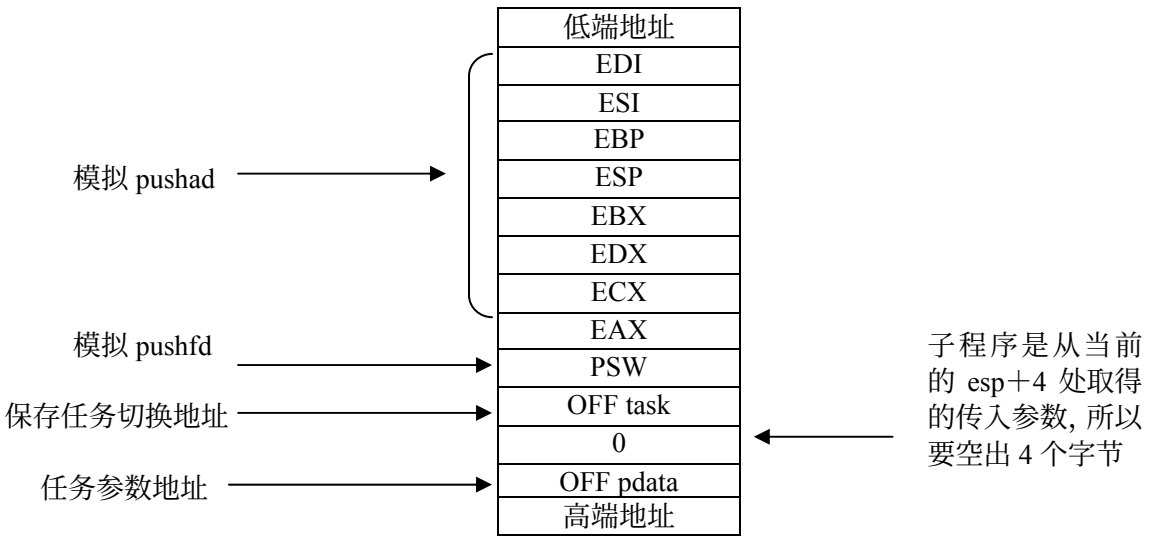

图 4.4 任务切换压栈后状态图

Fig.4.4 The task switch push state diagram

任务切换的函数为 OSCtxSw，它主要是用汇编完成的，该函数主要完成在 VC

环境下任务切换。具体程序如下：

```
void OSCtxSw(void)
{
    _asm{
    lea  eax, nextstart
    push eax
    pushfd;压栈标志寄存器
    pushad;压栈 EAX–EDI
    mov ebx, [OSTCBCur]
    mov [ebx], esp;保存堆栈入口的地址
    }
    OSTaskSwHook();
    OSTCBCur = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    _asm{
        mov ebx, [OSTCBCur];
        mov esp, [ebx];
        popad;
        popfd;
        ret  ;
        }
        nextstart://任务切换回来的运行地址
        return;
}
```

4.4 TLSF 算法在 μC/OS-Ⅲ系统上的实现

4.4.1 添加相应头文件定义

对 TLSF(Two-level Segregated Fit)的移植十分简单，需要我们需要的地方只是与 TLSF 锁相关函数。包括锁的创建、申请、释放、消耗等功能，相应的函数如下[46]：

```
/*加入头文件，需要使用互斥量来实现锁的功能*/
#include "ucos_iii.h"
INT8U perr;
#define TLSF_MLOCK_T                    OS_EVENT *
```

```
#define TLSF_CREATE_LOCK(l)          (*(l)) = OSMutexCreate (4, &perr)
#define TLSF_DESTROY_LOCK(l)         OSMutexDel (*(l), OS_DEL_ALWAYS, &perr)
#define TLSF_ACQUIRE_LOCK(l)         OSMutexPend (*(l), 0, &perr)
#define TLSF_RELEASE_LOCK(l)         OSMutexPost (*(l))
```

以下是对 TLSF 算法的配置：

```
#define MAX_FLI    (15)
#define MAX_LOG2_SLI     (3)
#define MAX_SLI    (1 << MAX_LOG2_SLI)
#define FLI_OFFSET     (4)
#define SMALL_BLOCK      (1<<(FLI_OFFSET+1))
#define REAL_FLI   (MAX_FLI - FLI_OFFSET)
```

4.4.2 删除 μC/OS-Ⅲ系统内存管理相关数据结构及函数

找到 μC/OS-Ⅲ系统里面的 os_mem.c 文件, 删除系统自带的内存相关的数据结构及内存管理函数。需要删除的数据结构有 OS_MEM，需要删除的内存管理函数如下：

1) 内存分区建立函数：OSMemCreate()
2) 内存块分配函数：OSMemGet()
3) 内存释放函数：OSMemPut()
4) 查询内存分区状态函数：OSMemQuery()

4.4.3 添加 TLSF 算法数据结构

在 os_mem.c 文件里添加相关的数据结构及 TLSF 算法相关的内存管理函数。需要添加的数据结构有：

```
/*用于空闲内存块的链接*/
typedef struct free_ptr_struct
{
    struct bhdr_struct *prev;
    struct bhdr_struct *next;
} free_ptr_t;
/*空闲链表头*/
typedef struct bhdr_struct
{
    struct bhdr_struct *prev_hdr;
    size_t size;
```

```
        union
        {
            struct free_ptr_struct free_ptr;
            u8_t buffer[1];
        } ptr;
    } bhdr_t;
    /*用于连接多个内存区，内存区（池）链接结构体*/
    typedef struct area_info_struct
    {
        /*指向末内存块，内存区（池）最后一块，低 2 字*/
        bhdr_t *end;
        /*指向下一个内存区（池），新增的内存*/
        struct area_info_struct *next;
    } area_info_t;

    /* TLSF 动态内存分配算法结构体  */
    typedef struct TLSF_struct
    {
        /* TLSF 算法标识符/标志符*/
        u32_t tlsf_signature;
        /*  用于链接所有的内存区（池），把不连续的内存区链接起来  */
        area_info_t *area_head;
        /*  位图*/
        u32_t fl_bitmap;
        /* the second-level bitmap */
        u32_t sl_bitmap[REAL_FLI];
        /*  空闲内存块链表，用于存储相应空闲内存块*/
        bhdr_t *matrix[REAL_FLI][MAX_SLI];
    } tlsf_t;
```

4.4.4 添加 TLSF 算法相关函数

需要添加的 TLSF 算法主要函数如下：

1) 查找合适的内存块函数：FIND_SUITABLE_BLOCK(tlsf_t * _tlsf, int *_fl, int *_sl)
2) 建立新的内存分区函数：get_new_area(size_t * size)

3) 内存池初始化函数：init_memory_pool()

4) 内存池销毁函数：destroy_memory_pool(void *mem_pool)

5) tlsf 内存分配函数：tlsf_malloc(size_t size)

6) tlsf 内存释放函数：tlsf_free(void *ptr)

7) tlsf 内存重分配函数：tlsf_realloc(void *ptr, size_t size)

4.5 ERMM 在 μC/OS-Ⅲ系统上的实现

ERMM(Embedded Real-time Memory Manager) 动态内存管理器是在 TLSF(Two-level Segregated Fit)算法的基础上，在小内存分配上加入了二级位图算法，并在 TLSF 算法上进行了改进，改变了 TLSF 的数据结构，以及内存分配时的分割策略、内存释放时的合并策略。4.2 章中详细讲述了 TLSF 算法在 μC/OS-Ⅲ系统上的实现，因此只需要在此基础上修改相应的数据结构，及内存分配和释放函数，即可完成 ERMM 在 μC/OS-Ⅲ系统上的实现。

4.5.1 添加数据结构

1) 小内存分配区数据结构：

```
u32_t    OSMemGrp;
u32_t    OSMemTbl[OS_RDY_TBL_SIZE];
```

OSMemGrp 中的每一位代表相应组中是否有空闲内存块，OSMemTbl[n]则记录第 n 组内空闲内存块分布情况。

2) 大内存分配区数据结构：

```
static struct ERMM_struct
{
    u32_t    OS_FLBitmap;
    u32_t    OS_SLBitmap[REAL_FLB];
    bhdr_t *matrix[REAL_FLB][MAX_SLB];
} ERMM_t;
```

OS_FLBitmap 及 OS_SLBitmap[REAL_FLB]描述二级位图, 每个内存块对应位图中的 1 位，位为 1 表示有空闲内存块，为 0 则表示没有，matrix[REAL_FLB][MAX_SLB]存储空闲链表头指针。

3) 空闲内存块表头

而结构体 struct free_ptr_struct 用来链接同一链表中的各个空闲内存块, 当内存块被使用时，此结构体所占空间用于分配内存块中。结构如下：

```
typedef struct free_ptr_struct {
    /*链接逻辑上的前一个内存块*/
```

```
    struct bhdr_struct *prev;
    /*链接逻辑上的后一个内存块*/
    struct bhdr_struct *next;
} free_ptr_t;
```

4.5.2 修改内存分配和释放函数

将 TLSF 算法中的添加小内存块内存序号查找，及大内存一级二级索引查找

```
void MAPPING_SEARCH(size_t * _r, int *_fl, int *_sl, int *_num)
{
    int _t, _x, _y;
    //所需内存块小于 128B 时，采用位示图算法，返回空闲内存块的序号
    if (*_r < 128)
    {
       _y = ls_bit(OSMemGrp);
        _x = ls_bit(OSRdyTbl[_y]);
        *_num = (_y << 5) + _x;
        *_sl = 0;
        *_fl = 0;
    }
else
 {
          _t = (1 << (ms_bit(*_r) - MAX_LOG2_SLI)) - 1;
          *_r = *_r + _t;
           *_fl = ms_bit(*_r);
          *_sl = (*_r >> (*_fl - MAX_LOG2_SLI)) - MAX_SLI;
          *_fl -= FLI_OFFSET;
          *_r &= ~_t;
        *_num = 0;
    }
}
```

修改 TLSF 的 tlsf_malloc(size_t size)内存分配函数，将内存分配分为小内存分配和大内存分配，首先判断请求内存的大小，若 size 为小内存，则通过查找小内存块的内存序号，继而查找其对应的地址，返回地址完成分配。若为大内存，用 MAPPING_SEARCH 函数查找一级二级索引，获取待分配的空闲内存块，然后比较待分配的空闲内存块 size_t 和请求内存大小 size，如果 size_t 与 size 的差值大于

切割门限，则对空闲内存块进行精确切割，否则不进行切割，返回空闲内存块地址完成分配。

修改 TLSF 的 tlsf_free(void *ptr)内存释放函数。首先计算内存碎片率，比较内存碎片率 F 和合并阈值 L 之间的大小。当 F 小于 L 时暂不合并，当 F 大于 L 时，查找待释放内存块是否有相邻空闲内存块，若有则立即合并。

4.6 本章小结

本章首先将 μC/OS-Ⅲ系统在 VC 环境下移植到 X86 处理器上，然后修改 μC/OS-Ⅲ系统内存分配相关文件, 实现 TLSF 算法, 最后在此基础上修改 μC/OS-Ⅲ系统的部分头文件，添加了相应的数据结构，修改 TLSF 算法的内存分配函数及内存释放函数，完成 ERMM 在 μC/OS-Ⅲ系统上的实现。

5 实验结果与分析

5.1 实验设计

文献[53]表明, 在平均指令周期、内存碎片率等方面, TLSF(Two-level Segregated Fit)算法比 First-Fit 算法、Half-Fit 算法、伙伴算法要好，因此选取 TLSF 与 ERMM(Embedded Real-time Memory Manager)进行比较。

嵌入式实时系统动态内存分配算法主要的性能指标有三个方面：(1)实时性。算法要控制最坏情况下运行时间的有界性，以确保系统能够及时响应，这在嵌入式系统中更加重要。(2)碎片率。包括内部碎片和外部碎片，碎片会造成内存的浪费。好的内存分配算法，必须严格控制碎片，提高内存的利用率。(3)可靠性。本文对比 ERMM 和 TLSF 算法的内存碎片率、内存分配和释放时间。

由于 TLSF 算法没有对内存进行相应的保护措施, 本文从实时性、内存碎片率、方面对 ERMM 和 TLSF 算法进行比较。在可靠性方面，对 ERMM 进行内存越界检测实验。

① 实时性对比实验

通过对比 ERMM 和 TLSF 算法的动态内存分配和动态内存释放时间来判断实时性。实验将用户请求的内存大小分为 8 个区间（单位是字节），分别为[0,128) , [128,256) , [256,1024) , [1024,4096), [4096,16384), [16384,65536), [65536,262144) 和[262144, 1048576) , 测试程序在每个区间范围内产生随机值, 并频繁地分配和释放随机值大小的内存。每组测试 100 次，取平均值。调用 μC/OS-Ⅲ系统里计时函数，测量内存调用及内存释放时间，对比 ERMM 在和 TLSF 算法实时性性能。

② 内存碎片率对比实验

在实验①中划分的区间范围内进行 100 次的内存分配和释放请求，求取实际使用的最大内存和分配器所使用的最大内存的比值，计算并比较 TLSF 算法和 ERMM 的内存碎片率。

③ ERMM 内存安全性检测实验

分配 1KB 大小的内存，在应用程序中对该内存进行操作，改写 1KB 大小内存块以外的内存值，然后对该内存进行释放。在本实验中，对内存有越界访问，并且改写了越界访问区的内存值，若程序中断运行，并抛出异常，表明 ERMM 具有内存保护功能。

本文在 VC 环境下将 μC/OS-Ⅲ系统移植到 X86 处理器上，使用 VC++ 6.0 开发环境进行系统的运行调试工作，实验的测试的环境为：

1）处理器：Pentium(R) Dual-Core E6500(主频 2.93GHz)

2）内存大小：2G

3）嵌入式实时操作系统：μC/OS-Ⅲ系统

4）TLSF 算法：TLSF2.4.6 版本

5）调试环境：VC++ 6.0 开发环境

调试环境的界面如图 5.1 所示。

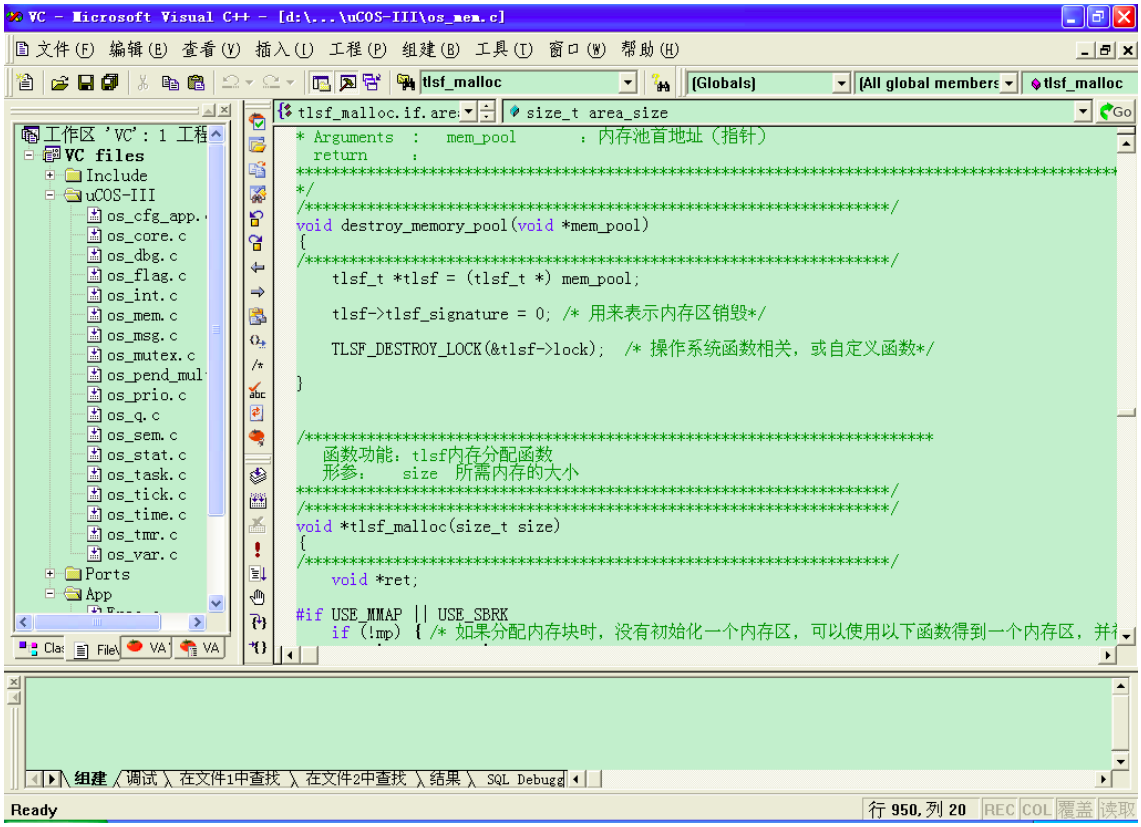

图 5.1 VC++ 6.0 集成开发环境界面

Fig.5.1 VC++ 6.0 integrated development environment interface

ERMM 内存管理器的相关参数定义：

1）二级位图最小内存块大小：128B

2）ERMM 一级索引起始值：8

3）内存分配的切割门限值：128B

4）内存释放合并阈值 L：0.09

5.2 实验结果分析

① 时间性能

1）内存分配和释放时间

动态内存的分配和释放时间的测试结果如表 5.1 所示，图 5.2 及图 5.3 将表 1 的实验结果以柱状图的形式表示。

表 5.1 分配和释放时间比较表(us)

Table 5.1 Comparison of allocation and release time table(us)

操作	算法	区间 1	区间 2	区间 3	区间 4
内存分配	TLSF	5.856	6.994	9.354	8.454
	ERMM	1.036	5.882	7.256	7.921
内存释放	TLSF	4.365	4.886	7.652	7.112
	ERMM	0.986	3.127	4.365	4.653
操作	算法	区间 5	区间 6	区间 7	区间 8
内存分配	TLSF	10.237	12.365	11.564	14.153
	ERMM	8.945	11.323	11.238	14.862
内存释放	TLSF	9.002	10.386	10.986	12.364
	ERMM	6.354	10.167	11.237	12.743

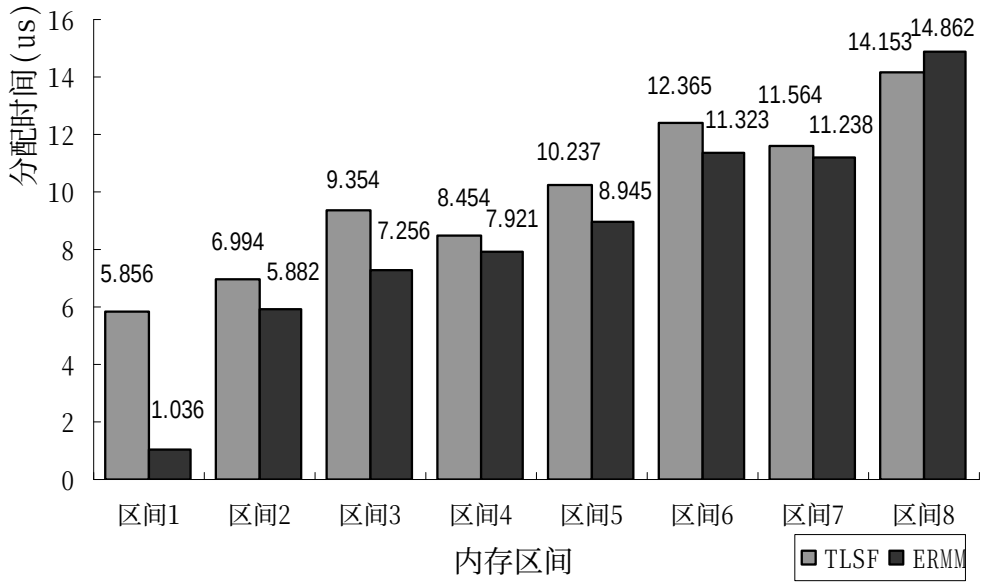

图 5.2 内存分配时间比较图

Fig.5.2 Memory allocation time comparing figure

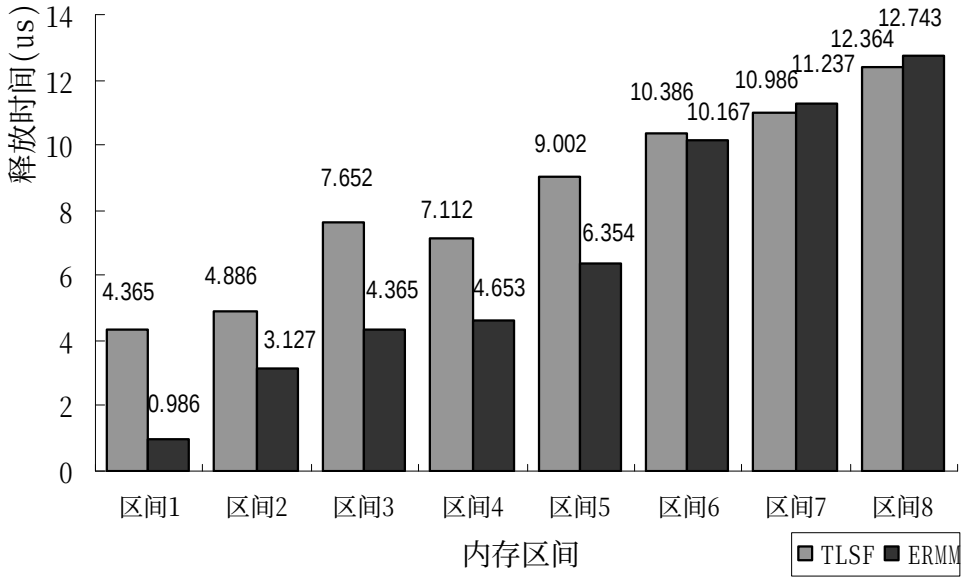

图 5.3 内存释放时间比较图

Fig.5.3 Memory release time comparing figure

由图 5.1 及图 5.2 可以看出，ERMM 的内存分配和释放时间整体上要比 TLSF 少，具有较好的实时性。ERMM 小内存分配和释放速度比 TLSF 更快，这是由于小内存分配和释放采用二级位图算法分配和释放固定大小的内存块，指令周期数比 TLSF 要少。

ERMM 的内存分配时间一直都是小于 TLSF 的，但内存释放时间随着分配内存的增大与 TLSF 的差距也越来越小, 甚至在第七组和第八组内存释放时间超过了 TLSF。这是由于第七组时，内存碎片率(见图 5.3）超过合并阈值，ERMM 开始在内存释放时对相邻的空闲内存块进行合并。

② 内存碎片

为了测量内存碎片，记录了每次内存分配和释放操作时的实际使用内存和已分配内存，并根据这两个值计算内存碎片率。TLSF 与 ERMM 碎片率见表 5.2，碎片率的直方图如图 5.4 所示。

表 5.2 碎片率比较表(%)

Table 5.2 The fragmentation rate comparison table(%)

算法	区间 1	区间 2	区间 3	区间 4
TLSF	5.9	6.3	6.7	8.3
ERMM	0.0	5.3	5.9	7.0
算法	区间 5	区间 6	区间 7	区间 8
TLSF	9.4	9.7	11.0	14.3
ERMM	8.1	8.3	9.2	9.8

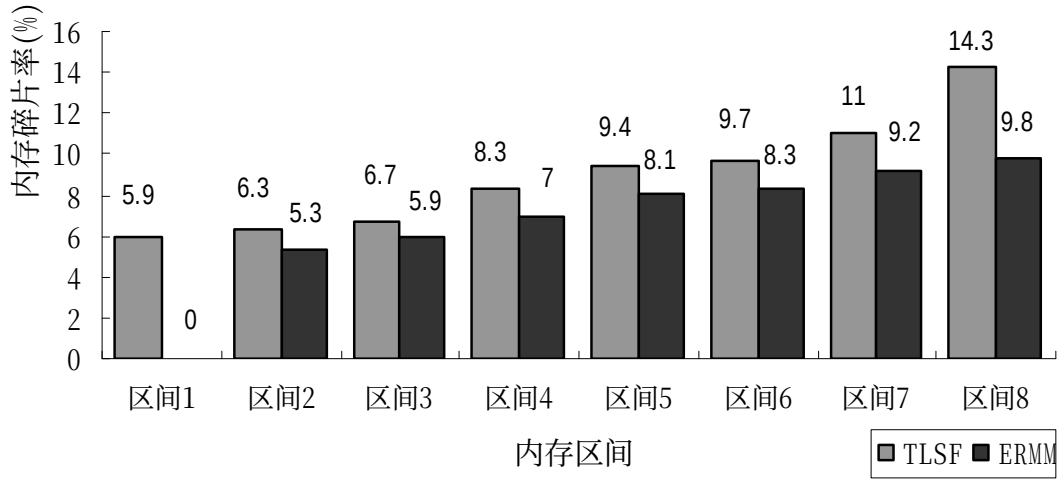

图 5.4 内存碎片率比较图

Fig.5.4 Memory fragmentation rate comparing figure

由图 5.3 可以看出，ERMM 的内存碎片率比 TLSF 的要低。分配小内存时，ERMM 采用的是固定尺寸的动态内存管理方式，这有效避免了外部碎片的产生。虽然有内部碎片，但固定大小内存块的分配方式，提高了内存的实时性，而且在每次释放内存的时候可以回收，极大提高了内存利用率。没有外部碎片。分配大内存时，由于本文在内存分配时采用了“精确切割”策略，消除了内部碎片；另外，ERMM 对 TLSF 的二级索引结构进行了改进，降低了外部碎片。

影响内存碎片率的另外一个因素是内存释放阈值的选取，本文中选取内存释放合并阈值 L 为 0.09，它的值是根据经验值确定的。若 L 的值过大，会产生过多的内存碎片，但这会提高内存释放的实时性；若 L 的值过小，虽然会降低内存碎片，但是会降低内存释放的实时性。

③ ERMM 内存安全性测试

ERMM 内存安全性测试，主要针对内存的越界访问，当内存的内容被越界改写后，该内存块释放时，ERMM 会判断内存块越界访问区间值是否为 0XFEEEEEEE，若该值已经发生改变，则表明发生了内存越界访问操作。

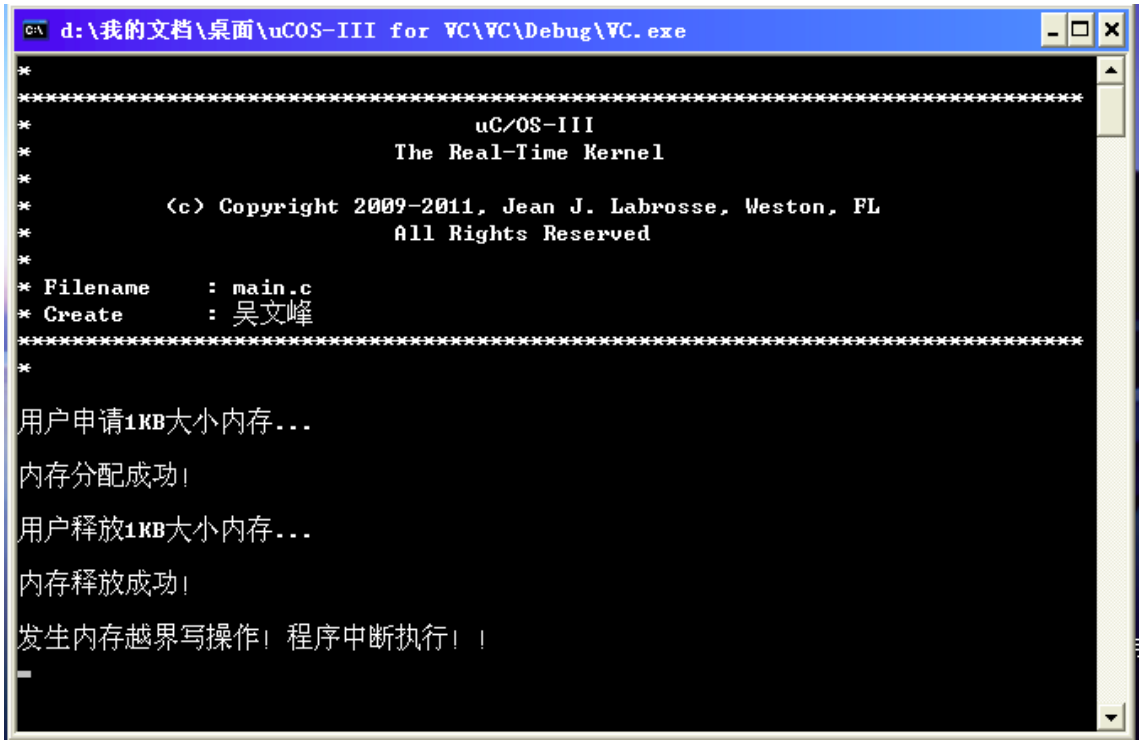

图 5.5 ERMM 内存安全性测试图

Fig.5.5 ERMM memory safety test chart

申请分配 1KB 大小的内存，在应用程序中对该内存进行越界访问操作，改写 1KB 大小内存块以外的内存值，然后对该内存进行释放。如图 5.5 所示，用户申请 1KB 大小内存成功，在越界访问后释放内存，内存释放成功，但因为发生了内存越界操作，ERMM 提示发生内存越界操作，并结束程序。这表明 ERMM 具有内存保护功能。

通过对 ERMM 和 TLSF 在内存碎片率、实时性实验对比，结果表明 ERMM 管理器相较于 TLSF 有更好的实时性和较低的内存碎片率。通过内存安全性测试实验，证明 ERMM 动态内存管理器具有较好的安全性和可靠性，能够检测出内存越界访问操作，并给提示，有效提高了嵌入式实时系统内存管理的可靠性。实验表明，ERMM 能够满足嵌入式实时系统的内存管理的实时性、内存碎片率及可靠性要求，有效提高了嵌入式实时系统的性能。

5.3 本章小结

本章在 μC/OS-Ⅲ系统上对 ERMM 及 TLSF 算法通过实验进行性能测试，并将这两种算法进行比较，给出实验结果，并对算法在分配响应时间、释放响应时间还有内碎片率进行分析。

6 总结与展望

本文在二级位图算法及 TLSF 算法的基础上提出了一个新的动态内存分配管理器，对这两种算法加以提高内存管理的实时性，降低内存碎片率。提出了新的数据结构，通过以空间换时间查找空闲内存块序号的方法提高了小内存的分配效率，通过设定“切割门限”、“合并阈值”的方法加快了大内存的分配和释放时间并降低了内存碎片。本文完成的具体工作如下：

① 在分析嵌入式系统特点及嵌入式实时系统的基础上，介绍典型的动态内存管理算法，并针对实时性及碎片率对不同算法进行分析，同时介绍了不同实时操作系统的动态内存分配方案，最后分析了实时系统动态内存方案存在的问题。根据嵌入式实时系统的对实时性及碎片率的要求，提出在 TLSF 算法基础上加以改进的内存管理方案。

② 针对 TLSF 算法在内存碎片率、实时性及安全性方面的不足，提出了 ERMM(Embedded Real-time Memory Manager)动态内存分配管理器。它在 TLSF 算法上加以改进，具体的改进包括改进数据结构，不同大小内存分配采用不同的分配策略，添加边界标示技术，内存分配采用“精确切割”，内存释放采用“延时合并”策略。通过这些改进提高了内存分配效率、降低了内存碎片率、减小了内存分配和释放时间，增加了内存分配的安全性和可靠性。

③ 在 VC 环境下将 μC/OS-Ⅲ系统移植到 Intel 奔腾系列处理器上，通过修改 μC/OS-Ⅲ系统内存管理的相关文件在 μC/OS-Ⅲ系统上实现 TLSF 算法，并在此基础上，修改 TLSF 算法的数据结构，及内存分配及释放函数，在 μC/OS-Ⅲ系统上实现 ERMM 内存管理器。

④ 测试 ERMM 嵌入式实时系统动态内存分配管理器在实时性、内存碎片率及内存安全性方面的性能。对算法在分配响应时间、释放响应时间及内碎片率方面和 TLSF 算法进行比较。实验结果验证了 ERMM 在内存实时性、内存碎片率及内存安全性方面比 TLSF 算法有更好的性能。

本文提出的新的嵌入式实时系统动态内存分配管理器，通过和 TLSF 实验验证，有更好的实时性和较低的内存碎片率，增加了内存安全性防护，有效地满足嵌入式实时系统对内存管理的要求。

进一步的工作如下：

① 进一步完善 ERMM 动态内存分配管理器，提高其稳定性；

② 在 ERMM 动态内存分配管理器加入内存泄露检测，提高内存管理的安全性。

③ 将 ERMM 动态内存分配管理器在其他实时系统上进行移植，并测试其性能。

致　　谢

本文是在导师孙棣华教授及李斌副教授的悉心指导下完成的，字里行间透漏着导师一丝不苟，治学严谨的科研态度，在此向我的导师致以最衷心的感谢和崇高的敬意。在我三年研究生生活学习中，导师以身作则，兢兢业业的工作态度，都深深的感染和激励着我，从恩师身上，我不仅学到了宽广、精深的专业知识，最重要的是学到了做人做事的道理，这些必将成为我以后工作和生活中的宝贵财富并使我受益终生！

衷心感谢我的爸爸、妈妈，还有弟弟，是你们无私的爱和关怀，是你们默默地付出，才成就了今天的我，你们是我的动力，也是我的奋斗目标。

感谢实验室的郑林江老师、赵敏老师、廖孝勇老师对我在学业和生活上的指导与关心，与你们亦师亦友的关系让我的研究生生活丰富而精彩！

感谢已经毕业的龚康师兄、武保全师兄、张路师兄，是你们两年多的热心地帮助，让我慢慢成长，让我摆脱了学习上以及生活上的迷茫，并渐渐找到自己的方向。祝你们工作顺利、事业有成。

感谢实验室的各位兄弟姐妹们，与你们相识并相知是我今生重要的经历，与你们朝夕相处、相互学习的快乐时光，将令我终身难忘！

在此，我向所有关心和帮助我的亲人、老师、同学和朋友们表示诚挚的感谢！

最后，衷心感谢在百忙之中评阅论文的各位专家、教授！

吴文峰

二 O 一三年四月 于重庆

参 考 文 献

[1] 张义磊, 于涛, 安吉宇等. 三星 S3C2410 在嵌入式工业控制系统中的应用[J]. 长春理工大学学报, 2004,27(3):92-95.

[2] 张飞. 实时嵌入式操作系统动态内存管理研究[D]. 中国科学技术大学硕士论文, 2011.

[3] 丁南菁. 嵌入式系统的内存管理设计研究[D]. 北京工业大学硕士学位论文, 2008.

[4] 张波. 基于嵌入式技术的移动终端设计[D]. 上海海事大学硕士学位论文, 2007.

[5] 陆小双, 帅建梅, 吴庆响. 一种新的面向对象程序的内存管理器[J]. 计算机工程, 2012, 38(9):21-23.

[6] 柴继国. 嵌入式系统内存管理的研究与实现[D]. 电子科技大学硕士学位论文, 2006.

[7] 李志军. 面向嵌入式实时系统的动态内存管理方法研究[D]. 重庆大学硕士学位论文, 2007.

[8] Wilson Paul R., Johnstone Mark S., Neely Michael,.et al. Dynamic Storage Allocation: A Survey and Critical Review[J]. International Workshop on Memory Management,1995,7(3):1-78.

[9] Ogasawara T. An algorithm with constant execution time for dynamic storage allocation[C]//Real-Time Computing Systems and Applications, 1995. Proceedings., Second International Workshop on. IEEE, 1995: 21-25.

[10] 闫广明. 嵌入式系统内存空间域隔离技术的研究与实现[D]. 哈尔滨工程大学硕士学位论文, 2011.

[11] 李法龙, 吴刚, 陈章龙. 位图在嵌入式系统内存管理中的应用[J]. 计算机工程与设计, 2005,26(4):1020-1023.

[12] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems[C]. Proceedings of the 16th Euromicro Conference on Real-Time Systems, Cantania, Italy:[s.n.],2004:79-88.

[13] Miguel Masmano. A constant-time dynamic storage alloc-ator for real-time systems [J].Real-Time Systems,2008,40(2):149-179.

[14] 阎梦天, 丁志刚, 王挺, 等. 实时操作系统内存分配性能检测[J]. 计算机应用, 2007, 27(11):2838-2840.

[15] 钱华明, 张振旅. VxWorks 内存管理机制的分析与研究[J]. 微计算机信息, 2009, 25(12):115-117.

[16] Sun X H, Wang J L, Chen X. An Improvement of TLSF Algorithm[C]//Real-Time Conference, 2007 15th IEEE-NPSS. IEEE, 2007: 1-5.

[17] Rune Prytz Anderson, Per Skarin. Memory Protection in a Real-Time Operating System [D]. Master Thesis of Lund Institute of Technology, 2004.

[18] 龚康. 面向生产设备管理的嵌入式 SNMP 代理研究与实现[D]. 重庆大学硕士学位论文, 2012.

[19] 嵌入式系统. 百度百科. http://baike.baidu.com.cn/view/6115.htm.

[20] 商斌. Linux 设备驱动开发入门与编程实践[M]. 北京:电子工业出版社, 2009:143-150.

[21] 黄贤英, 王越, 陈媛. 嵌入式实时系统内存管理策略[J]. 计算机工程与设计, 2004, 25(10): 1808-1810.

[22] 吕岭. 嵌入式数据库存储管理软件模型研究[D], 南京航空航天大学硕士学位论文, 2009.

[23] 胡雨翠. 嵌入式实时系统ARTs-OS的动态内存管理研究[D]. 华中科技大学硕士学位论文, 2010.

[24] Abraham Silberschatz, 郑扣根译.操作系统概念第六版[M].高等教育出版社, 2004:143-150.

[25] 李满丽. 复杂嵌入式系统内存管理方案的研究与实现.[D]. 厦门大学硕士学位论文, 2009.

[26] 杨鹏. 一种链式结构在内存管理中的应用[J]. 高等函授学报(自然科学版).2002, 8(5):59-83.

[27] Mark S.Johnstone, PaulR.Wilson. The memory fragmentation Problem:solved?[J]. ACM Sigplan Notiees, 1999, 34(3):26-36.

[28] 李慧璐. 对 VxWorks 中内存管理和定时器模块的改进[D]. 西安电子科技大学硕士学位论文, 2009.

[29] 高超, 韩锐, 倪宏. 嵌入式 Linux 平台内存管理方案 [J]. 小型微型计算机系统, 2011, 32(4):614-618.

[30] Labrosse J J, 邵贝贝译. μC/OS-II—源码公开的实时嵌入式操作系统[M]. 中国电力出版社, 2001.

[31] 梁乘铭, 韩坚华, 夏成文, 等. μC/OS-Ⅱ 中动态内存管理方案的改进与实现[J]. 微计算机信息, 2008, 24(22): 44-46.

[32] 赵跃华, 蔡贵贤, 黄卫菊. 一种嵌入式安全内存管理的设计与实现[J]. 计算机工程与设计, 2006, 27(16):3092-3095.

[33] 曹成. 嵌入式实时操作系统 RT_Thread 原理分析与应用[D]. 山东科技大学硕士学位论文, 2011.

[34] 陈洋, 胡向宇, 杨坚华. VxWorks 下的内存管理[J]. 计算机工程, 2007, 33(8):94-96.

[35] 刘东栋. VxWorks 内存管理机制研究及改进[J]. 科学技术与工程, 2007, 7(6):1218-1210.

[36] 邵明超, 杜强, 李荐民等. RTEMS 交叉编译环境的创建[J]. 中国科学院研究生院学报, 2004, 21(2):254-258.

[37] 董明峰, 谷建华. 嵌入式实时操作系统 eCos 内存分配策略的分析[J]. 微电子学与计算

机, 2004, 21(7): 120-123.

[38] 池元武. 嵌入式实时操作系统动态内存管理优化方案的研究[D]. 上海交通大学硕士学位论文, 2011.

[39] 董明峰, 谷建华. 嵌入式实时操作系统 eCos 内存分配策略的分析[J]. 微电子学与计算机, 2004, 21(7): 120-123.

[40] 顾胜元, 杨丹, 黄海伦. 嵌入式实时动态内存管理机制[J]. 计算机工程, 2009, 35(20):264-266.

[41] 李江雄. 嵌入式 linux 内存管理设计与实现[D]. 华中科技大学硕士论文, 2008.

[42] 韩志刚. 一个内存分配器的设计和实现[D]. 东北大学硕士学位论文, 2008.

[43] L Stephenson C J. Fast Fits: New Methods for Dynamic Storage Allocation[C]//Proc. of the 9th ACM Symposium on Operating Systems Principles. [S.l.]: ACM Press. 1983.

[44] 陆李江, 梅静静, 王申良, 等. TLSF 动态内存分配算法的研究与应用[J]. 单片机与嵌入式系统应用, 2011, 11(11):1-4.

[45] 魏海涛, 姜昱明,李建武, 等. 内存管理机制的高效实现研究[J]. 计算机工程与设计, 2009,30(16):3708-3712.

[46] Masmano, Miguel, Ismael Ripoll, and Alfons Crespo. A comparison of memory allocators for real-time applications. Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. ACM, 2006.

[47] 姜艳, 曾学文, 孙鹏, 等. 实时嵌入式多媒体系统的模糊阈值合并内存管理算法[J]. 西安电子科技大学报:自然科学版, 2012, 39(5):220-227.

[48] 湛辉来, 曾一. X86 体系中保护模式下的内存访问机制[J]. 重庆大学学报(自然科学版), 2002, 25(6):67-70.

[49] 楼永红. 面向嵌入式实时应用的内存管理技术研究[D]. 浙江大学硕士学位论文, 2006.

[50] 何昕. 嵌入式 MINIX 操作系统内存管理的设计 [D].兰州大学硕士学位论文, 2008.

[51] 李楠. μC/OS-II 内存管理方案的改进与实现[D]. 沈阳工业大学硕士学位论文, 2010.

[52] 王景丽, 文佳, 王海南, μC/OS-Ⅱ在 VC 下的移植[J]. 微型电脑应用, 2006, 22(3):33-35.

[53] M. Masmano, I. Ripoll, J. Real, A. Crespo,A. J. Wellings. Implementation of a constant-time dynamic storage allocator. Software: Practice and Experience, 2007, (10):112-115.

附　　录

A. 作者在攻读学位期间发表的论文目录

[1] 孙棣华, 吴文峰, 郑林江, 赵敏, 李斌. 一种嵌入式实时系统动态内存管理器的设计. 小型微型计算机系统(CSCD 核心). 2013. 已录用.