

# 开源 RTOS 内存管理机制分析和改进

何 巍, 何建忠

(上海理工大学光电与计算机工程学院, 上海 200093)

**摘 要:** 针对开源 RTOS(FreeRTOS)内存分配时间不确定及内存利用率低、不能很好支持动态内存分配等不足, 研究 FreeRTOS 的内存管理机制并比较几种典型动态内存管理算法的优缺点。移植修改过的 TLSF 算法对管理机制进行改进, 较小的内存分成固定大小的内存块, 用一级位图索引组织, 较大的内存用二级间隔表组织。实验结果表明该方法能较好地提高内存分配速度和利用率。

**关键词:** 实时操作系统; 开源 RTOS; TLSF 算法

## Analysis and Improvement on FreeRTOS Memory Management Mechanism

HE Wei, HE Jian-zhong

(School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093)

**【Abstract】** The uncertainty of memory allocation time and low memory utilization are shortages of Free Real Time Operating System(FreeRTOS), which also lacks good support to dynamic memory allocation. To solve such problems, this paper analyzes the memory management mechanism of FreeRTOS and compares several typical dynamic memory management algorithms and try to improve its performance through the transplantation of modified Two Level Segregated Fit(TLSF) algorithm. The smaller memory is divided into fixed-size memory blocks and organized by bitmap index, while the larger memory is organized by TLSF. Experimental results show that this method can improve the memory allocation speed and memory utilization.

**【Key words】** Real Time Operating System(RTOS); Free Real Time Operating System(FreeRTOS); Two Level Segregated Fit(TLSF) algorithm

### 1 概述

开源 RTOS(Free Real Time Operating System, FreeRTOS)是个轻量级的嵌入式操作系统,它完全免费,具有源码公开、可移植、可裁减、调度策略灵活的特点,可以方便地移植到各种微处理器上运行。目前最新的版本是 2009 年 3 月更新的 5.2.0。FreeRTOS 体积很小但功能齐全,能满足大多数场合的需要。当然这样的小型嵌入式操作系统也存在许多不足之处。特别是用在一些要求比较高的场合时,需要对 FreeRTOS 进行适当的修改。本文主要分析 FreeRTOS 的内存管理机制,并提出改进的方案。

### 2 FreeRTOS 内存管理机制

FreeRTOS 提供 2 种内存分配策略,用户可以根据需要选择。这 2 种策略实现简单,但是局限性也非常大。第 1 种方法是按照需要的内存大小简单地把一大块内存分割为若干小块,每个小块的大小对应于所需内存的大小。这样做的好处是实现简单,执行时间可严格确定,适用于任务和队列全部创建完毕后再进行内核调度的系统。本质上是静态内存分配。这样做的缺点显而易见,由于内存不能有效释放,因此系统运行时应用程序不能删除任务或队列。第 2 种方法采用单链表分配内存,可实现动态的创建、删除任务或队列。系统根据空闲内存块的大小按从小到大的顺序组织空闲内存链表。当应用程序申请一块内存时,系统根据申请内存的大小按顺序搜索空闲内存链表,找到满足申请内存要求的最小空闲内存块。为了提高内存的使用效率,在空闲内存块比申请内存大的情况下,系统会把此空闲内存块一分为二。一块用

于满足申请内存的要求,另一块作为新的空闲内存块插到链表中。

第 2 种方法的缺点在于:用单链表组织内存块,每次分配内存时都要搜索链表,执行时间不确定。内存块按大小顺序组织,回收内存块时同样须搜索之后才能插入。另外回收内存块时没有合并操作,内存块利用率低,并且随着程序的执行,内存块会越分越小,不能满足大内存块的需要。因此,在应用中通常会限制应用程序申请与释放的内存大小为一系列固定的值。

显然,这 2 种方法的局限性都很大,在需要动态内存分配(Dynamic Storage Allocation, DSA)的场合,它们都不能满足需求。为了在 FreeRTOS 中实现动态内存管理,下面讨论如何在 FreeRTOS 中引入其他内存分配策略。

### 3 典型的动态内存管理算法

在 RTOS 中,动态内存管理算法有很多<sup>[1]</sup>,下面对 3 个典型算法的优缺点进行分析。

#### (1)顺序查找

将空闲块以链表的方式组织在一起,每个空闲块包含边界信息来方便地定位其相邻的内存块。当任务发出内存请求时,可以采用 3 种匹配策略:1)首次适应算法;2)下次适应算法;3)最佳适应算法。缺点是:顺序查找时间不可预测,

**作者简介:** 何 巍(1983 - ),男,硕士研究生,主研方向:嵌入式系统;何建忠,副教授

**收稿日期:** 2009-10-25 **E-mail:** hewei535@gmail.com

不能满足强实时性。FreeRTOS 提供的第 2 种内存分配策略实际就是最佳适应法。

#### (2) 伙伴(Binary-Buddy)算法<sup>[2]</sup>

伙伴算法采用间隔表的数据结构, 将整个内存空间以 2 的指数幂为基本单位划分为不同的空闲块。对第 1 个内存请求, 整个初始内存空间被切割为 2 个伙伴, 其中, 一个以折半的方式切割, 直至得到合适的内存块; 在每次切割过程中, 另一个空闲块被重新插入空闲块列表。只有当互为伙伴的空闲块都空闲时, 才能进行合并操作。在伙伴算法中, 采用位操作可以方便地找到能够满足请求的空闲块。伙伴算法的执行时间很快, 缺点是内部碎片太多, 最多可达 50%。

#### (3) TLSF(Two Level Segregated Fit)算法

TLSF 算法即二级间隔表动态内存分配算法。如文献[3]所述, TLSF 用二级索引将空闲块组织起来。第 1 级索引按 2 的幂次把内存块分成多个区域。第 2 级索引表在第 1 级的内部再次进行线性分割。第 2 索引的每一位都对对应一条空闲内存块链表。链表中每个空闲内存块的大小都属于相应的范围, 但是大小不固定, 并且没有顺序, 如图 1 所示。

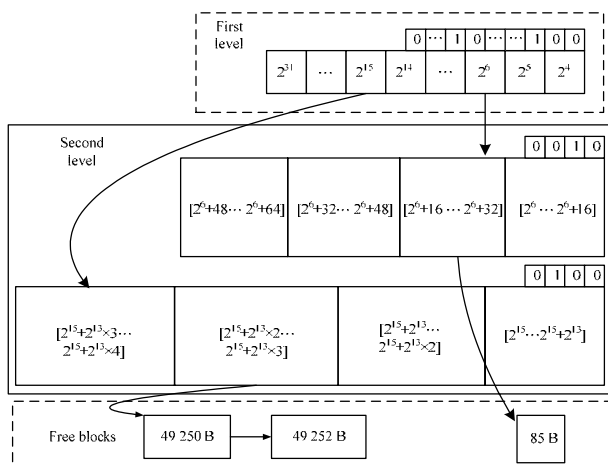


图 1 TLSF 数据结构

当任务请求大小为  $r$  的内存块时, TLSF 算法首先计算一级位图索引  $fli$  和二级位图索引  $sli$ , 判断  $r$  从属的子链  $k$ , 然后从子链  $k$  的下一级子链  $k+1$  取出第 1 个空闲块, 因为下级子链的内存块大小一定可以满足内存请求的需要, TLSF 寻找空闲内存块时用位操作完成, 所以时间复杂度是  $O(1)$ 。同时, TLSF 中采用分割和合并操作, 内存利用率很高。

### 4 FreeRTOS 内存管理的改进和实现

通过前面的分析可以看到, TLSF 用作动态内存分配策略非常适合。为了克服 FreeRTOS 处理动态内存分配的不足, 对 TLSF 算法适当修改并引入 FreeRTOS 中。

在 TLSF 算法中, 不同大小的内存分配请求使用相同的策略。然而, 在分配小块内存时, 产生的内部内存碎片本身就很小, 这时候频繁采用分割策略实际上会降低内存分配效率。文献[4]表明, 在大多数系统中, 小块的内存分配请求远远多于对大块内存的请求, 因此, 对小块内存分配的请求采用特殊策略具有重要意义。

当需要的内存较小时(<4 KB), 使用一级位图索引表, 把内存按 2 的幂次分为固定大小的块, 即 8 Byte, 16 Byte, ..., 4 KB, 一共 10 组。每组空闲内存块用链表组织起来, 位图索引表的每一位表示对应的空闲链表是否为空。它区别于 TLSF 算法的地方在于每个空闲链表的内存块大小固定, 相邻

链表间内存大小差距很小, 因此, 不需要分割和合并操作, 分配和释放内存块不仅速度快而且浪费的内存少, 充分利用了被管理的内存大小限制在 4 KB 以下的特点。

#### 4.1 数据结构

FreeRTOS 中所有的内存分配操作都要在 `ucHeap[configTOTAL_HEAP_SIZE]` 上进行。全局控制块的数据结构如下:

```
static struct xRTOS_HEAP
{
    Unsigned portCHAR ucHeap[ configTOTAL_HEAP_SIZE ];
    unsigned sBitmap;
    //小内存区位图索引
    STRUCT BLOCK_LINK *sMatrix[MAX_SI];
    unsigned flBitmap;
    //一级位图索引
    unsigned slBitmap[MAX_FLI];
    //二级索引
    STRUCT BLOCK_LINK *bMatrix[MAX_FLI][MAX_SLI];
} xHeap;
```

`sMatrix[MAX_SI]` 是指向小块内存区空闲链表头的指针数组。 `bMatrix [MAX_FLI] [MAX_SLI]` 是指向大块内存区空闲链表头的指针数组。

空闲内存块的数据结构设计如下:

```
typedef struct BLOCK_LINK
{
    size_t xBlockSize;
    struct BLOCK_LINK *pxNextFreeBlock;
    struct BLOCK_LINK *pxPrePhysicBlock;
} xBlockLink;
```

大小空闲内存块的结构设计是一致的, 这是为了保证释放内存块时能够正确找到内存块控制结构的开始地址。其中, `pxNextFreeBlock` 指向下一个空闲块, `pxPrePhysicBlock` 指向前一个物理相邻的内存块。算法分配的内存块按 8 字节对齐, 因此, 可以用 `xBlockSize` 最后 3 位存放控制信息。第 0 位表示该块是否空闲, 第 1 位表示该块是否是链表中最后一块, 第 2 位表示该块属于大内存块还是小内存块。

#### 4.2 内存分配

小内存区搜索函数 `pvSmallSearch()` 的实现如下: (1) 实际分配的内存块大小应该加上空闲块的控制结构即 `rSize += BlockSTRUCT_SIZE`。(2) 用位操作确定位图索引 `si = fls(rSize) + 1`; `fls()` 操作表示取 `rSize` 左数第 1 个 1 的位置。

例如请求 74 Byte 的内存块,  $rSize = 74_d = 01001010_b$ , 则  $si = 7$ , 对应的空闲块的大小是 128 Byte。(3) 用位操作寻找首个非空链表, 即 `while (!(xHeap.sBitmap & (0x01 << si)) si++`。如果找到非空空闲链表, 返回链表头指针指向的空闲块的地址 `xHeap.sMatrix[si].pxNextFreeBlock + BlockSTRUCT_SIZE`。(4) 删除空闲链表的头结点, 并更新控制块的信息。如果小内存区找不到合适的内存块, 则用 TLSF 算法到大内存区找。

大内存区搜索函数 `pvBigSearch()` 使用 TLSF 算法实现<sup>[5]</sup>: 首先计算一级位图 `fli`、二级位图索引值 `sli`, 接下来用位操作寻找非空链表。找到之后取出空闲链表的第 1 个空闲块, 计算出应该返回的内存地址。如果分配的内存块 `pxBlock` 的剩余部分大于分割阈值, 即 `(pxBlock->xBlockSize - rSize) > BlockThreshold`, 分割分配的内存块, 创建新的空闲块插入空闲链表, 并更新控制块的信息。

在 FreeRTOS 中用 `pvPortMalloc()` 作为分配内存的接口函

数，它返回可以使用的内存的首地址，设计如下：

```
void *pvPortMalloc(size_t rSize)
{
    vTaskSuspendAll();
    //挂起调度器
    { if(rSize<4 KB)
        {if(!(pvReturn=pvSmallSearch(rSize)))
            pvReturn=pvBigSearch(rSize);}
        else pvReturn =pvBigSearch(rSize);
    }
    xTaskResumeAll();
    //恢复调度
    return pvReturn;
    //返回分配的内存地址
}
```

#### 4.3 内存释放

内存释放时，通过要释放的内存块的 xBlockSize 字段的第 2 位判断该内存块属于大内存还是小内存，然后分别调用 vbFree()和 vsFree()函数处理。

小内存块释放函数 vsFree()处理过程很简单，首先计算位图索引值，然后插入对应空闲链表表头。大内存块释放函数 vbFree()的处理涉及合并操作，首先检查被释放内存物理相邻的内存块是否空闲，如果空闲，执行合并操作，从空闲链表中删去合并后的内存块，然后把它重新插入空闲链表的合适位置，并且更新控制块的信息。

在 FreeRTOS 中用 vPortFree()作为内存释放接口函数，设计如下：

```
void vPortFree( void *pv )
{
    xBlockLink *pxLink=( void * )(( unsigned portCHAR * ) pv-
    blockSTRUCT_SIZE);
    vTaskSuspendAll();
    {
        if(IsBig(pxLink)) vbFree(pxLink);
        else vsFree(pxLink);
    }
    xTaskResumeAll();
}
```

在配置 FreeRTOS 的内存管理策略时，在 heap\_x.c 文件移植修改后的 TLSF 算法代码，编译后就可以使用。

## 5 实验和分析

实验所用程序为一个 DSA 基准测试专用程序 cfrac，该程序分解给定数字的因数，在此过程中向系统申请和释放内存。分别对 FreeRTOS 采用单链表管理内存和修改后的 TLSF 算法管理内存进行实验，并且比较在管理不同大小内存空间的情况下两者的差别。使用平均所需指令数测量内存分配和释放时间。

#### 5.1 时间性能

在分配和回收内存时，单链表方法都要执行搜索操作，所以，随着管理的内存块大小的增加，消耗的时间增加很多。TLSF 方法不需要搜索，所以，比单链表方法快，而且 TLSF 方法时间消耗比较恒定。在释放内存时 TLSF 方法需要合并

空闲内存，所以，要慢一些，但还是能够满足时间要求，见图 2、图 3。

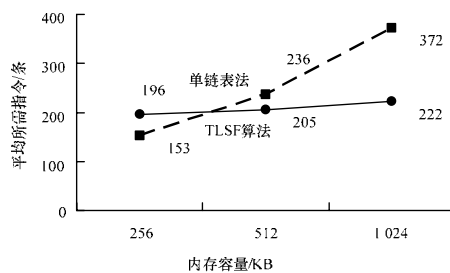


图 2 内存分配时间性能

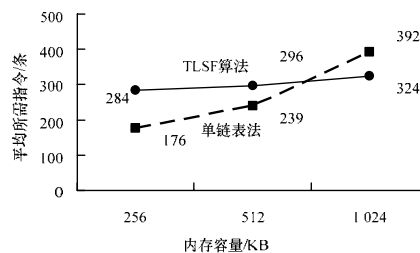


图 3 内存释放时间性能

#### 5.2 内存利用率

在使用单链表方法时，应用程序申请与释放内存的大小只能限定为一系列固定的值，不能处理随机大小的内存请求。而 TLSF 方法下，对申请的内存大小没有限制。在动态请求内存块的实验中，任务动态的申请大小在[8 Byte, 64 KB]之间的内存块，系统运行稳定，有效地提高了内存利用率。

## 6 结束语

本文通过在 FreeRTOS 中引入改进的 TLSF 算法，改善了原有系统的性能，使它能较好地应用于需要动态内存分配的场所。实验结果证明其达到了预期的目的，对相关研究工作有较好的参考价值。

#### 参考文献

- [1] 常铁原，刘娜，陈文军.  $\mu$ C/OSII 内存管理技术的研究[J]. 计算机工程, 2007, 33(9): 82-83.
- [2] Knuth D E. The Art of Computer Programming, Volume 1: Fundamental Algorithms[M]. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1998.
- [3] Masmano M, Ripoll I, Crespo A. TLSF: A New Dynamic Memory Allocator for Real-time Systems[C]//Proceedings of the 6th Euromicro Conference on Real-time Systems. Catania, Italy: [s. n.], 2004: 79-88.
- [4] Johnstone M S, Wilson P R. The Memory Fragmentation Problem: Solved?[C]//Proceedings of the International Symposium on Memory Management. Columbia, Canada: [s. n.], 1998: 26-36.
- [5] Masmano M, Ripoll I, Real J. Implementation of a Constant-time Dynamic Storage Allocator[J]. Software: Practice and Experience, 2007, 38(10): 995-1026.

编辑 张正兴