嵌入式系统的自适应内存管理方案的算法实现

张 为

(广东工业大学 机械电子工程学院,广东 广州 510006)

摘 要:鉴于嵌入式系统的实时性、可靠性对内存管理提出的要求,在最常用的几种内存分配算法的基础上提出了一种新的适用于嵌入系统的内存管理算法——自适应内存分配算法,并重点介绍了减小内存碎片,提高内存利用率,同时提出了对新构想的算法实现。

关键词:嵌入式系统;内存管理;内存碎片;内存分配;算法

中图分类号:TP301.6 **文献标识码**:A **文章编号**:1673-9833(2008)06-0090-04

Arithmetic Realization on Adaptive Memory Allocating Method for Embedded System

Zhang Wei

(School of Mechanical and Electrical Engineering, Guangdong University of Technology, Guangzhou 510006, China)

Abstract: In view of real-time characteristics, the reliability requirement, a new memory of their own managing arithmetic-memory adaptive arithmetic is put forward based on some common memory managing methods. Then it gives the emphasis on how to reduce fragment memory and improve using rate, and also presents new idea to arithmetic realization.

Key words: embedded system; memory management; memory fragment; arithmetic

内存作为嵌入式系统最重要的系统资源,其分配和释放策略对系统的运行效率起着至关重要的作用。系统内核和所有进程通过共享有限的物理内存来运行,一个系统的高效性与稳定性往往取决于它内存管理机制。因此,一个高效的内存管理系统不仅要能够有效地管理系统内存,减少频繁分配和回收内存而导致的内存碎片,还要尽力提高分配和回收的速度来提高系统的运行效率。此外,内存管理系统还应该保证内存分配和回收的公平性。

1 嵌入系统对内存管理的要求[1]

快速性:嵌入式操作系统的实时性要求内存分配过程尽可能快,因此不宜采用通用操作系统中完善但却复杂的内存分配策略。

可靠性:内存分配的请求必须得到满足,如果分配失败可能会带来灾难性的后果。嵌入式系统应用的环境千变万化,对可靠性要求极高。比如,汽车的自动驾驶系统中,系统检测到即将撞车,如果因为内存分配失败而不能进行相应的操作,就会发生车毁人亡的事故,这是不能容忍的。

高效性:考虑到成本,嵌入式操作系统的内存配置一般都不大,在有限的内存中还要装载操作系统和用户程序、数据,所以内存分配要尽可能地减少浪费,降低内存管理的开销。

2 内存碎片的分类[2]

事实上在系统中仍然有许多空闲内存时,仍会导致出现内存用完的情况。一个不断产生内存碎片的系统,不管产生的内存碎片多小,只要时间足够长,就会将内存用完。这种情况在许多嵌入式系统中,特别是在高可用性系统中是不可接受的。

内存分配程序浪费内存的3种基本方式:即额外开销、内部碎片以及外部碎片(见图1)。内存分配程序需要存储一些描述其分配状态的数据。这些存储的

收稿日期:2008-07-11

作者简介:张 为(1978-),男,湖南望城人,广东工业大学软件设计师,硕士研究生,主要研究方向为超精密材料加工和检测.

信息包括任何1个空闲内存块的位置、大小和所有权，以及其它内部状态详情。一般来说，1个运行时间分配程序存放这些额外信息最好的地方是它管理的内存。内存分配程序需要遵循基本的内存分配规则。例如，所有的内存分配必须起始于可被4、8或16整除（视处理器体系结构而定）的地址。内存分配程序把仅仅预定大小的内存块分配给用户，可能还有其它原因。当某个用户请求1个43字节的内存块时，它可能会获得44字节、48字节，甚至更多的字节。由所需大小四舍五入而产生的多余空间为内部碎片。

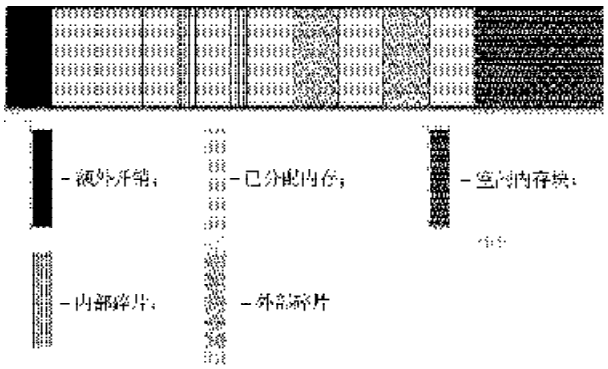

图1　内存碎片的几种形式

Fig. 1　Several forms of memory fragment

当已分配内存块之间出现未被使用的差额时，就会产生外部碎片。例如，1个应用程序分配3个连续的内存块，然后使中间的1个内存块空闲。内存分配程序可以重新使用中间内存块供将来进行分配，但分配的块不太可能正好与全部空闲内存一样大。若在运行期间，内存分配程序不改变其实现法与四舍五入策略，则额外开销和内部碎片在整个系统寿命期间保持不变。而额外开销和内部碎片会浪费内存，这是不可取的，但外部碎片才是嵌入系统开发人员真正的“敌人”，造成系统失效则是缘于分配问题。

3　减小内存碎片[3]

内存碎片是因为在分配1个内存块后，使之空闲，但不将空闲内存归还给最大内存块而产生。如果内存分配程序有效，就不能阻止系统分配内存块并使之空闲。即使1个内存分配程序不能保证返回的内存能与最大内存块相连接，但可以设法控制并限制内存碎片。所有这些做法涉及到内存块的分割，每当系统减少被分割内存块的数量，就会有所改进。目的是尽可能多次反复使用内存块，不要每次都对内存块进行分割，以正好符合请求的存储量。分割内存块产生大量的小内存碎片，犹如一堆散沙，以后很难把这些散沙与其余内存结合起来。比较好的办法是让每个内存块中都留有一些未用的字节，留有多少字节应根据系统在多大的程度上避免内存碎片而定。

4　内存分配算法的描述及分析

在嵌入式系统中最先适合内存分配算法、伙伴算法和固定存储量分配法是当今使用最多、最经典的分配算法[2]。最先适合内存分配算法是最常用的一种。它使用4个指针：MSTART 指向被管理内存的始端；END 指向被管理内存的末尾；BREAK指向MSTART和MEND之间已用内存的末端；PFREE 则指向第一个空闲内存块（如果有的话）。

伙伴（buddy）分配程序与本文描述的其它分配程序不同，它不能根据需要从被管理内存的开头部分创建新内存。它有明确的共性，各个内存块可分可合，但不是任意的分与合。每个块都有个朋友，或叫“伙伴”，既可与之分开，又可与之结合。伙伴分配程序把内存块存放在比链接表更先进的数据结构中，这些结构常常是桶型、树型和堆型的组合或变种。一般来说，伙伴分配程序的工作方式是难以描述的，因为这种技术随所选数据结构的不同而各异。由于有各种各样的具有已知特性的数据结构可供使用，所以伙伴分配程序得到广泛应用，通常在某种程度上限制内存碎片。

固定存储量分配程序有点像最先空闲算法。通常有一个以上的自由表，而且更重要的是，同一自由表中的所有内存块的存储量都相同。至少有4个指针：MSTART 指向被管理内存的起点，MEND 指向被管理内存的末端，MBREAK 指向MSTART与MEND之间已用内存的末端，而 PFREE[n]则是指向任何空闲内存块的一排指针。固定存储量分配程序很容易实现，而且便于计算内存碎片，至少在块存储量的数量较少时是这样。但这种分配程序的局限性是往往要有一个它可以分配的最大存储量。固定存储量分配程序速度快，并可在任何状况下保持速度。这些分配程序可能会产生大量的内部内存碎片，但对某些系统而言，它们的优点是重要的。

5　自适应内存管理算法及实现

buddy算法中用多条空闲链表去管理空闲内存块的方法十分有效，它把多个固定大小的空闲内存块分别置于不同的空闲块链表中，对分配适当大小的内存块是很有好处的，其分配和回收速度相当快，并且内存回收也很简单。借鉴buddy方法也建立类似的空闲内存块链表。现以1 MB内存为例说明自适应动态内存管理算法。将系统内存区分成128个内存仓，小于512 B（作为系统默认大小）的内存仓由固定大小的内存块组成，大于512 B的内存仓，由小于等于当前空闲块链表基数而又大于前一空闲块链表基数的空闲内存块组

成。内存仓从16 B开始以8 B递增到512 B，大于512 B将内存仓分成1 kB，2 kB，4 kB，…，128 kB，512 kB，1 024 kB。正是与buddy算法不同，所以需要再定义一个双向链表——物理块链表，以实现内存块的快速回收和相临空闲块的合并，形成大块空闲内存。

5.1 自适应内存管理算法原理及数据结构

下面给出自适应动态内存管理算法的工作原理描述。初始化时，如图2所示。

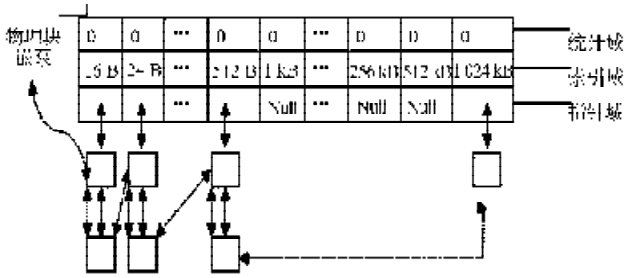

图2 系统初始化情况

Fig. 2 System initialization situation

统计域是记录相应空闲链表中内存块被应用程序所请求的频率，并根据统计域作为调整空闲块链表的依据。物理块链表定义为双向链表，用来连接所有物理位置相临的内存块（空闲块和占用块），引入物理链表极大地节约了内存回收后相临块合并的时间，实现了内存块的快速回收，因为它只需在物理链表中检查回收块的相临两个内存块的状态便可决定是否合并。虚线表示物理块链表连接内存块在物理位置上是相临，实线表示空闲链表的连接。

内存分配机制的数据结构定义如下[4]：

```
#define MAX_Num 1 084
typedef struct ArcNode{
int freesize; // 占用后，内存块剩余大小
int flag; // 内存块是否占用，0 表示不占用，1 表示占用
struct ArcNode *prior1; // 相邻内存块前指针
struct ArcNode *next1; // 相邻内存块后指针
struct ArcNode *prior2; // 空闲内存块前指针
struct ArcNode *next2; // 空闲内存块后指针
}ArcNode;
typedef struct VNode{
int frequence; // 统计域
int size; // 分配内存块大小
ArcNode *firstarc; // 指向第一个内存块
}VNode, AdjList[MAX_Num]
typedef struct{
AdjList Vertices;
}
```

5.2 固定大小内存块算法描述

初始化时，分配16 B到512 B的空闲链表固定大小的内存块，每个链表30块（图2中为表示方便已简化），那么计算(8×(2+3+…+64) ×30)／1 024≈ 480 kB，(1 024−480)kB =544 kB ≈ 540 kB，即大于512 B的链表中只有1 024 kB链表中有唯一的剩余块540 kB，而统计域值都为0。对于16 B到512 B的小块空闲链表来说，采用分配固定大小内存块的方式来满足分配请求是合适、合理的，这样处理可以避免分割本来就已经很小的内存块。当系统运行一段时间以后，根据统计域值信息，把空闲块链表进行适当的调整：若统计域为0，则说明该固定大小的内存块在这个系统中是很少用到的或者说是根本不需要的，此时可考虑将该空闲链表中的所有内存块合并成一个大的内存快插入到相应链表中；若统计域值很大，说明这种大小的内存块是最常用的，此时即便是物理位置相临的空闲内存块也不进行合并，以后可以快速地分配内存；若统计域信息不是很大，说明此类大小的内存块是需要应用的，但不是最频繁使用的内存块，此时可适当合并物理位置相临的内存块，并将所得的大块内存插入到相应的空闲链表中。

算法描述：

```
if (请求分配内存块<=512 B)
switch{
case AdjList[X].frequence=0:
{合并空闲链表中小块内存块成大块；
break;}
case AdjList[X].frequence= 大值：
   {break;}
case AdjList[X].frequence= 小值：
    {
for(i=0, p=AdjList[X].firstArc; i<小值；i++)
   p=p->next2；
    合并空闲链表中 p 指针之后的内存块；
   break;
}
}
```

系统运行一段时间后内存状态如图3所示。

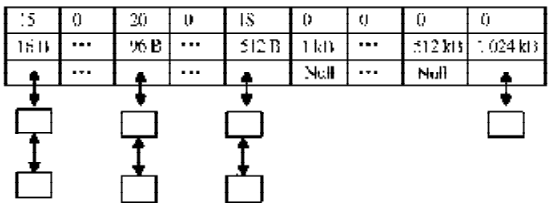

图3 系统运行一段时间后的情况

Fig. 3 Situation of system openration for a period time

按图4所示系统运行一段时间后，假设达到了稳定状态，可以发现它对16 B，96 B和512 B的内存块需

求最多也最频繁，这时可以根据统计域的信息将其它固定大小链表中的空闲块合并成大块的内存，并插入到相应的链表中。为表示清晰仅给出其中一小部分，仅以 400 B 和 500 B 为例，说明根据统计信息调整后的情况。(400 × 30) / 1 024 ≈ 12 kB；(500 × 30) / 1 024 ≈ 15 kB。

计算可知 400 B 和 500 B 的内存块合并后其大小约为 12 kB 和 15 kB，则合并完后，12 kB 和 15 kB 的空闲链表就由原来的空状态变为如图 4 所示。图中实框表示系统运行一段时间后稳定时内存情况。虚框表示根据统计信息合并后新插入的大块内存。

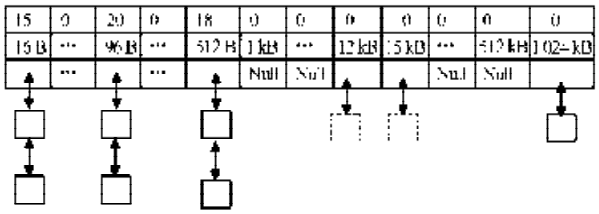

图 4　系统稳定，合并相应的空闲块后的情况

Fig. 4　Situation of unit merged together when the system was stable

5.3　可变大小内存块算法描述

对于 1 kB 到 1 024 kB 的大块空闲链表而言，由于这些链表不是固定大小的内存块，所以不仅要有以上所述根据统计域调节空闲链表的问题，而且还存在着内存块的分割问题。对于根据统计域调节空闲链表的方法文中采用如上方式，而内存块的分割方式，不采用固定大小的分割方式，其原因就是固定大小的内存分配方案内存利用率不高，很容易产生大量的内部碎片，其内存损失对嵌入式这样内存很有限的系统来说是接受不了的。此时考虑采用按实际大小分配内存的方法，这势必要分割内存块。从前面的论述可知无限制地分割内存块，最后会出现大量的内存碎片，为了避免这种情况的出现，若当分割后的空闲内存块小于 16 B，则本次分配请求不分割内存块，而是将整个内存块分配给请求。当分割后剩余的空闲块大于 16 B 时要插入到相应的空闲链表的头部，当以后有内存请求时应首先检查刚刚分割剩余的内存块是否满足请求，如果满足请求应首先分配它，这样处理就把问题归到了一个局部，避免分割其它的大块内存。

算法描述：

```
if (请求分配内存块>512 kB)
  {
   根据实际大小分割内存块；
   if (分割内存块 – 实际请求分配内存块<16 B)
     整块分割内存分配；
  else
    {
再次分割内存块并满足请求分配；
      将分割的空闲内存块插入相应空闲链表；
}
}
```

6　结语

自适应内存分配算法结合了最先适合内存算法、伙伴分配算法的优点，并进行了相关的改进，同时增加了统计域可以分析应用对内存块的需求特点，但同时它也是以牺牲时间来换得空间应用的效率。如何缩短空闲链表的查询时间、内存的分割与合并时间也是个值得探讨的问题。

参考文献：

[1] 邝　坚. Tomado/VxWorks 入门与提高[M]. 北京：科学出版社，2004：121-122.

[2] Lindblad J. 内存碎片处理技术[EB/QL]. [2004-10-08]. http://article.ednchina.com/2004-10/AtcShow2005128110353.htm.

[3] 王　铮，李志军. 一种适用嵌入式系统的自适应动态内存管理方案[J]. 计算机技术与发展，2007，17(3)：48-51.

[4] 严蔚敏，吴伟明. 数据结构(C 语言版)[M]. 北京：清华大学出版社，1997：203-206.

（责任编辑：罗立宇）