

一种面向智能芯片的快速存储管理算法

张宝军^{1,2}, 潘瑞芳^{1,2}

(1. 浙江传媒学院 新媒体学院, 杭州 310018; 2. 浙江省协同创新中心 媒体传播优化协同创新中心, 杭州 310018)

摘 要: 为有效管理智能芯片的存储资源, 提高存储空间利用率, 提出一种具有高存储空间利用率的存储管理算法。该算法将整个存储空间划分为大小相同的存储单元, 采用存储块数组和自由块数组进行管理。存储块由多个连续的存储单元组成, 分为占用块和空闲块, 存储块数组用来记录存储块的信息, 包括存储块的占用情况和大小。利用数组下标将大小相同的空闲块组织成多个双向链表, 方便存储块的管理。空闲块链表的头存储在自由块数组与空闲块大小对应的下标的元素中, 根据要分配的空闲块大小在自由块数组中快速找到空闲块链表的头, 实现存储的快速分配。实验结果表明, 该算法实现了存储块的分配、释放和合并的功能, 存储空间的利用率高, 在整体性能上优于伙伴算法。

关键词: 存储管理算法; 伙伴算法; 内部碎片; 外部碎片; 存储单元

中文引用格式: 张宝军, 潘瑞芳. 一种面向智能芯片的快速存储管理算法[J]. 计算机工程, 2016, 42(10): 101-106, 112.

英文引用格式: Zhang Baojun, Pan Ruifang. A Fast Storage Management Algorithm for Intelligent Chips[J]. Computer Engineering, 2016, 42(10): 101-106, 112.

A Fast Storage Management Algorithm for Intelligent Chips

ZHANG Baojun^{1,2}, PAN Ruifang^{1,2}

(1. New Media Institute, Zhejiang University of Media and Communications, Hangzhou 310018, China;

2. Media Communication Optimization of Collaborative Innovation Center,

Zhejiang Province Collaborative Innovation Center, Hangzhou 310018, China)

[Abstract] In order to effectively manage storage resources of the intelligent chip and improve the utilization rate of storage space, a storage management algorithm which has fast processing and high storage utilization is proposed. The algorithm divides the whole storage space into multiple units with the same size, and uses block array and free array to manage them. The block contains multiple continuous units. There are two kinds of blocks, the occupied block and the free block. The block array records the information of blocks, such as it is occupied or not and the size of it. The algorithm uses the subscript of block array to organize the free blocks into multiple double direction link tables, which makes the management of blocks easy. The head of free block link table is stored in the free array where the subscript of array is the same as the size of the free block. According to the size of the storage required, the free block with the size can be find quickly in the free array. Experimental results show the proposed realizes the function of storage allocating, releasing and merging, so it has high storage utilization, and the overall performance of the proposed algorithm is better than that of the buddy algorithm.

[Key words] storage management algorithm; buddy algorithm; internal fragmentation; external fragmentation; storage unit

DOI: 10.3969/j.issn.1000-3428.2016.10.018

1 概述

智能芯片在社会生产中发挥着重要的作用, 被广泛地应用于工业控制、电子行业和数码产品中。智能芯片的工作需要操作系统, 而存储管理是操作系统最核心的功能之一。操作系统本身的运行及其

各种应用程序的运行都离不开存储管理, 高效的存储管理算法能够充分发挥智能芯片的功能, 提高系统和应用程序运行的效率。反之则会影响系统的运行, 最终影响产品的质量和应用。

目前有效管理智能芯片存储资源的存储管理算法有位图算法^[1-2]、顺序适应算法^[3]、隔离列表算

基金项目: 浙江省重点科技创新团队基金资助项目(2011R50019); 国家新闻出版广电总局科研基金资助项目(2014-42)。

作者简介: 张宝军(1977—), 男, 讲师、博士, 主研方向为计算机体系结构、网络信息安全; 潘瑞芳, 教授。

收稿日期: 2015-10-09 **修回日期:** 2015-11-19 **E-mail:** zbjhover@zju.edu.cn

法^[4]和伙伴算法^[5-10]。

衡量存储管理算法的标准主要有 2 个:运行速度和存储空间利用率。在上述常见存储管理算法中,位图算法需要遍历位图数组,顺序适应算法需要遍历链表,分配速度都比较慢^[11-12]。伙伴算法的分配速度则有很大的不确定性,可能需要多次分裂才能分配到大小合适的存储空间。隔离列表算法分配速度很快,并且不需合并存储块,但是会产生较大的内部碎片,造成存储空间的浪费^[12-13]。

以上 4 种算法代表了存储管理的几种典型方法,近年来,随着智能芯片的广泛应用,对存储管理算法实时性的要求越来越高,出现了一些新的存储管理算法。SGI STL 算法^[14-15]采用 16 个链表管理来管理 8 Byte, 16 Byte, 24 Byte, …… , 128 Byte 大小的存储块,可以满足小于 128 Byte 大小的存储分配,且分配快速,根据需要的存储大小,从对应的链表中分配即可,释放存储单元时只要将其放入对应大小的链表中,无需合并存储单元。其缺点是只有 16 种大小的存储块可供分配,并且存储块很小。尽管可以通过增加链表数量的方法来管理更多大小的存储块,但是要管理较大的存储块需要增加的链表的数量很多。因此,SGI STL 不适合管理大的存储块。文献[16]提出一种具有可控存储大小的高速的存储管理算法。该算法具有很高的实时性,但是由于没有存储碎片合并的功能,因此,存储空间的利用率很低。为了提高存储空间的利用率,文献[17]提出一种自适应动态存储管理算法,该算法有利于减小存储碎片,但是运行效率较低。

可以看出,这些算法都是以上 4 种基本方法的组合和变种,虽然在某些方面有所改进,但是又引入了其他新的问题。在开发 IBM NP4GX 系列网络处理器的过程中,为有效管理 NP4GX 的存储资源,本

文提出一种高效的存储管理算法 HESMA,并与其他常见存储管理算法进行了比较与分析。

2 HESMA 存储管理算法

HESMA 将要管理的存储空间划分为大小相同的存储单元,存储单元是存储管理的最小单位,其大小可根据用户需要进行设定,管理的存储块为存储单元的整数倍。

根据存储单元在存储空间的位置对其进行编号,只要知道存储空间的首地址和存储单元的大小,就可根据存储单元的编号算出存储单元的地址。为方便介绍,下文的描述中将用存储单元的编号来表示其地址。

2.1 核心变量

为管理存储资源,HESMA 定义了 3 个核心变量,分别是:StorageUnitA,FreeListA 和 MaxFreeBlock Size。

(1) StorageUnitA。StorageUnitA 是一个结构体数组,其定义如下:

```
typedef struct _tag_StorageUnit
{
    int First;
    int Second;
} StorageUnit;
StorageUnit StorageUnitA[N];
N-存储单元的总数。
```

根据定义,StorageUnitA 数组的长度等于存储单元的个数,用来记录每个存储单元的信息。StorageUnitA 数组中的每个元素是一个 StorageUnit 结构体类型,该类型包括 2 个成员变量,存储单元的信息就存储在这 2 个成员变量中。

一个存储块可能由 1 个或多个存储单元组成,示意图如图 1 所示。

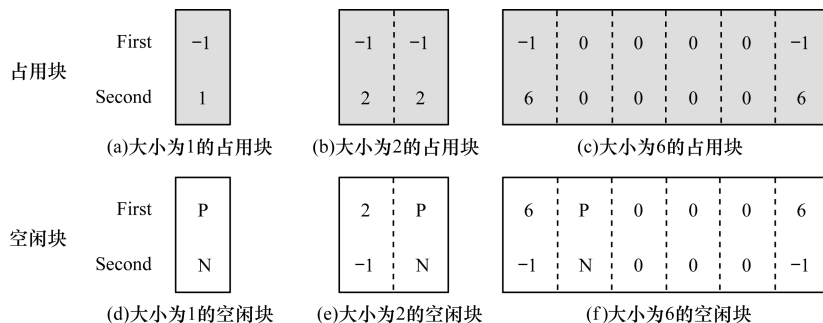


图 1 存储块示意图

根据图 1,存储块分为 2 种类型,分别是占用块(灰色的块)和空闲块(白色的块)。

1) 占用块

存储块分配时占用连续的存储单元,根据存储块的大小分为 3 种情况,分别是占用 1 个存储单元、占用 2 个存储单元和占用多个存储单元,如图 1(a)、图 1(b)、

图 1(c)所示。

StorageUnitA 数组的大小为存储单元的个数,记录所有存储单元的信息。当 First 为 -1 时表示该存储单元被占用,而相应的 Second 记录连续被占用的存储单元的个数,也就是该存储块的大小。对于占用多个存储单元的存储块(如图 1(c)所示),只需记录

首尾 2 个存储单元的信息,其他存储单元的值设置为 0。因为 *StorageUnitA* 数组会被初始化为 0,所以也就省去了设置 0 值的过程,节省了运算。

事实上,占 2 个存储单元的存储块可看成是占多个存储单元存储块的特例。

2) 空闲块

空闲块是指那些没有被分配的存储单元,其大小是连续的未被占用的存储单元的个数。空闲块的信息同样存储在 *StorageUnitA* 数组中,通过使用双向指针,巧妙地将相同大小的空闲块组织成双向链表的形式,从而方便管理。空闲块的描述也分为 3 种情况,如图 1(d)、图 1(e)、图 1(f) 所示。其中,First 为 P,表示前向指针,指向与本空闲块大小相同的链表中的上一个空闲块;Second 为 N,表示后向指针,指向与本空闲块大小相同的链表中的下一个空闲块。

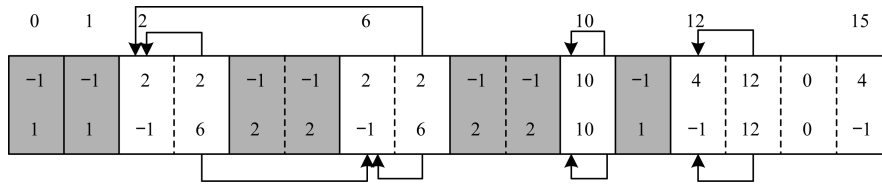


图 2 *StorageUnitA* 数组的记录信息

根据图 2 中 *StorageUnitA* 数组记录的信息,可以看出:首先存储单元的编号与 *StorageUnitA* 数组下标是相关联的,编号为 m 的存储单元的信息记录在 *StorageUnitA*[m] 中;其次存储单元的编号被作为指针,用来将具有相同大小的空闲块组织成双向链表的结构;最后就是存储单元的使用情况,编号为 0 和 1 的存储单元都是只占有一个存储单元的占用块,编号为 2 和 3 的存储单元未被分配,因此构成一个大小为 2 的空闲块。编号为 4 和 5 的存储单元是一个有 2 个存储单元的占用块。

图 2 中有 2 个大小为 2 的空闲块,这 2 个空闲块的起始地址分别为 2 和 6,被串成双向链表的形式。起始地址为 2 的空闲块为链表的头,其前向指针指向自己的起始地址,后向指针指向 6,也就是下一个和自己大小相同的空闲块。而起始地址为 6 的空闲块是

如果空闲块只有一个存储单元,那么该存储单元对应的信息就是 First 为 P,Second 为 N。P 和 N 指针将大小为 1 个存储单元的空闲块串起来,形成双向链表。

如果空闲块有 2 个存储单元,那么第 1 个存储单元的 First 为 2,表示空闲块的大小,而 Second 为 -1,表示从该存储单元开始是空闲块。第 2 个存储单元存储的仍然是指针 P 和 N。同样的 P 和 N 指针将大小为 2 个存储单元的空闲块串起来,形成双向链表。

如果空闲块有多个存储单元,以空闲块为 6 个存储单元为例,其示意图如图 1(f) 所示,首尾存储单元和第 2 个存储单元记录的信息的含义与前面相同,中间的其他存储单元记录为 0。

例如,假设要管理的存储空间为 64 KB,而存储单元的大小为 4 KB,那么一共有 16 个存储单元,假设运行一段时间后 *StorageUnitA* 数组存储的信息如图 2 所示。

链表的尾,其前向指针指向 2,后向指针指向自己。

(2) *FreeListA*。FreeListA 是一个整数数组,其定义如下:

```
int FreeListA[N];
N-存储单元的总数。
```

上文在介绍数组 *StorageUnitA* 时讲到大小相同的空闲块被串成双向链表的结构,那么链表的头被记录在 *FreeListA* 中。

这里巧妙地利用了数组下标来表示空闲块的大小,例如大小为 M 的空闲块链表的头被保存在 *FreeListA*[$M-1$] 中(注:数组下标从 0 开始)。

如果某些大小的空闲块为空,那么 *FreeListA* 数组中对应下标处元素的值为 -1。

例如,针对图 2 所示的空闲块信息,FreeListA 记录的信息如图 3 所示。

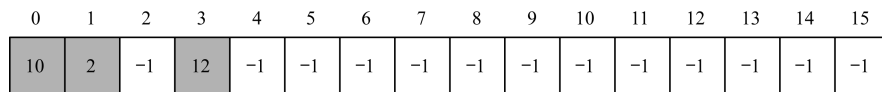


图 3 *FreeListA* 数组的记录信息

根据图 3 *FreeListA* 记录的信息,可以看出目前还有大小分别为 1, 2, 4 的空闲块,其空闲块链表的头地址分别是 10, 2, 12。

(3) *MaxFreeBlockSize*。在分配存储块时,如果需要的存储块的大小在当前的空闲块链表中没有相匹配的,那么可以从最大的空闲块中分配,MaxFreeBlockSize 变量用来记录最大空闲块的大小,其定义

如下:

```
UINT MaxFreeBlockSize;
```

例如,图 3 中 *FreeListA* 记录的最大空闲块的大小为 4,那么 *MaxFreeBlockSize* 的值为 4。

2.2 算法描述

本节按照算法功能从以下 4 个方面对 HESMA 算法进行介绍。

(1) 初始化

在初始化时,首先根据存储空间的大小 x 和存储单元的大小 y 算出存储单元的个数 N 。

$$N = x/y$$

(1)

数组 $StorageUnitA[N]$ 的值全部清 0, 初始化代码如下:

```
for (int i = 0; i < N; i++)
{
    StorageUnitA[i].First = 0;
    StorageUnitA[i].Second = 0;
}
```

数组 $FreeListA[N]$ 初始化过程如下:

```
for (int i = 0; i < N - 1; i++)
```

```
FreeListA[i] = -1;
```

```
FreeListA[N - 1] = 0;
```

```
MaxFreeBlockSize = N;
```

因为在最开始时,整个存储空间是一个大小为 N 的空闲块,该空闲块的首地址为 0,所以 $MaxFreeBlockSize$ 的值为 N ,而 $FreeListA[N - 1]$ 指向该空闲块的首地址 0。

(2) 存储分配

假设要分配的存储块大小为 $sizeOfBlock$, 一个存储单元的大小为 $sizeOfUnit$, 用 $numOfUnit$ 表示要分配的存储单元的个数,存储分配过程如图 4 所示。

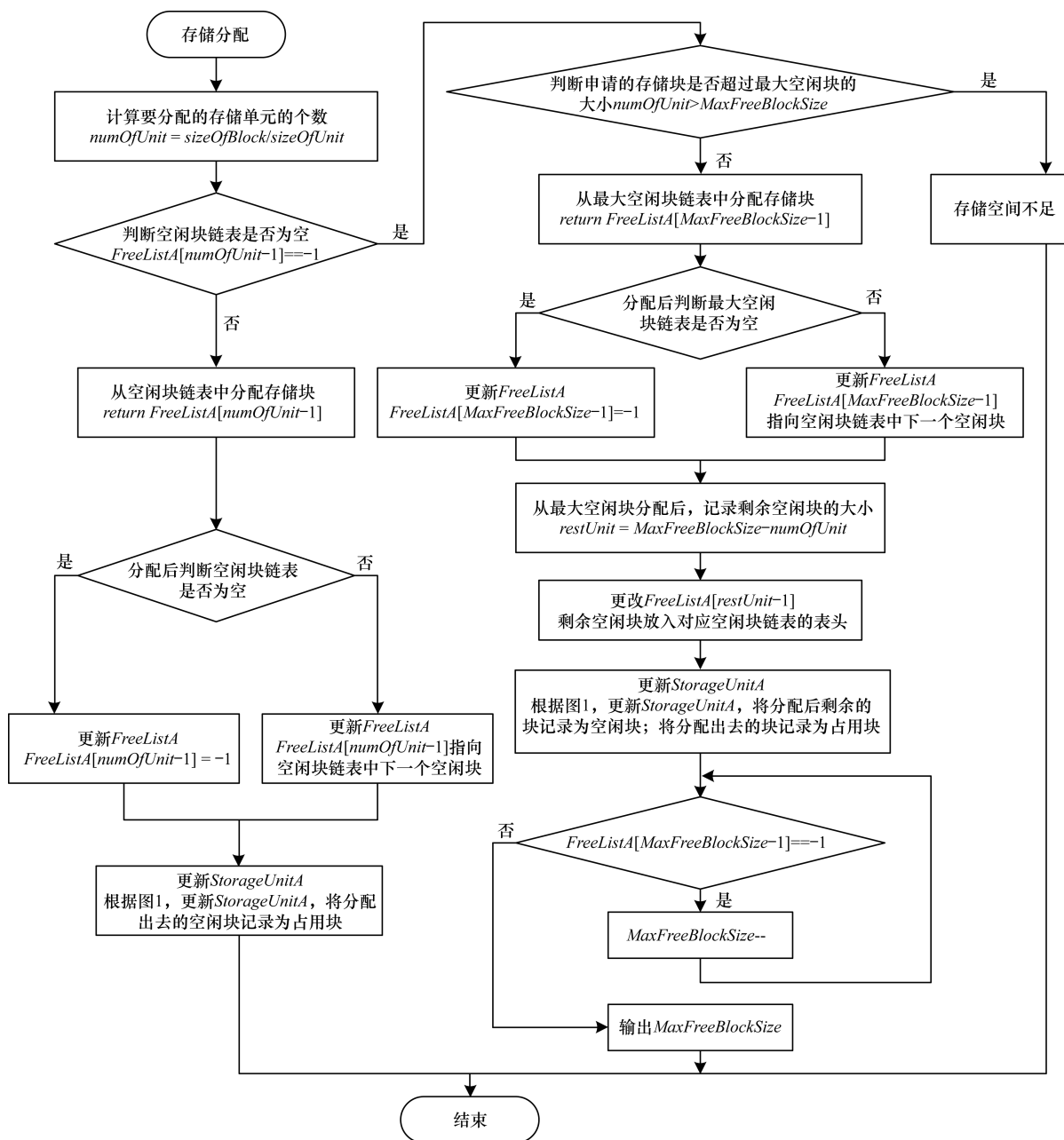


图 4 存储分配流程

存储分配的基本流程是根据要分配的存储块大小首先在相应的空闲块链表中分配,如果空闲块链

表为空,则到最大空闲块链表中分配。如果申请的存储空间超过最大空闲块大小,则显示存储不够。

在存储分配过程中有以下 3 点需要说明:

1) 在最开始时,整个存储空间是一个大的空闲块,因此只有一个最大空闲块,而没有其他对应大小的空闲块链表,随着存储分配、释放及合并的运行,会出现各种大小的空闲块链表。

2) 每次分配存储块后,需要更新 *StorageUnitA* 和 *FreeListA*。尽管看起来较为复杂,但是由于本文算法用数组下标对存储单元进行定位,并巧妙地将数组下标作为指针实现了链表的功能,因此能够快速定位 *StorageUnitA* 和 *FreeListA* 中要修改的元素,对其进行更新。

3) 如果存储块是从最大空闲块链表中分配的,并且最大空闲块只有一个,那么 *MaxFreeBlockSize* 的值也需要更新。判断新的最大空闲块大小的方法很简单,通过查看数组 *FreeListA* 即可获得,最大空闲块的大小为更新后 *FreeListA* 中下标最大的值不为 -1 的元素的下标加上 1。

(3) 存储释放与合并

存储释放与合并是先后进行的,存储块释放后,根据 *StorageUnitA* 的记录检查其周围是否有其他空闲块,如果有则将连续的空闲块合并成一个大的空闲块,然后将新的空闲块放到对应的空闲块链表中,最后更新 *FreeListA*, *StorageUnitA* 和 *MaxFreeBlockSize* 的值,其过程如图 5 所示。

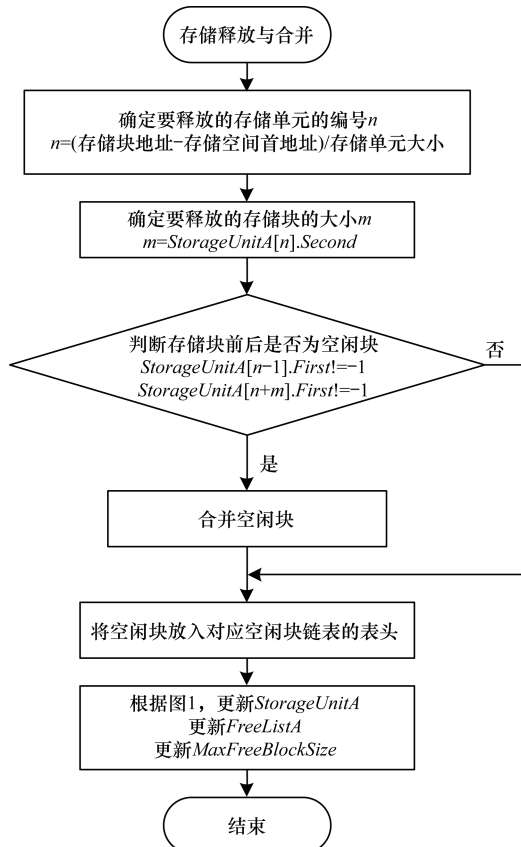


图 5 存储释放与合并流程

存储释放与合并后,同样需要更新 *StorageUnitA*, *FreeListA* 和 *MaxFreeBlockSize* 的值,其更新过程与存储分配一样。

2.3 算法分析

通过上述介绍, HESMA 是一种高效的存储管理算法,该算法具有以下特点:

(1) 通用性。HESMA 采用预分配的方式从 NP4GX 分配一大块存储资源,然后对存储资源进行管理,这种方式屏蔽了底层存储设备的类型和结构,因此具有很好的通用性。

(2) 功能全面。HESMA 算法提供全面的存储管理功能,能够分配、释放和合并存储单元。

(3) 运行快速。采用数组来管理存储单元,并通过数组下标将空闲块组织成双向链表的结构,以及对最大空闲块链表的管理,使得 HESMA 能够快速定位和分配大小合适的空闲块,释放和合并存储单元也非常快速。

(4) 节省存储空间。HESMA 算法具有较高的存储空间利用率,一方面因为将存储空间划分为大小相同的存储单元来管理,使得 HESMA 没有外部存储碎片,而内部存储碎片可通过设置存储单元的大小进行控制;另一方面因为具有释放和合并存储单元的功能,使存储资源得以回收利用。

3 实验与结果分析

Buddy 算法是目前应用最为广泛的存储管理算法,本节对 HESMA 和 Buddy 算法在存储分配上的时间复杂度进行了比较分析,并通过实验比较了 HESMA 和 Buddy 算法各方面的性能。

(1) 时间复杂度比较

根据 HESMA 和 Buddy 算法进行存储分配的具体过程,2 种算法分配存储的时间复杂度如表 1 所示。

表 1 HESMA 与 Buddy 时间复杂度比较

算法	时间复杂度
HESMA 算法	$O(1), O(2)$
Buddy 算法	$O(\lg n), n = 2k$

在表 1 中, n 表示整个存储空间的大小; k 表示内存拆分的最大次数。根据表 1, Buddy 算法的分配速度则有很大的不确定性,可能需要多次分裂才能分配到大小合适的存储空间,如果整个存储空间的大小为 n ,其时间复杂度为 $O(\lg n)$ 。而 HESMA 算法中空闲块的信息存储在自由块数组 *FreeListA* 中,根据要申请的存储块的大小找到 *FreeListA* 中对应下标的元素,就能获得对应大小的空闲块。如果 *FreeListA* 中对用下标的元素的值为 -1,表明没有找

到对应大小的空闲块,那么会到最大空闲块中进行分配,其存储分配的时间复杂度为 $O(1)$ 或 $O(2)$ 。可见, HESMA 算法分配存储的速度非常快,可以立刻返回用户需要的存储单元。

(2) 实验测试结果比较

实验环境: 计算机型号: Dell Latitude E6440; 处理器: Intel (R) Core (TM) i7-4610M CPU @ 3.00 GHz; 内存: 8.00 GB; 系统类型: Windows 7 SP1 64 位。

实验方法: 首先从内存中分配 2 MB 的连续存储空间作为管理的对象, 然后对该存储空间分别采用 HESMA 算法和 Buddy 算法进行分配、释放和合并操作, 并记录操作完成的时间。对于 HESMA 算法, 将 2 MB 的存储空间划分为 2^{16} 个 32 Byte 的存储单元进行管理。

1) 存储分配比较

分别用 HESMA 算法和 Buddy 算法进行 400 次、4 000 次和 40 000 次存储分配, 记录分配的时间, 其结果如表 2 和图 6 所示。

表 2 2 种算法存储分配性能比较 ms

算法	存储分配 400 次	存储分配 4 000 次	存储分配 40 000 次
Buddy 算法	0.29	3.62	44.28
HESMA 算法	0.22	2.15	22.19

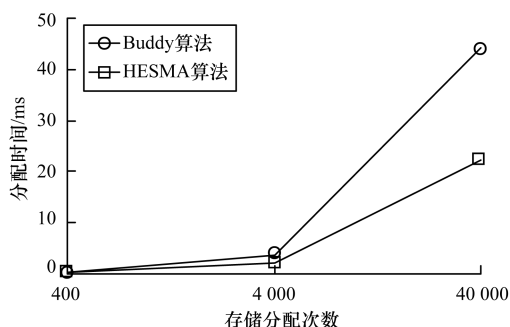


图 6 2 种算法存储分配性能比较

根据表 2 和图 6 可知, HESMA 算法的分配速度要快于 Buddy 算法。这是因为 Buddy 算法需要多次分裂才能找到大小合适的存储块, 每次存储分配的时间是不定的, 而 HESMA 算法通过数组下标定位的方式可以快速找到合适的空闲块, 完成存储的分配, 并且整个分配过程较为固定, 每次分配消耗的时间也比较接近, 因此分配速度较快。分配时间的稳定性同时解释了表 2 中 HESMA 算法分配次数与分配时间的关系接近线性增长, 分配次数近 10 倍增长, 消耗的时间也近 10 倍增长的测试结果。

2) 整体性能比较

整体性能比较是对算法分配、释放、合并等全部功能的综合测试, 因为这 2 种算法的合并功能都自动包含在存储释放的过程中, 所以只需要调用分配和释放函数即可。

同样分别用 HESMA 算法和 Buddy 算法进行 400 次、4 000 次和 40 000 次存储分配和释放的综合测试, 记录消耗的时间, 其结果如表 3 和图 7 所示。

表 3 2 种算法整体性能比较 ms

算法	存储分配 400 次	存储分配 4 000 次	存储分配 40 000 次
Buddy 算法	0.35	3.88	45.03
HESMA 算法	0.26	2.33	22.84

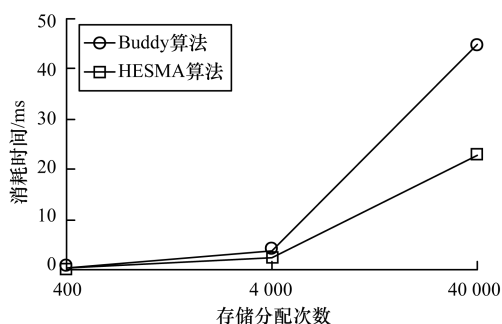


图 7 2 种算法综合性能比较

根据表 3 和图 7, HESMA 算法和 Buddy 算法的释放与合并速度都很快, 其处理速度主要由存储分配速度决定, 因此, 综合性能的比较结果与存储分配性能的比较结果类似。

4 结束语

为了对智能芯片有限的存储资源进行有效管理, 本文设计并实现一种高效的存储管理算法 HESMA。该算法最大的特点是分配快速, 存储空间利用率高, 其核心思想是将整个存储空间划分为大小相同的基本存储单元, 存储块 (包括占用块和自由块) 的大小为存储单元的整数倍; 将大小相同的自由块组织成一个个双向链表, 表头存储在自由块大小对应的数组下标中, 分配存储时可根据需要快速找到对应的自由块; 记录存储块的起始地址、结束地址和大小等信息, 当释放占用块时, 可根据记录的信息与其前后相邻的存储块进行合并, 从而节省存储空间。实验结果表明, HESMA 算法功能全面、分配快速、存储空间利用率高、通用性强, 适用于智能芯片的存储管理。

(下转第 112 页)

参考文献

- [1] Lawton K P. Bochs: A Portable PC Emulator for Unix/x[J]. Linux Journal, 1996, 1996(29): 7.
- [2] Bellard F. QEMU: A Fast and Portable Dynamic Translator[C]//Proceedings of Annual Conference on USENIX Annual Technical Conference. Berkeley, USA: USENIX Association, 2005: 41-46.
- [3] Smith J, Nair R. Virtual Machines: Versatile Platforms for Systems and Processes[M]. Amsterdam, Holland: Elsevier, 2005.
- [4] 黄聪会, 陈 静, 罗 樵, 等. 面向二进制移植的虚拟化技术[J]. 计算机应用研究, 2012, 29(11): 4185-4188.
- [5] 陈 乔, 蒋烈辉, 董卫宇, 等. 基于动态二进制翻译技术的仿真器研究[J]. 计算机工程, 2011, 37(20): 277-279.
- [6] Wine User's Guide[EB/OL]. (2015-05-25). <https://www.winehq.org/docs/wineusr-guide/index>.
- [7] 胡嘉杰, 蒋乐天. Android 系统中 Win32 环境的高性能虚拟[J]. 计算机应用, 2015, 35(3): 886-890.
- [8] 苏 煜, 张 涛, 靳国杰, 等. 基于龙芯的 Windows 软件运行环境的设计与实现[J]. 江苏科技大学学报: 自然科学版, 2014(4): 387-391.
- [9] Chiueh T C. A Survey on Virtualization Technologies, Technology Report: 179[R]. New York, USA: State University of New York at Stony Brook, 2005.
- [10] Wine Developer's Guide[EB/OL]. (2015-05-25). <https://www.winehq.org/docs/winedev-guide/index>.
- [11] 钟柏松. Wineserver 机制在 Linux 内核中实现的研究[D]. 南昌: 南昌大学, 2012.
- [12] Henderson A, Prakash A, Yan L K, et al. Make it Work, Make it Right, Make it Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform[C]//Proceedings of International Symposium on Software Testing and Analysis. New York, USA: ACM Press, 2014: 248-258.
- [13] Gligor M, Fournel N, Pétrot F. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation[C]//Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis. New York, USA: ACM Press, 2009: 71-80.
- [14] Huang Conghui, Chen Jing, Zhang Li, et al. Performance Evaluation of Virtualization Technologies for Windows Programs Running on Linux Operating System[C]//Proceedings of Conference on Network Computing and Information Security. Berlin, Germany: Springer, 2012: 759-766.
- [15] Corporation C. SQLIO Disk Subsystem Benchmark Tool[EB/OL]. (2015-07-03). <http://www.microsoft.com/en-us/download/details.aspx?id=20163>.

编辑 索书志

(上接第 106 页)

参考文献

- [1] Blunden B. Memory Management-algorithms and Implementation in C/C++[M]. [S. l.]: Wordware Publishing, Inc., 2003.
- [2] 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): 1020-1021.
- [3] Daniel S G. A Class of Dynamic Memory Allocation Algorithm[J]. Communications of the ACM, 1973, 16(10): 615-618.
- [4] Margolin B H, Parmelee R P, Schatzoff M. Analysis of Free Storage Algorithms[J]. IBM Systems Journal, 1971, 10(4): 283-304.
- [5] 严蔚敏, 吴伟民. 数据结构(C语言版)[M]. 北京: 清华大学出版社, 1992.
- [6] 李小群, 孙玉芳. Linux 内存管理机制的分析与研究[J]. 计算机科学, 2002, 29(4): 18-20.
- [7] 曹国钧. 计算机内存管理及其优化技术[M]. 成都: 电子科技大学出版社, 1998.
- [8] Ellen S G. 内存管理深入剖析[M]. 李谊瑞, 周克昌, 译. 北京: 电子工业出版社, 1995.
- [9] 沈 勇, 王志平, 庞丽萍. 对伙伴算法内存管理的讨论[J]. 计算机与数字工程, 2004, 32(3): 40-43.
- [10] Puttkamer E V. A Simple Hardware Buddy System Memory Allocator[J]. IEEE Transactions on Computers, 1975, 24(10): 953-957.
- [11] 何 伟, 何建中. FreeRTOS 内存管理机制的分析和改进[J]. 计算机工程, 2010, 36(10): 67-69.
- [12] 姜力波. 基于 Linux 内存管理的分析和研究[D]. 合肥: 中国电子科技大学, 2011.
- [13] 余广桥, 李玉涛, 王江汉. 嵌入式终端平台内存管理算法比较[J]. 单片机与嵌入式系统应用, 2012, 12(5): 64-65.
- [14] 侯 捷. STL 源码剖析[M]. 武汉: 华中科技大学出版社, 2002.
- [15] 陈力军, 魏永军, 连 超. 一种内存管理系统及其分配方法: CN01139150.2[P]. 2001-07-21.
- [16] Dong Wensheng, Shen Chunfeng. Memory Management System with Controlled Volume and High Performance[J]. Control Engineering of China, 2013, 20(S0): 69-71.
- [17] 王 铮, 李志军. 一个嵌入式系统的自适应动态内存分配方法[J]. 计算机技术与发展, 2007, 17(3): 48-50.

编辑 索书志