Dynamic storage allocation for real-time embedded systems*

M. Masmano, I. Ripoll, and A. Crespo
Universidad Politécnica de Valencia, Spain.
{mmasmano,iripoll,alfons}@disca.upv.es

Abstract

Dynamic memory allocation DSA algorithms have played in important role in the modern software engineering paradigms and techniques (as object oriented paradigm). Additionally, its utilization allows to increase the flexibility and functionalities of the applications. There exists in the literature a large number of works and references to this particular issue. However, in the real-time community the use of dynamic memory techniques has been not considered as an important issue because spatial and temporal worst case for allocation and deallocation operations were insufficiently bounded. It is significant the reduced number of papers about this topic in the more relevant real-time events.

Considering these reasons new dynamic storage allocator DSA algorithms with a bounded and acceptable temporary behavior must be developed to be used by RTOS. On this paper a new DSA algorithms called Two Level Segregated Fit (TLSF), developed specifically to be used by RTOS, is introduced and compared with other previously proposals.

1 Introduction

The study of dynamic storage allocation (*DSA*) algorithms has been an important topic in the operating systems research and implementation and has been widely analysed. Wilson et al. [7] wrote an excellent survey/tutorial of the research about storage allocation done between 1961 and 1995. Due to the large amount of existing *DSA* algorithms and studies, the reader can get to think that the problem of dynamic memory allocation has been already solved. This can be true in most applications types, however the situation for the real time applications is quiet different.

In real-time systems it is needed to know in advance the operation bounds in order to analyse the system. The dynamic memory allocation operations lack of a sufficient bounds to provide efficient support to the applications which means that the time required to allocate and deallocate memory are both unpredictable . Additionally, the dynamic memory management can introduce an important memory fragmentation that can generate unreliable service when the application runs for large period of time. It is important to point out that a fast response and high thoughtput are important characteristics that has to be taken into account in any systems, and in particular in a RTS. But the main characteristic that defines a RTS is to guarantee the timing constrains.

Most of the DSA algorithms were designed to provide fast response time for the most probable case achieving a good overall performance, although the worst case can be high. For these reasons, most RTOS developers and researchers avoid the use of dynamic memory at all, or use it in a restricted way, for example, only during the system startup [6].

This paper presents a new algorithm for dynamic memory alocation that tries to solve the problem of the worst case bound maintaining the efficiency of the allocation and deallocation operations. Also, the fragmentation problem is improved enougth to be used in real-time systems running for long time. Section 2 presents a review of the requirements that should be fulfilled by the dynamic memory allocation algorithms for real-time applications. Section 3 analyses some of the most relevant DSA implementations used in real-time and non-real-time environments. In section 4, the design guidelines of the proposed DSA are presented. The propossed algorithm is outlined in section 5.

2 Real-Time requirements

One of the key issues in real-time systems is the schedulability analysis to determine whether the timing constraints will be satisfied at run-time or not. Regardless of which analysis and scheduling techniques are used, it is essential that the real-time designer be able to determine the worst-case execution time of all the activities involved in the application. Moreover, real-time applications run for large periods of time. During its operation, memory can be allocated and deallocated many times which aggravates the memory fragmentation problem.

Considering these aspects, the requirements of real-time applications regarding dynamic memory can be stated as follows:

Bounded response time. The WCRT has to be known a priority and, if possible, be independent of application data. This is the main requirement the must be meet.

*This work has been supported by the European Commision project: IST-2001-35102(OCERA) http://www.ocera.org.

Fast response time. Although having a bounded response time is a must, the response time has to be fast to be usable. A bounded DSA algorithm which is 10 times slower than a conventional one, is not practical.

Memory requests have to be always satisfied. In non-real time systems applications can receive a null pointer or are just killed by the OS when the system runs out of memory. It is obvious that is not possible to always grant all the memory requested. But the DSA algorithm has to minimise the chances of exhausting the memory pool by minimising the fragmentation and wasted memory.

Although there is large range of real-time systems with different memory constraints and hardware support, the study presented in this work in progress is focussed on embedded systems where memory is a scarce resource and there is no MMU support.

3 DSA in real-time systems

This section analyses some of the most relevant DSA available implementations used in real-time and non-real-time environments. A comparison with other implementations (like QNX, Lynxs, or VxWorks) were not possible because that code is not public or only released under non-disclosure agreements.

3.1 Doug Lea implementation

Doug Lea's allocator [4] is one of the most used allocators, both in standard applications and in real-time systems (eCos, glibc, etc.). M.S. Johnstone and P.R. Wilson [2] concluded that it is an *excellent allocator* in relation to fragmentation; and widely considered to be among the best uniprocessor allocators [1].

Doug Lea's allocator is an implementation of several strategies combined to provide good performance in several parameters: speed, space-conserving, portability, block locality and tunable. It uses different strategies depending on the size of the requested blocks. Also, it is important to note that free blocks are immediately coalesced.

This algorithm has some limitations that make it not suitable in real-time systems. It focusses optimisations on most used block sizes (from 16 to 512 bytes) using a fine grain (8 bytes) segregated fit structure; but for larger blocks, it implements a simple free list with best-fist search, which can degenerate in lengthy searches along a simple linked list. One important area where this allocator do not perform well is with network or disk drivers that work with medium size blocks. Although it do not provides the required performance for real-time applications, this implementation has been widely used and studied, and it is considered as one of the best allocators.

3.2 BGET

This a portable and customisable implementation of the first-fit and best-fit strategies, the strategy used is selected via conditional compilation. It was written in 1972 and used in a wider range of application from small embedded systems to large mainframes. When bget is initialised, it is possible to register several user call back functions that the allocator can call to compact, grow and shrink the managed memory pool.

As expected, when BGET is compiled to use first-fit strategy it provides a good mean execution time under real workloads. But the worst case execution time depends on the size of the memory pool, which is not acceptable for real-time systems.

3.3 RTEMS

RTEMS is a real-time executive which provides a high performance environment for embedded applications. RTEMS is a full featured but small and compact RTOS. The DSA policy used in RTEMS is the first-fit, implemented using a single linked list. It never coalesce blocks, that is, free blocks are just inserter into the free list. It uses the system facility `sbrk()` to enlarge the memory poll.

3.4 Half-Fit

Proposed by T. Ogasawara [5] this algorithm is the only DSA designed –known by the authors– to fulfil the specific requirements of real-time systems. It is based on segregated lists, where the lists are power of 2. It also suggests the use of advanced bit instructions ("find first bit set" `FFS()` that are available on most modern processors and libraries) to find in constant time the best free queue to allocate the requested block. The asymptotic complexity of the `malloc()` function is $O(1)$.

Deallocated blocks are immediately coalesced, therefore in the worst case, the `free()` function will merge tree block (current block with physically previous and next blocks). Although the asymptotic complexity of the `free()` function is constant, the free function perform more operations and the time host is higher than the malloc function.

4 TLSF strategy

As described before, there is not a single DSA algorithm suitable for all application types. As already stated, real-time applications are quiet different from conventional applications. This sections analyses how real-time applications requirements determined the design of TLSF to fullfill most of their requirements.

Real-time applications requirements:

- Bounded response time.
- Fast response time.
- Bounded fragmentation.
- Low fragmentation.

Characteristics of the hardware used in embedded systems:

- Small amount of memory.
- No special hardware (MMU) available to support virtual memory.

In order to meet these constrains and requirements the TLSF was designed following the next guidelines:

Immediate coalescing: As soon as a block of memory is freed, *TLSF* will immediately merge it with adjacent free blocks, if any, to buildup a larger free block. Although delayed coalescing can improve the performance of the allocator, it adds unpredictability (a block request may require to merge a unbounded number freed but not merged blocks) and also increases amount of fragmentation. Therefore, deferred coalescing can not be used in real-time.

Splitting threshold: The smallest block of memory allocated is 8 bytes. By limiting the minimum block size to 8 bytes, it is possible to store inside the free blocks the information needed to manage free block, therefore optimising the memory usage. The list of free blocks is stored inside the same free memory blocks.

Good-fit strategy: TLSF will return the smallest chunk big enough to hold the requested block. Since most applications only use a small range of sizes, best-fit tend to produce the least fragmentation on real loads compared to other general approaches such as first-fit [7, 2]. Also, a best-fit (or an almost best-fit, also called *good-fit*) strategy can be implemented in a efficient and predictable way using segregated free lists. On the other hand, other strategies like first-fit or next-fit are difficult to implement with a predictable algorithm. Depending on the request sequence, a first-fit strategy can degenerate in a long sequential search in a linked list. TLSF implements a *good-fit* strategy, that is, it uses a large set of free lists, where each list is a **non-ordered** list of blocks between the size class and the next size class.

No reallocation: It is assumed that the original memory pool is a single large block of free memory, and no `sbrk()` function is available.

Same strategy for all block sizes: The same allocation strategy and policy used for any requested size.

Memory is not cleaned-up: Neither the initial pool nor the free blocks are zeroed. It is assumed that TLSF will be used in a trusted environment, where applications are written by well intended programmers. Therefore, initialising the memory introduces overhead with a useless feature. The programmer has to initialise all the data he allocates, as good programming practices recommends.

5 *TLSF* structure

TLSF uses a segregated fit mechanism to implement a good-fit policy. The basic segregated fit mechanism uses an array of free lists, with each array holding free blocks within a size class. In order to speedup the access to the free blocks and also to manage a large set of segregated lists (to reduce fragmentation) the array or lists has been organised as a two level array. The first-level array divides free blocks in classes that are a power of two apart (8, 16, 32, 64, etc.); and the second-level sub-divides each first-level class linearly, where the number of divisions (SLI) is a user configurable parameter. Each array of lists has an associate bitmap used to mark which lists are empty and which ones contain free blocks. Information regarding each block is stored inside the block itself.

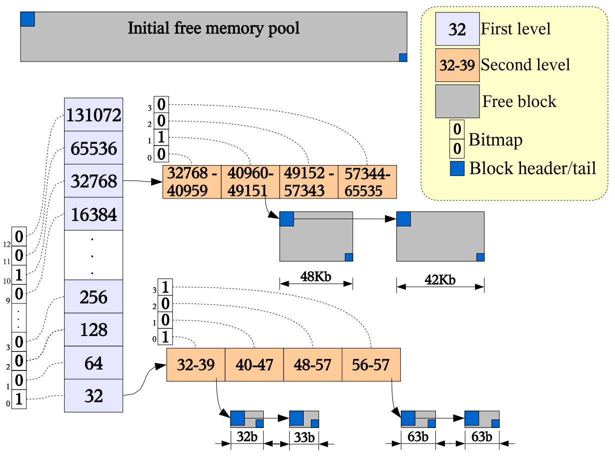

Figure 1: TLSF free data strcuture overview.

In order to coalesce easily free blocks, the *TLSF* employs the *boundary tag* technique proposed by D. Knuth in [3] which consists on adding a footer field to each free or used block which is a pointer that points to the start of the same block. When a block is freed the footer of the previous block (which is located one word before the freed block) is used access the head of the block to check whether it is free or not and merge both block accordingly.

Therefore, each free block is linked in two double linked lists: the segregated list with hold the blocks belonging to the same size class, and a list ordered by physical size.

6 Algorithm description

Most *TLSF* internal operations relay on the `segregate_list()` mapping function which given the size of a block calculates the indexes of the two arrays that points to the corresponding segregate list.

$$\texttt{segregate\_list}\,(size) = \begin{cases} f & := \lfloor log_2\,(size) \rfloor \\ s & := \left(size - 2^f\right) \frac{SLI}{2^f} \end{cases}$$

This function can be efficiently implemented using shift and search bit instructions (available in most modern processors) and exploiting numerical characteristics. The first level index ($\lfloor log_2\,(size) \rfloor$) can be computed as the position of the last bit set (bit set to one) of the size. The second level index can be obtained using a mask on the first SLI bits of the size that follow the last bit set. For example, supposing a SLI=4 and given the size 460, the first level index is `f=8` and the second and the second level index is `s=12`:

$$\texttt{size} = 460_d = \overset{15}{0}\,\overset{14}{0}\,\overset{13}{0}\,\overset{12}{0}\,\overset{11}{0}\,\overset{10}{0}\,\overset{9}{0}\,\overset{\overset{\texttt{f=8}}{8}}{1}\;\underbrace{\overset{7}{1}\overset{6}{1}\overset{5}{0}\overset{4}{0}}_{\texttt{s=12}}\;\overset{3}{1}\overset{2}{1}\overset{1}{0}\overset{0}{0}{}_b$$

This function gives a constant time search ($O(1)$) in the *TLSF* structure.

The internal operations provided by *TLSF* required to implement malloc and free functions are:

Get a free block: This function returns a pointer to a free block of the required size or bigger. The request size is rounded up to the nearest free list and the search sequence is as follows:

1.- calculate the "`f`" and "`s`" indexes (using the mapping function) which are used to get the head of the free list holding the closer class list. If this list is not empty then the block at the head of the list is removed from the list (marked as busy) and returned to the user; otherwise,

2.- search the next (bigger than the requested size) non empty list in the TSLF structure. This search is done in constant time using the bitmap masks and the find first set (ffs) bit operation. If a list is found, then the block at the head of the list will be used to fulfil the request. Since this block is bigger than which was requested it is split and the remaining left is inserted into the corresponding free list. If no bigger block is found, then

3.- the memory is taken directly from the *memory pool*. If there is not enough memory in the pool, then the allocator just fails due to memory exhaustion.

Insert a free block: The mapping function is used to calculate the "f" and "s" indexes to find the class list where the block has to be inserted.

Coalesce blocks: Using the boundary tag technique the head of the previous block is consulted to check whether it is free or not. If the previous block is free then the block is removed from the segregated list and merged with the current block. The same operation is carried out with the following physical block.

Using these internal operations, the malloc and free functions are implemented with a $O(1)$ time. On the other hand, exact layout of the *TLSF* data structure has been designed to enhance cache and TLB locality.

7 Results and conclusions

Figure 2 show the preliminary results of a comparison between Douglas Lee, BGET and *TLSF*. Time has been measured using the `tsc` register (i386) which is a counter incremented on every processor clock cycle. The test code used to produce this plot is:

```
for (x=32; x<10*1024; x+=32){
  start=time();malloc(x);end=time();
  y=end-start;
}
```

As can bee seen, *TLSF* is always below 1000 cpu ticks.

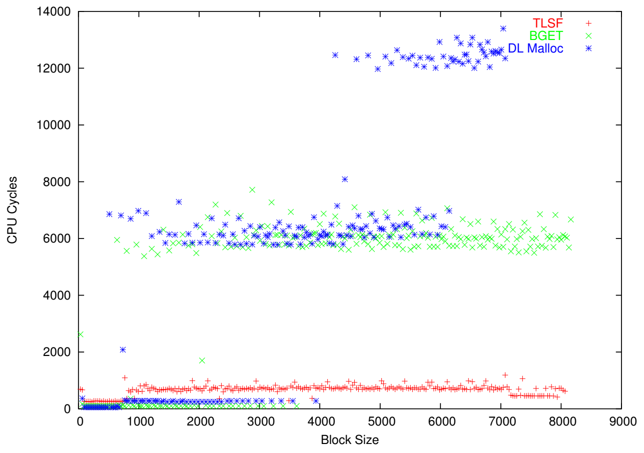

Figure 2: Comparative test

The first comparative tests are quiet encouraging, and new algorithms are under study using MMU support. The new versions will be freely available at the OCERA web site.

References

[1] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001.

[2] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved ? In *Proceedings of the International Symposium on Memory Management (ISMM'98), Vancouver, Canada.* ACM Press, 1998.

[3] D.E. Knuth. *The Art of Computer Programming, volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, USA, 1973.

[4] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[5] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. *2nd International Workshop on Real-Time Computing Systems and Applications*, page 21, 1995.

[6] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. *14 th Euromicro Conference on Real-Time Systems (ECRTS'02)*, page 41, 2002.

[7] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In H.G. Baker, editor, *Proceedings of the International Workshop on Memory Management (IWMM'95), Kinross, Scotland, UK*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, Berlin, Germany, 1995.