A Survey of Dynamic Real-Time Memory Management Systems

Hannes Payer
University of Salzburg
hpayer@cosy.sbg.ac.at

Abstract

In general, hard real-time applications avoid the use of dynamic memory management systems due to the unbounded response times of dynamic memory management operations. Since the complexity of real-time applications grows, dynamic memory management systems are needed to increase their flexibility and functionalities. There exist memory management systems that offer $\mathcal{O}(1)$ operations' response times, but do not consider memory fragmentation. In contrast is the Java domain, where memory fragmentation is handled by real-time garbage collected systems. This survey provides an overview of the approaches in both domains with focusing on the administration of memory.

1. Introduction

Dynamic memory management is a fundamental and well studied part of operating systems. This core unit has to keep track of used and unused parts of the memory. Application programs use the dynamic memory management system to allocate and free blocks of arbitrary size in arbitrary order. Memory deallocation can lead to memory holes, which can not be reused by future allocation requests, if they are too small. Dynamic memory management systems have to minimize this problem, called the *fragmentation problem*.

There are two general approaches to perform allocation and deallocation operations in a dynamic memory management system: explicit memory allocation and deallocation, where the application has to explicitly call the corresponding procedures of the dynamic memory management system for allocating and deallocating memory and implicit memory deallocation where allocated memory, which is not used anymore, is detected and freed automatically. The explicit dynamic memory management approaches cover low level implementations in comparison to the implicit dynamic memory management approaches, where more complex Java real-time systems are examined.

2. Dynamic real-time memory management system requirements

Traditional dynamic memory management strategies are typically non-deterministic and have been avoided in the real-time domain. All the used memory of real-time applications was allocated statically. This was a sufficient solution for many real-time controllers. Nowadays, real-time applications are getting more complex and more flexibility is needed. Dynamic memory management systems allow applications to create different types and an arbitrary amount of objects dynamically.

Most of the established dynamic memory management systems are optimized to offer excellent best and average-case response times, but in the worst-case they are unbounded. This is suitable for non real-time systems, but for hard real-time systems tight bounds have to exist. In hard real-time systems all the running tasks have to meet their deadlines. Therefore the timing behaviour of the whole operating system, all the used libraries and certainly the real-time applications have to be totally predictable.

For schedulability analysis, the worst case execution times of the tasks mus be known a priori. Worst cases where the whole memory has to be examined are unacceptable. Operation response times have to be bounded and fast to guarantee the system schedulability.

3. Explicit dynamic memory management systems

The procedures for allocating (`malloc`) and deallocating (`free`) memory have to be called explicitly, if an explicit dynamic memory management system is used. Masmano et al. [8] and Puaut [10] presented in their works evaluations of explicit dynamic memory management systems under real-time loads. This section gives a brief overview of some established allocators. Note that these allocators operate on a single contiguous piece of memory.

The fragmentation problem is not explicitly handled by explicit dynamic memory management systems. This means that memory compaction is not performed. The algorithms just try to minimize fragmentation for general purpose applications, but for arbitrary allocation scenarios fragmentation can be inadmissibly high. This is unacceptable for safety-critical hard real-time systems. Consider a memory of size m and an allocator that allocates n blocks of size s such that $m = s \times n$. Furthermore every block other than the first, the last and the $\frac{m}{n}$-ths block are deallocated. An allocation of a block of size r, with $r > \frac{m}{2}$, can not be allocated anymore. In this case fragmentation is $\frac{3 \times s}{m - 3 \times s}$.

3.1. Doug Lea's allocator

Doug Leas's allocator [6] is a hybrid allocator, which is widely used in several environments. It uses two different types of free lists: There are 48 free lists of the first free list type, which represent exact block sizes (from 16 to 64 bytes), called *fast bins*. The remaining free lists (the second type) are segregated free list, called

bins. Allocation operations are handled by the corresponding free list that fits the allocation request. The allocator uses a delayed coalescing strategy. This means that neighboring free blocks are not coalesced after deallocation operations. Instead a global block coalescing is done, if an allocation requests can not be fulfilled. Therefore deallocation operations are pretty fast and perform in constant time, but the allocation operations offer imprecise bounds, caused by the global delayed free blocks coalesce operations that can occur. Therefore Dough Lea's allocation strategy is not predictable and not suitable for a hard real-time system.

3.2. Half-Fit

Half-Fit [9] groups free blocks in the range of $[2^i, 2^{i+1}[$ into a free list denoted by i. Bitmaps are used to keep track of empty lists and bitmap processor instructions are used to find set bits in constant time. If an allocation request of size s is performed, the search for a suitable free block starts at i, where $i = \lfloor log_2(s-1) \rfloor + 1$ or 0 if $s = 1$. If list i contains no free element, then the next free list $i+1$ is examined. If a free block of a larger size class has to be used, this free block is split into two blocks of sizes r and r', where the free block r' is reinserted into the corresponding free list. Fragmentation is high in the Half-Fit allocator, if many allocations are performed that are not close to power of two.

3.3. Two-Level Segregated Fit

The Two-Level Segregated Fit (TLSF) allocator [7] implements a combination of a segregated free list and a bitmap allocation strategy. The first dimension of the free list is an array that represents size classes that are a power of two apart. The second dimension sub-divides each first-level size class linearly. Each free list array has an associated bitmap where free lists are marked that contain free blocks. Processor instructions are used to find an adequate free memory location for an allocation request in constant time. If there are neighboring free blocks after deallocation operations, then they are immediately coalesced using the *boundary tag* technique [5]. This immediate coalescing technique leads to larger reusable memory ranges and therefore to less fragmentation in comparison to the Half-Fit approach.

3.4. Algorithms Complexity

Table 1 shows the complexity of the allocation and deallocation operations of the three presented explicit dynamic memory management systems. All of them offer constant time behavior except for the allocation operation of Doug Lea's implementation. Let m denote the memory size and s denote the minimum block size than $\mathcal{O}(\frac{m}{s})$ represents the complexity of coalesce operations that can occur, if an allocation call can not be performed.

	Allocation	Deallocation
DLmalloc	$\mathcal{O}(\frac{m}{s})$	$\mathcal{O}(1)$
Half-Fit	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TLSF	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 1. Algorithms Complexity

4. Implicit dynamic memory management systems

An implicit dynamic memory management system is in charge of collecting allocated memory objects that are not in use anymore. Implicit dynamic memory deallocation is known as *garbage collection* and is not addressed in this survey. The garbage collector is responsible for deallocating sufficient unused allocated blocks to handle prospective allocation request of arbitrary size. This survey provieds presents the memory management system concepts of the real-time garbage collected systems.

Two established real-time garbage collected systems are examined: the time-triggered Metronome [1, 2] and the event-triggered Jamaica [12] approach.

4.1. Metronome

In Metronome [1, 2], allocation is performed using segregated free lists. The whole memory is divided into pages of equal size. Each page itself is divided into fixed-size blocks of a particular size. There are n different block sizes which lead to n different size classes. All pages that consist of blocks of the same size build up a size class. Allocation operations are handled by the smallest size class that can fit the allocation request. This is done in constant time. Unused pages can be used by any size class.

The internal fragmentation can be bounded by an adequate ratio between adjacent block sizes. Let c_i denote the size class i and let p denote the maximal acceptable internal fragmentation ratio. Berger et al. [3] proposed the following size classes differences to minimize internal fragmentation: $c_i = \lceil c_{i-1}(1+p) \rceil$.

Compaction operations are performed, if pages of a size class become fragmented to a certain degree due to garbage collection. First of all, the pages of a size class are sorted by the number of unused blocks per page. There is an allocation pointer, which is set to the first not-full page of the resulting list and a compaction pointer, which is set to the last page of the resulting list. Allocated objects are moved from the page, which is referenced by the compaction pointer to the page, which is referenced by the allocation pointer. Compaction is performed until both pointers reference the same page.

Relocation of objects is achieved by using a forwarding pointer. This pointer is located in the header of each object. A Brooks-style read barrier [4] maintains the to-space invariant. A mutator always sees its objects in to-space.

Since Metronome is a time-triggered real-time garbage collector, compaction is part of the collection cycles, which are performed at predefined points in time. It is shown that compaction takes no more than 6% of the collection time. Therefore

the compaction overhead is bounded in the Metronome approach. The remaining time is used to detect allocated objects that are not in use anymore. The duration of the collection interval has to be preset application specific in advance. An improper choice of the duration of the collection interval could lead to missed deadlines or out of memory errors.

4.2. Jamaica

Siebert presented the Jamaica [12, 11] real-time garbage collector, which does not perform compaction operations. A new object model is introduced that is based on fixed size blocks. The whole memory is subdivided into blocks of equal size. Small allocation request can be satisfied by just using a single block. Larger ones require a possibly non-contiguous set of blocks, where each block holds a reference to its successor. The non-contiguity is the reason, why compaction is not necessary anymore. Objects can be build up by arbitrarily distributed blocks, which are connected by a singly-linked list or a tree data structure.

When using blocks of fixed size, the most important decision is to chose an adequate block size. Siebert proposed block sizes in the range of 16 to 64 bytes. This parameter has to be chosen program specific.

The complexity of allocation and deallocation operations depends on the size of the affected object and the used block size. Let s denote the size of an object and let b denote the block size. An object of size s requires $n = \lceil \frac{s}{b} \rceil$ blocks. This means that if an allocation or deallocation operation of an object of size s is performed, n list operations are required. Therefore allocation and deallocations operations are performed in linear time $\mathcal{O}(\frac{s}{b})$, depending on the object size.

Pointer dereferencing can not be done in constant time with the object model of Jamaica. Since an object is build up of non-contiguous blocks, access to the last field of an object requires going through all the blocks of the object, if they are connected via a linked list. Therefore memory dereferencing takes linear time and depends on the location of the field in the object.

Jamaica performs event-triggered garbage collection, which is executed when allocation operations are performed. m blocks have to be examined at every allocation operation to guarantee that all allocation request can be fulfilled. In Jamaica, the amount m of blocks that have to be checked depends on the total amount of allocated blocks. If there are just a few allocated blocks, then less collection has to be performed. Otherwise more work has to be done. In the worst case, if the memory is completely full and an allocation operation is performed, all allocated objects have to be checked. Therefore the collection overhead varies and depends on the global memory state.

5. Conclusion

The presented explicit dynamic memory management systems offer constant response times for allocation and deallocation operations, except for the allocation

operation of Doug Lea's approach. Since TLSF handles fragmentation better than Half-Fit, it is the most applicable candidate for real-time systems. The main problem, which is not considered by these systems is that fragmentation is not handled explicitly. Therefore a case that leads to high fragmentation could be easily created. Such risks are definitively problematic for hard real-time systems.

Metronome and Jamaica, the presented implicit dynamic memory management systems handle fragmentation explicitly. Metronome performs time-triggered garbage collection. The duration of the collection interval, where compaction is performed, has to be precisely chosen to guarantee that the real-time system is able to meet its deadlines and that sufficient memory is always available. Otherwise the system may fail. The event-triggered Jamaica system uses an object model that avoids external fragmentation. Here internal-fragmentation is the problem that has to be minimized by choosing an application adapted block size. The garbage collection overhead varies and depends on the global memory state. Therefore the system is not predictable.

Memory management is the basis for a predictable hard real-time system. None of the outlined dynamic memory management systems offers predictable memory operations in combination with explicit fragmentation elimination and independence of the global memory state. Hard real-time systems require dynamic memory management systems that offer all of these properties.

References

[1] D. Bacon and P. Cheng. The metronome: A simpler approach to garbage collection in real-time systems, 2003.

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92, New York, NY, USA, 2003. ACM Press.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[4] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM Press.

[5] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1973.

[6] Doug Lea. A memory allocator. Unix/Mail/, 6/96, 1996.

[7] M. Masmano, I. Ripoll, A. Crespo, and J. Real. Tlsf: A new dynamic memory allocator for real-time systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. A comparison of memory allocators for real-time applications. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 68–76, New York, NY, USA, 2006. ACM Press.

[9] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, page 21, Washington, DC, USA, 1995. IEEE Computer Society.

[10] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 41, Washington, DC, USA, 2002. IEEE Computer Society.

[11] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 96, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 9–17, New York, NY, USA, 2000. ACM Press.